Business

- Advertising
- Branding
- Business Management
- Business Ethics
- Careers, Jobs & Employment
- Customer Service
- Marketing
- Networking
- Network Marketing
- Pay-Per-Click Advertising
- Presentation
- Public Relations
- Sales
- Sales Management
- Sales Telemarketing
- Sales Training
- Small Business
- Strategic Planning
- Entrepreneur
- Negotiation Tips
- Team Building
- Top Quick Tips

Internet & Businesses Online

- Affiliate Revenue
- Blogging, RSS & Feeds
- Domain Name
- E-Book
- E-commerce
- Email Marketing
- Ezine Marketing
- Ezine Publishing
- Forums & Boards
- Internet Marketing
- Online Auction
- Search Engine Optimization (SEO)
- Spam Blocking
- Streaming Audio & Online Music
- Traffic Building
- Video Streaming
- Web Design
- Web Development
- Web Hosting
- Web Site Promotion

Finance

- Credit
- Currency Trading
- Debt Consolidation
- Debt Relief
- Loan
- Insurance
- Investing
- Mortgage Refinance
- Personal Finance
- Real Estate
- Taxes
- Stocks & Mutual Fund
- Structured Settlements
- Leases & Leasing
- Wealth Building

Communications

- Broadband Internet
- Mobile & Cell Phone
- VOIP
- Video Conferencing
- Satellite TV

Reference & Education

- Book Reviews
- College & University
- Psychology
- Science Articles

Food & Drinks

- Coffee
- Cooking Tips
- Recipes & Food and Drink
- Wine & Spirits

Home & Family

- Crafts & Hobbies
- Elder Care
- Holiday
- Home Improvement
- Home Security
- Interior Design & Decorating
- Landscaping & Gardening
- Babies & Toddler
- Pets
- Parenting
- Pregnancy

News & Society

- Dating
- Divorce
- Marriage & Wedding
- Political
- Relationships
- Religion
- Sexuality

Computers & Technology

- Computer Hardware
- Data Recovery & Computer Backup
- Game
- Internet Security
- Personal Technology
- Software

Arts & Entertainment

- Casino & Gambling
- Humanities
- Humor & Entertainment
- Language
- Music & MP3
- Philosophy
- Photography
- Poetry

Shopping & Product Reviews

- Book Reviews
- Fashion & Style

Health & Fitness

- Acne
- Aerobics & Cardio
- Alternative Medicine
- Beauty Tips
- Depression
- Diabetes
- Exercise & Fitness
- Fitness Equipment
- Hair Loss
- Medicine
- Meditation
- Muscle Building & Bodybuilding
- Nutrition
- Nutritional Supplements
- Weight Loss
- Yoga

Recreation and Sport

- Fishing
- Golf
- Martial Arts
- Motorcycle

Self Improvement & Motivation

- Attraction
- Coaching
- Creativity
- Dealing with Grief & Loss
- Finding Happiness
- Get Organized - Organization
- Leadership
- Motivation
- Inspirational
- Positive Attitude Tips
- Goal Setting
- Innovation
- Spirituality
- Stress Management
- Success
- Time Management

Writing & Speaking

- Article Writing
- Book Marketing
- Copywriting
- Public Speaking
- Writing

Travel & Leisure

- Aviation & Flying
- Cruising & Sailing
- Outdoors
- Vacation Rental

Cancer

- Breast Cancer
- Mesothelioma & Asbestos Cancer

100%
ONE HUNDRED PERCENT
COMPREHENSIVE
AUTHORITATIVE
WHAT YOU NEED
ONE HUNDRED PERCENT

Master JavaScript and DOM concepts with The Evaluator, an exclusive interactive workbench

Capture your target audience with Web sites featuring creative effects and instant interactivity

Optimize your scripts for effective presentation on the newest browser versions

JavaScript™ Bible

5th Edition

Danny Goodman with Michael Morrison
Foreword by Brendan Eich, creator of JavaScript

BONUS CD-ROM
Includes 22 bonus chapters, over 300 ready-to-run HTML documents, nine complete JavaScript real-world applications, a searchable e-book, and much more!

JavaScript™ Bible

5th Edition

Danny Goodman with Michael Morrison
With a foreword by Brendan Eich, JavaScript's creator

Wiley Publishing, Inc.

Praise for Danny Goodman's *JavaScript™ Bible*

"*JavaScript™ Bible* is the definitive resource in JavaScript programming. I am never more than three feet from my copy."

— Steve Reich, CEO, PageCoders

"This book is a must-have for any web developer or programmer."

— Thoma Lile, President, Kanis Technologies, Inc.

"Outstanding book. I would recommend this book to anyone interested in learning to develop advanced Web sites. Mr. Goodman did an excellent job of organizing this book and writing it so that even a beginning programmer can understand it."

— Jason Hensley, Director of Internet Services, NetVoice, Inc.

"Goodman is always great at delivering clear and concise technical books!"

— Dwayne King, Chief Technology Officer, White Horse

"*JavaScript™ Bible* is well worth the money spent!"

— Yen C.Y. Leong, IT Director, Moo Mooltimedia, a member of SmartTransact Group

"A must-have book for any internet developer."

— Uri Fremder, Senior Consultant, TopTier Software

"I love this book! I use it all the time, and it always delivers. It's the only JavaScript book I use!"

— Jason Badger, Web Developer

"Whether you are a professional or a beginner, this is a great book to get."

— Brant Mutch, Web Application Developer, Wells Fargo Card Services, Inc.

"I never thought I'd ever teach programming before reading your book [*JavaScript™ Bible*]. It's so simple to use — the Programming Fundamentals section brought it all back! Thank you for such a wonderful book, and for breaking through my programming block!"

— Susan Sann Mahon, Certified Lotus Instructor, TechNet Training

"Danny Goodman is very good at leading the reader into the subject. *JavaScript™ Bible* has everything we could possibly need."

— Philip Gurdon

"An excellent book that builds solidly from whatever level the reader is at. A book that is both witty and educational."

— Dave Vane

"I continue to use the book on a daily basis and would be lost without it."

— Mike Warner, Founder, Oak Place Productions

"*JavaScript™ Bible* is by *far* the best JavaScript resource I've ever seen (and I've seen quite a few)."

— Robert J. Mirro, Independent Consultant, RJM Consulting

JavaScript™ Bible

5th Edition

Danny Goodman with Michael Morrison
With a foreword by Brendan Eich, JavaScript's creator

Wiley Publishing, Inc.

JavaScript™ Bible, 5th Edition

Published by:
Wiley Publishing, Inc.
10475 Crosspoint Blvd.
Indianapolis, Indiana 46256
www.wiley.com

Copyright © 2004 Danny Goodman. All rights reserved.

Library of Congress Control Number: 2004101606

ISBN: 0-7645-5743-2

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

5B/RY/QS/QU/IN

Published by Wiley Publishing, Inc., Indianapolis, Indiana
Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4447, E-mail: permcoordinator@wiley.com.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: WHILE THE PUBLISHER AND AUTHOR HAVE USED THEIR BEST EFFORTS IN PREPARING THIS BOOK, THEY MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS BOOK AND SPECIFICALLY DISCLAIM ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES REPRESENTATIVES OR WRITTEN SALES MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR YOUR SITUATION. YOU SHOULD CONSULT WITH A PROFESSIONAL WHERE APPROPRIATE. NEITHER THE PUBLISHER NOR AUTHOR SHALL BE LIABLE FOR ANY LOSS OF PROFIT OR ANY OTHER COMMERCIAL DAMAGES, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR OTHER DAMAGES.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Trademarks: Wiley and the Wiley Publishing logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates. JavaScript is a trademark or registered trademark of Sun Microsystems, Inc. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

About the Authors

Danny Goodman is the author of numerous critically acclaimed and best-selling books, including *The Complete HyperCard Handbook*, *Danny Goodman's AppleScript Handbook*, *Dynamic HTML: The Definitive Reference*, and *JavaScript & DHTML Cookbook*. He is a renowned authority and expert teacher of computer scripting languages. His writing style and pedagogy continue to earn praise from readers and teachers around the world. To help keep his finger on the pulse of real-world programming challenges, Goodman frequently lends his touch as consulting programmer and designer to leading-edge World Wide Web and intranet sites from his home base in the San Francisco area.

Michael Morrison is a writer, developer, toy inventor, and author of a variety of books covering topics such as Java, Web scripting, game development, ActiveX, and Pocket PCs. Some of Michael's notable writing projects include *Faster Smarter HTML and XML*, *Teach Yourself XML in 24 Hours*, and *The Complete Idiot's Guide to Java 2*. Michael is also the founder of Stalefish Labs (`www.stalefishlabs.com`), an entertainment company specializing in traditional games and toys.

Credits

Vice President and Executive Group Publisher
Richard Swadley

Vice President and Executive Publisher
Robert Ipsen

Vice President and Publisher
Joseph B. Wikert

Acquisitions Editor
Debra Williams Cauley

Production Editor
Angela Smith

Copy Editor
Kim Cofer

Permissions Editor
Laura Moss

Media Development Specialist
Travis Silvers

Project Coordinator
Ryan Steffen

Graphics and Production Specialists
Karl Brandt
Amanda Carter
Lauren Goddard
Denny Hager
Joyce Haughey
Jennifer Heleine
Michael Kruzil
Heather Ryan
Mary Gillot Virgin

Quality Control Technician
Carl William Pierce

Proofreading and Indexing
TECHBOOKS Production Services

Foreword

As JavaScript's creator, I would like to say a few words about where JavaScript has been, where it is going, and how the book you're holding will help you to make the most of the language.

JavaScript was born out of a desire to let HTML authors write scripts directly in their documents. This may seem obvious now, but in the spring of 1995 it was novel and more than a little at odds with both the conventional wisdom (that HTML should describe static document structure only) and the Next Big Thing (Java applets, which were hyped as the one true way to enliven and extend Web pages). Once I got past these contentions, JavaScript quickly shaped up along the following lines:

- "Java-lite" syntax. Although the "natural language" syntax of HyperTalk was fresh in my mind after a friend lent me *The Complete HyperCard Handbook* by some fellow named Goodman, the Next Big Thing weighed heavier, especially in light of another goal: scripting Java applets. If the scripting language resembled Java, then those programmers who made the jump from JavaScript to Java would welcome similarities in syntax. But insisting on Java's class and type declarations, or on a semicolon after each statement when a line ending would do, were out of the question — scripting for most people is about writing short snippets of code, quickly and without fuss.
- Events for HTML elements. Buttons should have onClick event handlers. Documents load and unload from windows, so windows should have onLoad and onUnload handlers. Users and scripts submit forms: thus the onSubmit handler. Although not initially as flexible as HyperCard's messages (whose handlers inspired the onEvent naming convention), JavaScript events let HTML authors take control of user interaction from remote servers and respond quickly to user gestures and browser actions. With the adoption of the W3C DOM Level 2 event handling recommendations, JavaScript in modern browsers has fully flexible control over events.
- Objects without classes. The Self programming language proved the notion of prototype-based inheritance. For JavaScript, I wanted a single prototype per object (for simplicity and efficiency), based by default on the function called using the new operator (for consonance with Java). To avoid distinguishing constructors from methods from functions, all functions receive the object naming them as the property that was called in the parameter. Although prototypes didn't appear until Navigator 3, they were prefigured in Version 2 by quoted text being treated as an object (the Strong object prototype, to which users could attach methods).
- Generated HTML. Embedding JavaScript in HTML gave rise to a thought: Let the script speak HTML, as if the emitted text and markup were loaded in place of the script itself. The possibilities went beyond automating current or last-modified dates, to computing whole trees of tables where all the repeated structure was rolled up in a scripted loop, while the varying contents to be tabulated came in minimal fashion from JavaScript objects forming a catalog or mini-database.

At first, I thought JavaScript would most often find use in validating input to HTML forms. But before long, I was surprised to see how many Web designers devised compelling applications by way of script-generated HTML and JavaScript objects. It became clear from user demonstration and feedback that Web designers sought to build significant applications quickly and effectively with just a few images, HTML, and JavaScript. Eventually they demanded that the browser support what is now known as "Dynamic HTML" (one fun link: `http://www.javascript-games.org/`).

As legions of Web authors embraced the authoring power of JavaScript, they, in turn, demonstrated the crucial advantages of a scripting environment over old-school application development. Not only were the HTML and JavaScript languages comparatively easy to use, but development did not require the programming expertise needed to light all pixels and handle all events as in a big, traditional application.

The primacy of JavaScript on the Web today vindicates our early belief in the value of a scripting language for HTML authors. By keeping the "pixel-lighting" bar low, HTML with images has made Web designers out of millions of people. By keeping the "event-handling" bar low, JavaScript has helped many thousands of those designers become programmers. Perhaps the ultimate example of Web development's convergence with application development is the Mozilla browser, wherein all of the user-interface and even some custom widgets and modular components are implemented entirely using JavaScript, Cascading Style Sheets (CSS), custom XML-based markup languages, and images.

JavaScript is also a general language, useful apart from HTML and XML. It has been embedded in servers, authoring tools, browser plug-ins, and other kinds of browsers (for such things as 3D graphical worlds). Its international standard, ECMA-262 (ISO 16262), has advanced to a Third Edition. But compared to languages such as Perl and even Java, it is still relatively young. Work toward a Fourth Edition of the language, supporting optional types, classes, and versioning facilities progresses within the ECMA technical committee (see the "JS2" proposal to the ECMA technical committee documented at `http://www.mozilla.org/js/language/js20/`).

It is clear to me that JavaScript would not have survived without a creative, loyal, and patient community of developers; I owe them each a huge debt of thanks. Those developers who took up the beta releases of Navigator 2, and disseminated vital workarounds and feature requests by e-mail and net-news, are the language's godparents. Developer support and feedback continue to make JavaScript the eclectic, rambunctious success it is.

The book in your hands compiles thousands of those "developer miles" with the insight of an expert guide and teacher. Danny didn't know at the time how much inspiration I found in his HyperCard book, but it was on my desk throughout the development of JavaScript in 1995. His energy, compassion, and clear prose helped me keep the goal of "a language for all" in mind. It is enormously gratifying to write the foreword to the fourth edition of this book, which has earned so many "satisfied reader miles."

I highly recommend Danny Goodman's *JavaScript Bible* to anyone who wants to learn JavaScript, and especially to those HTML authors who've so far written only a few scripts or programs — you're in for a lifetime of fun on the "scripting road" with a trusty guide at your side.

Brendan Eich
The Mozilla Organization (http://www.mozilla.org)

Preface

For over 20 years, I have written the books I wished had already been written to help me learn or use a new technology. Whenever possible, I like to get in at the very beginning of a new authoring or programming environment, feel the growing pains, and share with readers the solutions to my struggles. This fifth edition of the *JavaScript™ Bible* represents knowledge and experience accumulated over eight years of daily work in JavaScript and a constant monitoring of newsgroups for questions, problems, and challenges facing scripters at all levels. My goal is to help you avoid the same frustration and head scratching I and others have experienced through multiple generations of scriptable browsers.

While the earliest editions of this book focused on the then predominant Netscape Navigator browser, the swing of the browser market share pendulum currently favors Microsoft Internet Explorer on the Windows platform. But the more important trend is the Web developer community's demand for browser compliance with a growing body of industry standards. Potential conflicts arise when the dominant browser on the Internet does not fully support existing standards, but instead provides proprietary alternatives. The job of a book claiming to be the "bible" is not only to present both the standard and proprietary details when they diverge, but also to show you how to write scripts that blend the two so that they work on as many browsers as possible. Empowering you to design and write good scripts is my passion, regardless of browser. It's true that my bias is toward industry standards, but not to the exclusion of proprietary features that may be necessary to get your content and scripting ideas flowing equally well on today's and tomorrow's browsers.

Organization and Features of This Edition

Like the previous fourth and Gold editions of the *JavaScript Bible*, this fifth edition contains far more information than can be printed and bound into a single volume. The complete contents can be found in the electronic version of this book (in Adobe Acrobat form) on the CD-ROM that accompanies the book. This new edition is structured in such a way as to supply the most commonly needed information in its entirety in the printed portion of the book. Thus, numerous complete code listings, which had been diverted to CD-ROM in the previous editions, are now readily available in the printed pages. Content that you use to learn the fundamentals of JavaScript and reference frequently are at your fingertips in the printed version, while chapters with advanced content are in the searchable electronic version on the CD-ROM. Here are some details about the book's structure.

Part I

Part I of the book begins with a chapter that shows how JavaScript compares with Java and discusses its role within the rest of the World Wide Web. The Web browser and scripting world have undergone significant changes since JavaScript first arrived on the scene. That's why Chapter 2 is devoted to addressing challenges facing scripters who must develop applications for both single- and cross-platform browser audiences amid rapidly changing standards efforts. Chapter 3 provides the first foray into JavaScript, where you get to write your first practical script.

Part II

All of Part II is handed over to a tutorial for newcomers to JavaScript. Nine lessons provide you with a gradual path through browser internals, basic programming skills, and genuine browser scripting with an emphasis on industry standards as supported by most of the scriptable browsers in use today. Exercises follow at the end of each lesson to help reinforce what you just learned and challenge you to use your new knowledge (you'll find answers to the exercises in Appendix C). The goal of the tutorial is to equip you with sufficient experience to start scripting simple pages right away while making it easier for you to understand the in-depth discussions and examples in the rest of the book. By the end of the final lesson, you'll know how to create the mouse-rollover image swapping effect that is popular in a lot of Web pages these days, and modify the content of a Web page dynamically.

Part III

Part III, the largest section of the book, provides in-depth coverage of the document object models as implemented in today's browsers. In all reference chapters, a compatibility chart indicates the browser version that supports each object and object feature. One chapter in particular, Chapter 15, contains reference material that is shared by most of the remaining chapters of Part III. To help you refer back to Chapter 15 from other chapters, a dark tab along the outside edge of the page shows you at a glance where the chapter is located. Additional navigation aids include guide words near the tops of most pages to indicate which object and object feature is covered on the page.

Part IV

Reference information for the core JavaScript language fills Part IV. As with reference chapters of Part III, the JavaScript chapters display browser compatibility charts for every JavaScript language term. Guide words near the tops of pages help you find a particular term quickly.

Part V

Several appendixes at the end of the book provide helpful reference information. These resources include a JavaScript and Browser Objects Quick Reference in Appendix A, a list of JavaScript reserved words in Appendix B, answers to Part II's tutorial exercises in Appendix C, and Internet resources in Appendix D. In Appendix E, you also find information on using the CD-ROM that comes with this book, which includes numerous bonus chapters and examples.

CD-ROM

The CD-ROM is a gold mine of information. It begins with an Adobe Acrobat (PDF) version of the entire contents of this fifth edition of the *JavaScript Bible*. This version includes not only the unprinted advanced material from Parts III and IV, but also 23 bonus chapters covering:

- ✦ Advanced DOM, XML, and JavaScript objects
- ✦ Dynamic HTML, data validation, plug-ins, and security
- ✦ Techniques for developing and debugging professional Web-based applications
- ✦ Nine full-fledged JavaScript real-world applications

Another treasure trove on the CD-ROM is the Listings folder where you'll find over 300 ready-to-run HTML documents that serve as examples of most of the document object model and JavaScript vocabulary words in Parts III and IV. All of the bonus chapter example listings are also included. You can run these examples with your JavaScript-enabled browser, but be sure to use the `index.html` page in the Listings folder as a gateway to running the listings. This page shows you the browsers that are compatible with each example listing. I could have provided you with humorous little sample code fragments out of context, but I think that seeing full-fledged HTML documents (simple though they may be) for employing these concepts is important. I intentionally omitted the script listings from the tutorial part (Part II) of this book to encourage you to type the scripts. I believe you learn a lot, even by aping listings from the book, as you get used to the rhythms of typing scripts in documents.

Be sure to check out the Chapter 13 listing file called `evaluator.html`. Many segments of Parts III and IV invite you to try out an object model or language feature with the help of an interactive workbench, called The Evaluator — a *JavaScript Bible* exclusive! You see instant results and quickly learn how the feature works.

The Quick Reference from Appendix A is in PDF format on the CD-ROM for you to print out and assemble as a handy reference, if desired. Adobe Acrobat Reader is also included on the CD-ROM, in case you don't already have it, so that you can read both of these PDF files.

Prerequisites to Learning JavaScript

Although this book doesn't demand that you have a great deal of programming experience behind you, the more Web pages you've created with HTML, the easier you will find it to understand how JavaScript interacts with the familiar elements you normally place in your pages. Occasionally, you will need to modify HTML tags to take advantage of scripting. If you are familiar with those tags already, the JavaScript enhancements will be simple to digest.

Forms and their control elements (text fields, buttons, and selection lists) play an especially important role in much of typical JavaScript work. You should be familiar with these elements and their HTML attributes. Fortunately, you won't need to know about server scripting or passing information from a form to a server. The focus here is on client-side scripting, which operates independently of the server after the JavaScript-enhanced HTML page is fully loaded into the browser.

The basic vocabulary of the current HTML standard should be part of your working knowledge. You should also be familiar with some of the latest document markup standards, such as XHTML and Cascading Style Sheets (CSS). You don't need to be an expert, by any means. Web searches for these terms will uncover numerous tutorials on the subjects.

If you've never programmed before

To someone who learned HTML from a slim guidebook a few years ago, the size of this book must be daunting. JavaScript may not be the easiest language in the world to learn, but believe me, it's a far cry from having to learn a full programming language, such as Java or C. Unlike developing a full-fledged monolithic application (such as the productivity programs you buy in the stores), JavaScript lets you experiment by writing small snippets of program code to accomplish big things. The JavaScript interpreter built into every scriptable browser does a great deal of the technical work for you.

Programming, at its most basic level, consists of nothing more than writing a series of instructions for the computer to follow. We humans follow instructions all the time, even if we don't realize it. Traveling to a friend's house is a sequence of small instructions: Go three blocks that way; turn left here; turn right there. Amid these instructions are some decisions that we have to make: If the stoplight is red, then stop; if the light is green, then go; if the light is yellow, then floor it. Occasionally, we must repeat some operations several times (kind of like having to go around the block until a parking space opens up). A computer program not only contains the main sequence of steps, but it also anticipates what decisions or repetitions may be needed to accomplish the program's goal (such as how to handle the various states of a stoplight or what to do if someone just stole the parking spot you were aiming for).

The initial hurdle of learning to program is becoming comfortable with the way a programming language wants its words and numbers organized in these instructions. Such rules are called syntax, the same as in a living language. Because computers generally are dumb electronic hulks, they aren't very forgiving if you don't communicate with them in the specific language they understand. When speaking to another human, you can flub a sentence's syntax and still have a good chance of the other person's understanding you fully. Not so with computer programming languages. If the syntax isn't perfect (or at least within the language's range of knowledge that it can correct), the computer has the brazenness to tell you that you have made a syntax error.

The best thing you can do is to just chalk up the syntax errors you receive as learning experiences. Even experienced programmers make them. Every syntax error you get — and every resolution of that error made by rewriting the wayward statement — adds to your knowledge of the language.

If you've done a little programming before

Programming experience in a procedural language, such as BASIC, may almost be a hindrance rather than a help to learning JavaScript. Although you may have an appreciation for precision in syntax, the overall concept of how a program fits into the world is probably radically different from JavaScript. Part of this has to do with the typical tasks a script performs (carrying out a very specific task in response to user action within a Web page), but a large part also has to do with the nature of object-oriented programming.

In a typical procedural program, the programmer is responsible for everything that appears on the screen and everything that happens under the hood. When the program first runs, a great deal of code is dedicated to setting up the visual environment. Perhaps the screen contains several text entry fields or clickable buttons. To determine which button a user clicks, the program examines the coordinates of the click and compares those coordinates against a list of all button coordinates on the screen. Program execution then branches out to perform the instructions reserved for clicking in that space.

Object-oriented programming is almost the inverse of that process. A button is considered an object — something tangible. An object has properties, such as its label, size, alignment, and so on. An object may also contain a script. At the same time, the system software and browser, working together, can send a message to an object — depending on what the user does — to trigger the script. For example, if a user clicks in a text entry field, the system/browser tells the field that somebody has clicked there (that is, has set the focus to that field), giving the field the task of deciding what to do about it. That's where the script comes in. The script is connected to the field, and it contains the instructions that the field carries out after the user activates it. Another set of instructions may control what happens when the user types an entry and tabs or clicks out of the field, thereby changing the content of the field.

Some of the scripts you write may seem to be procedural in construction: They contain a simple list of instructions that are carried out in order. But when dealing with data from form elements, these instructions work with the object-based nature of JavaScript. The form is an object; each radio button or text field is an object as well. The script then acts on the properties of those objects to get some work done.

Making the transition from procedural to object-oriented programming may be the most difficult challenge for you. When I was first introduced to object-oriented programming a number of years ago, I didn't get it at first. But when the concept clicked — a long, pensive walk helped — so many light bulbs went on inside my head that I thought I might glow in the dark. From then on, object orientation seemed to be the only sensible way to program.

If you've programmed in C before

By borrowing syntax from Java (which, in turn, is derived from C and C++), JavaScript shares many syntactical characteristics with C. Programmers familiar with C will feel right at home. Operator symbols, conditional structures, and repeat loops follow very much in the C tradition. You will be less concerned about data types in JavaScript than you are in C. In JavaScript, a variable is not restricted to any particular data type.

With so much of JavaScript's syntax familiar to you, you will be able to concentrate on document object model concepts, which may be entirely new to you. You will still need a good grounding in HTML to put your expertise to work in JavaScript.

If you've programmed in Java before

Despite the similarity in their names, the two languages share only surface aspects: loop and conditional constructions, C-like "dot" object references, curly braces for grouping statements, several keywords, and a few other attributes. Variable declarations, however, are quite different, because JavaScript is a loosely typed language. A variable can contain an integer value in one statement and a string in the next (though I'm not saying that this is good style). What Java refers to as methods, JavaScript calls methods (when associated with a predefined

object) or functions (for scripter-defined actions). JavaScript methods and functions may return values of any type without having to state the data type ahead of time.

Perhaps the most important aspects of Java to suppress when writing JavaScript are the object-oriented notions of classes, inheritance, instantiation, and message passing. These aspects are simply non-issues when scripting. At the same time, however, JavaScript's designers knew that you'd have some hard-to-break habits. For example, although JavaScript does not require a semicolon at the end of each statement line, if you type one in your JavaScript source code, the JavaScript interpreter won't balk.

If you've written scripts (or macros) before

Experience with writing scripts in other authoring tools or macros in productivity programs is helpful for grasping a number of JavaScript's concepts. Perhaps the most important concept is the idea of combining a handful of statements to perform a specific task on some data. For example, you can write a macro in Microsoft Excel that performs a data transformation on daily figures that come in from a corporate financial report on another computer. The macro is built into the Macro menu, and you run it by choosing that menu item whenever a new set of figures arrives.

Some modern programming environments, such as Visual Basic, resemble scripting environments in some ways. They present the programmer with an interface builder, which does most of the work or displaying screen objects with which the user will interact. A big part of the programmer's job is to write little bits of code that are executed when a user interacts with those objects. A great deal of the scripting you will do with JavaScript matches that pattern exactly. In fact, those environments resemble the scriptable browser environment in another way: They provide a finite set of predefined objects that have fixed sets of properties and behaviors. This predictability makes learning the entire environment and planning an application easier to accomplish.

Formatting and Naming Conventions

The script listings and words in this book are presented in a `monospace font` to set them apart from the rest of the text. Because of restrictions in page width, lines of script listings may, from time to time, break unnaturally. In such cases, the remainder of the script appears in the following line, flush with the left margin of the listing, just as they would appear in a text editor with word wrapping turned on. If these line breaks cause you problems when you type a script listing into a document yourself, I encourage you to access the corresponding listing on the CD-ROM to see how it should look when you type it.

As soon as you reach Part III of this book, you won't likely go for more than a page before reading about an object model or language feature that requires a specific minimum version of one browser or another. To make it easier to spot in the text when a particular browser and browser version is required, most browser references consist of an abbreviation and a version number. For example, WinIE5 means Internet Explorer 5 for Windows; NN6 means Netscape Navigator 6 for any operating system; Moz stands for the relatively new browsers now under the stewardship of The Mozilla Foundation; and Safari is Apple's own browser for MacOS X. If a feature is introduced with a particular version of browser and is supported in subsequent versions, a plus symbol (+) follows the number. For example, a feature marked WinIE5.5+ indicates that Internet Explorer 5.5 for Windows is required at a minimum, but the feature is also available in

WinIE6 and probably future WinIE versions. Occasionally, a feature or some highlighted behavior applies to only one browser. For example, a feature marked NN4 means that it works only in Netscape Navigator 4.x. A minus sign (e.g., WinIE-) means that the browser does not support the item being discussed.

The format of HTML and code listings in this edition has changed radically from previous editions. To emphasize the Web developer community's trend toward the latest Web standards, the HTML examples in this edition follow XHTML coding conventions, which dictate all-lowercase tag and attribute names, as well as self-closing tags that do not act as containers (such as the XHTML `<br />` tag in place of the HTML `<br>` tag). Another pervasive style change is the explicit inclusion of semicolons at the end of JavaScript statement lines. Semicolons are still optional in the language, but I have changed my personal coding style, as demonstrated throughout this book.

Note, Tip, and Caution icons occasionally appear in the book to flag important points.

Acknowledgments

Bringing this 1750-page edition up to the current state of the art was a task greater than one author could possibly handle in a timely manner. While I busied myself with updating the tutorial and validating terminology and listings with the latest browsers, I was fortunate to have the tireless help of Michael Morrison — a first-rate author in his own right — who carefully pruned obsolete content, modernized tens of thousands of lines of code, restored linearity to the previous edition's maze, and kept the project on track at a breakneck pace. David Wall, a long-time *JavaScript Bible* helper, thankfully contributed his time and expertise to this effort as well. Many thanks to the hard-working folks at Wiley Publishing: Debra Williams Cauley, Mary Beth Wakefield, Angela Smith. Above all, I want to thank the many readers of the earlier editions of this book for investing in this ongoing effort. I wish I had the space here to acknowledge by name so many who have sent e-mail notes and suggestions: Your input has been most welcome and greatly appreciated.

Contents at a Glance

Foreword vii
Preface ix
Acknowledgments xvii

PART I: Getting Started with JavaScript 1
Chapter 1: JavaScript's Role in the World Wide Web and Beyond 3
Chapter 2: Authoring Challenges Amid the Browser Wars 9
Chapter 3: Your First JavaScript Script 17

PART II: JavaScript Tutorial 25
Chapter 4: Browser and Document Objects 27
Chapter 5: Scripts and HTML Documents 47
Chapter 6: Programming Fundamentals, Part I 59
Chapter 7: Programming Fundamentals, Part II 69
Chapter 8: Window and Document Objects 83
Chapter 9: Forms and Form Elements 97
Chapter 10: Strings, Math, and Dates 111
Chapter 11: Scripting Frames and Multiple Windows 121
Chapter 12: Images and Dynamic HTML 131

PART III: Document Objects Reference 141
Chapter 13: JavaScript Essentials 143
Chapter 14: Document Object Model Essentials 163
Chapter 15: Generic HTML Element Objects 201
Chapter 16: Window and Frame Objects 359
Chapter 17: Location and History Objects 485
Chapter 18: The Document and Body Objects 509
Chapter 19: Link and Anchor Objects 591
Chapter 20: Image, Area, and Map Objects 601
Chapter 21: The Form and Related Objects 629
Chapter 22: Button Objects 651
Chapter 23: Text-Related Form Objects 673
Chapter 24: Select, Option, and FileUpload Objects 695
Chapter 25: Event Objects 719
Chapter 26: Style Sheet and Style Objects 793

PART IV: JavaScript Core Language Reference 847
Chapter 27: The String Object 849
Chapter 28: The Math, Number, and Boolean Objects 877
Chapter 29: The Date Object 891
Chapter 30: The Array Object 909
Chapter 31: Control Structures and Exception Handling 931
Chapter 32: JavaScript Operators 961
Chapter 33: Functions and Custom Objects 981
Chapter 34: Global Functions and Statements 1009
Chapter 35: Body Text Objects 1023

PART V: Appendixes 1105
Appendix A: JavaScript and Browser Object Quick Reference 1107
Appendix B: JavaScript Reserved Words 1121
Appendix C: Answers to Tutorial Exercises 1123
Appendix D: JavaScript and DOM Internet Resources 1139
Appendix E: What's on the CD-ROM 1143

Index 1147
End-User License Agreement 1243

PART VI: Bonus Chapters . . . . . On the CD-ROM
Chapter 36: HTML Directive Objects
Chapter 37: Table and List Objects
Chapter 38: The Navigator and Other Environment Objects
Chapter 39: Positioned Objects
Chapter 40: Embedded Objects
Chapter 41: XML Objects
Chapter 42: The Regular Expression and RegExp Objects
Chapter 43: Data-Entry Validation
Chapter 44: Scripting Java Applets and Plug-Ins
Chapter 45: Debugging Scripts
Chapter 46: Security and Netscape Signed Scripts
Chapter 47: Cross-Browser Dynamic HTML Issues
Chapter 48: Internet Explorer Behaviors
Chapter 49: Application: Tables and Calendars

Chapter 50: Application: A Lookup Table
Chapter 51: Application: A “Poor Man’s” Order Form
Chapter 52: Application: Outline-Style Table of Contents
Chapter 53: Application: Calculations and Graphics
Chapter 54: Application: Intelligent “Updated” Flags
Chapter 55: Application: Decision Helper
Chapter 56: Application: Cross-Browser DHTML Map Puzzle
Chapter 57: Application: Transforming XML Data

Contents

Foreword vii
Preface ix
Acknowledgments xvii

Part I: Getting Started with JavaScript 1

Chapter 1: JavaScript's Role in the World Wide Web and Beyond 3

Competing for Web Traffic 4
Other Web Technologies 4
JavaScript: A Language for All 6
JavaScript: The Right Tool for the Right Job 8

Chapter 2: Authoring Challenges Amid the Browser Wars 9

Leapfrog 9
Duck and Cover 10
Compatibility Issues Today 11
Developing a Scripting Strategy 14

Chapter 3: Your First JavaScript Script 17

The Software Tools 17
Setting Up Your Authoring Environment 18
What Your First Script Will Do 20
Entering Your First Script 21
Examining the Script 22
Have Some Fun 24

Part II: JavaScript Tutorial 25

Chapter 4: Browser and Document Objects 27

Scripts Run the Show 27
JavaScript in Action 28
The Document Object Model 34
When a Document Loads 36
Object References 39
Node Terminology 41
What Defines an Object? 42
Exercises 46

Chapter 5: Scripts and HTML Documents 47

Where Scripts Go in Documents 47
JavaScript Statements 51
When Script Statements Execute 51
Viewing Script Errors 54
Scripting versus Programming 55
Exercises 56

Chapter 6: Programming Fundamentals, Part I 59

What Language Is This? 59
Working with Information 59
Variables 60
Expressions and Evaluation 62
Data Type Conversions 64
Operators 66
Exercises 67

Chapter 7: Programming Fundamentals, Part II 69

Decisions and Loops 69
Control Structures 70
About Repeat Loops 71
Functions 72
About Curly Braces 76
Arrays 76
Exercises 80

Chapter 8: Window and Document Objects 83

Top-Level Objects 83
The window Object 84
Window Properties and Methods 87
The location Object 89
The navigator Object 90
The document Object 90
Exercises 95

Chapter 9: Forms and Form Elements 97

The form Object 97
Form Controls as Objects 100
The Button Object 102
The Checkbox Object 102
The Radio Object 103
The select Object 104
Passing Form Data and Elements to Functions 106
Submitting and Prevalidating Forms 108
Exercises 110

Chapter 10: Strings, Math, and Dates 111

Core Language Objects 111
String Objects 111
The Math Object 115
The Date Object 116
Date Calculations 117
Exercises 119

Chapter 11: Scripting Frames and Multiple Windows 121

Frames: Parents and Children 121
References among Family Members 123
Frame Scripting Tips 125
About iframe Elements 125
Controlling Multiple Frames — Navigation Bars 126
References for Multiple Windows 128
Exercises 130

Chapter 12: Images and Dynamic HTML 131

The Image Object 131
The javascript: Pseudo-URL 137
Popular Dynamic HTML Techniques 137
Exercises 140

Part III: Document Objects Reference 141

Chapter 13: JavaScript Essentials 143

JavaScript Versions 143
Core Language Standard — ECMAScript 144
Embedding Scripts in HTML Documents 144
Browser Version Detection 148
Designing for Compatibility 155
Language Essentials for Experienced Programmers 158
Onward to Object Models 161

Chapter 14: Document Object Model Essentials 163

The Object Model Hierarchy 163
How Document Objects Are Born 165
Object Properties 166
Object Methods 167
Object Event Handlers 168
Object Model Smorgasbord 169
Basic Object Model 170
Basic Object Model Plus Images 171
Navigator 4–Only Extensions 171
Internet Explorer 4+ Extensions 173
Internet Explorer 5+ Extensions 177
The W3C DOM 177

Mixing Object Models . . . 192
Standards Compatibility Modes (DOCTYPE Switching) . . . 198
Where to Go from Here . . . 199

Chapter 15: Generic HTML Element Objects . . . 201

Generic Objects . . . 201

Chapter 16: Window and Frame Objects . . . 359

Window Terminology . . . 359
Frames . . . 359
window Object . . . 366
frame Element Object . . . 462
frameset Element Object . . . 468
iframe Element Object . . . 474
popup Object . . . 480

Chapter 17: Location and History Objects . . . 485

location Object . . . 485
history Object . . . 501

Chapter 18: The Document and Body Objects . . . 509

document Object . . . 510
body Element Object . . . 576
TreeWalker Object . . . 586

Chapter 19: Link and Anchor Objects . . . 591

Anchor, Link, and a Element Objects . . . 592

Chapter 20: Image, Area, and Map Objects . . . 601

Image and img Element Objects . . . 601
area Element Object . . . 621
map Element Object . . . 624

Chapter 21: The Form and Related Objects . . . 629

The Form in the Object Hierarchy . . . 629
form Object . . . 630
fieldset and legend Element Objects . . . 646
label Element Object . . . 648

Chapter 22: Button Objects . . . 651

The button Element Object, and the Button, Submit, and Reset Input Objects . . . 651
checkbox Input Object . . . 656
radio Input Object . . . 663
image Input Object . . . 669

Chapter 23: Text-Related Form Objects 673

Text Input Object 674
password Input Object 689
hidden Input Object 689
textarea Element Object 690

Chapter 24: Select, Option, and FileUpload Objects 695

select Element Object 695
option Element Object 713
optgroup Element Object 714
file Input Element Object 717

Chapter 25: Event Objects 719

Why "Events"? 719
Event Propagation 721
Referencing the event object 740
event Object Compatibility 741
Dueling Event Models 742
Event Types 745
NN4 event Object 747
IE4+ event Object 753
NN6+/Moz/Safari event Object 773

Chapter 26: Style Sheet and Style Objects 793

Making Sense of the Object Names 794
Imported Stylesheets 795
Reading Style Properties 795
style Element Object 796
styleSheet Object 798
cssRule and rule Objects 807
currentStyle , runtimeStyle , and style Objects 810
filter Object 837

Part IV: JavaScript Core Language Reference 847

Chapter 27: The String Object 849

String and Number Data Types 849
String Object 851
String Utility Functions 873
URL String Encoding and Decoding 876

Chapter 28: The Math, Number, and Boolean Objects 877

Numbers in JavaScript 877
Math Object 883
Number Object 886
Boolean Object 890

Chapter 29: The Date Object 891

Time Zones and GMT 891
The Date Object 892
Validating Date Entries in Forms 904

Chapter 30: The Array Object 909

Structured Data 909
Creating an Empty Array 910
Populating an Array 911
JavaScript Array Creation Enhancements 911
Deleting Array Entries 912
Parallel Arrays 912
Multidimensional Arrays 915
Simulating a Hash Table 916
Array Object Properties 916
Array Object Methods 918

Chapter 31: Control Structures and Exception Handling 931

If and If. . .Else Decisions 931
Conditional Expressions 936
Repeat (for) Loops 936
The while Loop 940
The do-while Loop 942
Looping through Properties (for-in) 942
The with Statement 943
Labeled Statements 944
The switch Statement 946
Exception Handling 948
Using try-catch-finally Constructions 950
Throwing Exceptions 953
Error Object 957

Chapter 32: JavaScript Operators 961

Operator Categories 961
Comparison Operators 962
Equality of Disparate Data Types 963
Connubial Operators 965
Assignment Operators 967
Boolean Operators 968
Bitwise Operators 972
Object Operators 973
Miscellaneous Operators 976
Operator Precedence 978

Chapter 33: Functions and Custom Objects 981

Function Object 981
Function Application Notes 988
Custom Objects 994

Object-Oriented Concepts . . . 1003
Object Object . . . 1006

Chapter 34: Global Functions and Statements . . . 1009

Functions . . . 1010
Statements . . . 1016

Chapter 35: Body Text Objects . . . 1023

blockquote and q Element Objects . . . 1023
br Element Object . . . 1024
font Element Object . . . 1025
h1...h6 Element Objects . . . 1028
hr Element Object . . . 1029
label Element Object . . . 1033
marquee Element Object . . . 1034
Methods . . . 1038
Event Handlers . . . 1039
Range Object . . . 1039
selection Object . . . 1061
Text and TextNode Objects . . . 1068
TextRange Object . . . 1074
TextRectangle Object . . . 1101

Part V: Appendixes 1105

Appendix A: JavaScript and Browser Object Quick Reference . . . 1107

Appendix B: JavaScript Reserved Words . . . 1121

Appendix C: Answers to Tutorial Exercises . . . 1123

Appendix D: JavaScript and DOM Internet Resources . . . 1139

Appendix E: What's on the CD-ROM . . . 1143

Index . . . 1147
End-User License Agreement . . . 1243

Part VI: Bonus Chapters On the CD-ROM

Chapter 36: HTML Directive Objects

Chapter 37: Table and List Objects

Chapter 38: The Navigator and Other Environment Objects

Chapter 39: Positioned Objects

Chapter 40: Embedded Objects

Chapter 41: XML Objects

Chapter 42: The Regular Expression and RegExp Objects

Chapter 43: Data-Entry Validation

Chapter 44: Scripting Java Applets and Plug-Ins

Chapter 45: Debugging Scripts

Chapter 46: Security and Netscape Signed Scripts

Chapter 47: Cross-Browser Dynamic HTML Issues

Chapter 48: Internet Explorer Behaviors

Chapter 49: Application: Tables and Calendars

Chapter 50: Application: A Lookup Table

Chapter 51: Application: A "Poor Man's" Order Form

Chapter 52: Application: Outline-Style Table of Contents

Chapter 53: Application: Calculations and Graphics

Chapter 54: Application: Intelligent "Updated" Flags

Chapter 55: Application: Decision Helper

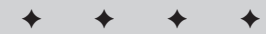
Chapter 56: Application: Cross-Browser DHTML Map Puzzle

Chapter 57: Application: Transforming XML Data

P A R T

I

Getting Started with JavaScript

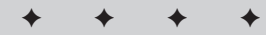


In This Part

Chapter 1
JavaScript's Role in the World Wide Web and Beyond

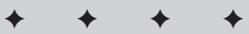
Chapter 2
Authoring Challenges Amid the Browser Wars

Chapter 3
Your First JavaScript Script



CHAPTER 1

JavaScript's Role in the World Wide Web and Beyond

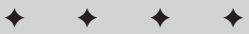


In This Chapter

How JavaScript blends with other Web-authoring technologies

The history of JavaScript

What kinds of jobs you should and should not entrust to JavaScript



Many of the technologies that make the World Wide Web possible have far exceeded their original goals. Envisioned at the outset as a medium for publishing static text and image content across a network, the Web is forever being probed, pushed, and pulled by content authors. By taking for granted so much of the "dirty work" of conveying the bits between server and client computers, content developers and programmers dream of exploiting that connection to generate new user experiences and practical applications. It's not uncommon for a developer community to take ownership of a technology and mold it to do new and exciting things. But with so many Web technologies — especially browser programming with JavaScript — being within reach of everyday folks, we have witnessed an unprecedented explosion in turning the World Wide Web from a bland publishing medium into a highly interactive, operating system–agnostic authoring platform.

The JavaScript language, working in tandem with related browser features, is a Web-enhancing technology. When employed on the client computer, the language can help turn a static page of content into an engaging, interactive, and intelligent experience. Applications can be as subtle as welcoming a site's visitor with the greeting "Good morning!" when it is morning in the client computer's time zone — even though it is dinnertime where the server is located. Or applications can be much more obvious, such as delivering the content of a slide show in a one-page download while JavaScript controls the sequence of hiding, showing, and "flying slide" transitions while navigating through the presentation.

Of course, JavaScript is not the only technology that can give life to drab Web content. Therefore, it is important to understand where JavaScript fits within the array of standards, tools, and other technologies at your disposal. The alternative technologies described in this chapter are HTML, Cascading Style Sheets (CSS), server programs, and plug-ins. In most cases, JavaScript can work side by side with these other technologies, even though the hype around some make them sound like one-stop shopping places for all your interactive needs. That's rarely the case. Finally, you learn about the origins of JavaScript and what role it plays in today's advanced Web browsers.

Competing for Web Traffic

Web page publishers revel in logging as many visits to their sites as possible. Regardless of the questionable accuracy of Web page *hit* counts, a site consistently logging 10,000 dubious hits per week is clearly far more popular than one with 1,000 dubious hits per week. Even if the precise number is unknown, relative popularity is a valuable measure. Another useful number is how many links from outside pages lead to a site. A popular site will have many other sites pointing to it — a key to earning high visibility in Web searches.

Encouraging people to visit a site frequently is the Holy Grail of Web publishing. Competition for viewers is enormous. Not only is the Web like a fifty million–channel television, but the Web competes for viewers' attention with all kinds of computer-generated information. That includes anything that appears onscreen as interactive multimedia.

Users of entertainment programs, multimedia encyclopedias, and other colorful, engaging, and mouse finger-numbing actions are accustomed to high-quality presentations. Frequently, these programs sport first-rate graphics, animation, live-action video, and synchronized sound. In contrast, the lowest common denominator Web page has little in the way of razzle-dazzle. Even with the help of Dynamic HTML and stylesheets, the layout of pictures and text is highly constrained compared with the kinds of desktop publishing documents you see all the time. Regardless of the quality of its content, an unscripted, vanilla HTML document is flat. At best, interaction is limited to whatever navigation the author offers in the way of hypertext links or forms whose filled-in content magically disappears into the Web site's server.

Other Web Technologies

With so many ways to spice up Web sites and pages, you can count on competitors for your site's visitors to do their darndest to make their sites more engaging than yours. Unless you are the sole purveyor of information that is in high demand, you continually must devise ways to keep your visitors coming back and entice new ones. If you design for an intranet, your competition is the drive for improved productivity by colleagues who use the internal Web sites for getting their jobs done.

These are all excellent reasons why you should care about using one or more Web technologies to raise your pages above the noise. Let's look at the major technologies you should know about.

Hypertext Markup Language (HTML and XHTML)

As an outgrowth of *SGML (Standard Generalized Markup Language)*, *HTML* is generally viewed as nothing more than a document formatting, or *tagging,* language. The tags (inside <> delimiter characters) instruct a viewer program (the *browser* or, more generically, the *client*) how to display chunks of text or images.

Relegating HTML to the category of a tagging language does disservice not only to the effort that goes into fashioning a first-rate Web page, but also to the way users interact with the pages. To my way of thinking, any collection of commands and other syntax that directs the way users interact with digital information is *programming*. With HTML, a Web page author controls the user experience with the content just as the engineers who program Microsoft Excel craft the way users interact with spreadsheet content and functions.

Version 4.0 and later of the published HTML standards endeavor to define the purpose of HTML as assigning context to content, leaving the appearance to a separate standard for stylesheets. In other words, it's not HTML's role to signify that some text is italic, but rather to signify *why*

it is italic. For example, you would tag a chunk of text that conveys emphasis (via the tag) regardless of how the stylesheet or browser sets the appearance of that emphasized text.

XHTML is a more recent adaptation of HTML that adheres to stylistic conventions established by the XML (eXtensible Markup Language) standard. No new tags come with XHTML, but it reinforces the notion of tagging to denote a document's structure and content.

Cascading Style Sheets (CSS)

Specifying the look and feel of a Web page via stylesheets is a major trend taking over the modern Web. The basic idea is that given a document's structure spelled out by its HTML or XHTML, a stylesheet defines the layout, colors, fonts, and other visual characteristics to present the content. Applying a different set of CSS definitions to the same document can make it look entirely different, even though the words and images are the same.

Mastery of the fine points of CSS takes time and experimentation, but the results are worth the effort. The days of using HTML tables and transparent "spacer" images to generate elaborate multicolumn layouts are on the wane. Every Web developer should have a solid grounding in CSS.

Server scripting

Web sites that rely on database access or change their content very frequently incorporate programming on the server that generates the HTML output for browsers and/or processes forms that site visitors fill out on the page. Even submissions from a simple login or search form ultimately trigger some server process that sends the results to your browser. Server programming takes on many guises, the names of which you may recognize from your surfing through Web development sites. PHP, ASP, .Net, JSP, and Coldfusion are among the most popular. Associated programming languages include Perl, Python, Java, C++, C#, Visual Basic, and even server-side JavaScript in some environments.

Whatever language you use, the job definitely requires the Web page author to be in control of the server, including whatever *back-end* programs (such as databases) are needed to supply results or massage the information coming from the user. Even with the new, server-based Web site design tools available, server scripting often is a task that a content-oriented HTML author will need to hand off to a more experienced programmer.

As powerful and useful as server scripting can be, it does a poor job of facilitating interactivity in a Web page. Without the help of browser scripting, each change to a page must be processed on the server, causing delays for the visitor and an extra burden on the server for simple tasks. This wastes desktop processing horsepower, especially if the process running on the server doesn't need to access big databases or other external computers.

Of helpers and plug-ins

In the early days of the World Wide Web, a browser needed to present only a few kinds of data before a user's eyes. The power to render text (tagged with HTML) and images (in popular formats such as GIF and JPEG) was built into browsers intended for desktop operating systems. Not wanting to be limited by those data types, developers worked hard to extend browsers so that data in other formats could be rendered on the client computer. It was unlikely, however, that a browser would ever be built that could download and render, say, any of several sound file formats.

One way to solve the problem was to allow the browser, upon recognizing an incoming file of a particular type, to launch a separate application on the client machine to render the

content. As long as this helper application was installed on the client computer (and the association with the helper program set in the browser's preferences), the browser would launch the program and send the incoming file to that program. Thus, you might have one helper application for a MIDI sound file and another for an animation file.

Beginning with Netscape Navigator 2 in early 1996, software *plug-ins* for browsers enabled developers to extend the capabilities of the browser without having to modify the browser. Unlike a helper application, a plug-in can enable external content to blend into the document seamlessly.

The most common plug-ins are those that facilitate the playback of audio and video from the server. Audio may include music tracks that play in the background while visiting a page or live (streaming) audio, similar to a radio station. Video and animation can operate in a space on the page when played through a plug-in that knows how to process such data.

Today's browsers tend to ship with plug-ins that decode the most common sound file types. Developers of plug-ins for Internet Explorer for the Windows operating system commonly implement plug-ins as ActiveX controls — a distinction that is important to the underpinnings of the operating system but not to the user.

Plug-ins and helpers are valuable for more than just audio and video playback. A popular helper application is the *Adobe Acrobat Reader*, which displays Acrobat files that are formatted just as if they were being printed. But for interactivity, developers today frequently rely on Macromedia Corporation's *Flash* plug-in. Created using the Macromedia Flash authoring environment, a Flash document can have active clickable areas and draggable elements. Some authors even simulate artistic video games and animated stories in Flash. A browser equipped with the Flash plug-in displays the content in a rectangular area embedded within the browser page.

One potential downside for authoring interactive content in Flash or similar environments is that if the user does not have the correct plug-in version installed, it can take some time to download the plug-in (if the user even wants to bother). Moreover, once the plug-in is installed, highly graphic and interactive content can take longer to download to the client (especially on a dial-up connection) than some users are willing to wait. This is one of those situations in which you must balance your creative palette with the user's desire for your interactive content.

Another client-side technology — the Java applet — was popular for a while in the late 1990s but has fallen out of favor for a variety of reasons (some technical, some corporate-political). But this has not diminished the use of Java as a language for server and even cellular telephone programming, extending well beyond the scope of the language's founding company, Sun Microsystems.

JavaScript: A Language for All

Sun's Java language is derived from C and C++, but it is a distinct language. Its main audience is the experienced programmer. That leaves out many Web page authors. I was dismayed at this situation when I first read about Java's preliminary specifications in 1995. I would have preferred a language that casual programmers and scripters who were comfortable with authoring tools, such as Apple's once-formidable HyperCard and Microsoft's Visual Basic, could adopt quickly. As these accessible development platforms have shown, nonprofessional authors can dream up many creative applications, often for very specific tasks that no professional programmer would have the inclination to work on. Personal needs often drive development in the classroom, office, den, or garage. But Java was not going to be that kind of inclusive language.

My spirits lifted several months later, in November 1995, when I heard of a scripting language project brewing at Netscape Communications, Inc. Initially born under the name LiveScript, this language was developed in parallel with a new version of Netscape's Web server software. The language was to serve two purposes with the same syntax. One purpose was as a scripting language that Web server administrators could use to manage the server and connect its pages to other services, such as back-end databases and search engines for users looking up information. Extending the "Live" brand name further, Netscape assigned the name LiveWire to the database connectivity usage of LiveScript on the server.

On the client side — in HTML documents — authors could employ scripts written in this new language to enhance Web pages in a number of ways. For example, an author could use LiveScript to make sure that the user had filled in a required text field with an e-mail address or credit card number. Instead of forcing the server or database to do the data validation (requiring data exchanges between the client browser and the server), the user's computer handles all the calculation work — putting some of that otherwise wasted computing horsepower to work. In essence, LiveScript could provide HTML-level interaction for the user.

LiveScript becomes JavaScript

In early December 1995, just prior to the formal release of Navigator 2, Netscape and Sun Microsystems jointly announced that the scripting language thereafter would be known as JavaScript. Though Netscape had several good marketing reasons for adopting this name, the changeover may have contributed more confusion to both the Java and HTML scripting worlds than anyone expected.

Before the announcement, the language was already related to Java in some ways. Many of the basic syntax elements of the scripting language were reminiscent of the Java style. For client-side scripting, the language was intended for very different purposes than Java — essentially to function as a programming language integrated into HTML documents rather than as a language for writing applets that occupy a fixed rectangular area on the page (and that are oblivious to anything else on the page). Instead of Java's full-blown programming language vocabulary (and conceptually more difficult to learn object-oriented approach), JavaScript had a small vocabulary and a more easily digestible programming model.

The true difficulty, it turned out, was making the distinction between Java and JavaScript clear to the world. Many computer journalists made major blunders when they said or implied that JavaScript provided a simpler way of building Java applets. To this day, some new programmers believe JavaScript is synonymous with the Java language: They post Java queries to JavaScript-specific Internet newsgroups and mailing lists.

The fact remains that client-side Java and JavaScript are more different than they are similar. The two languages employ entirely different interpreter engines to execute their lines of code.

Enter Microsoft and others

Although the JavaScript language originated at Netscape, Microsoft acknowledged the potential power and popularity of the language by implementing it (under the JScript name) in Internet Explorer 3. Even if Microsoft might prefer that the world use the VBScript (Visual Basic Script) language that it provides in the Windows versions of IE, the fact that JavaScript is available on more browsers and operating systems makes it the client-side scripter's choice for anyone who must design for a broad range of users.

With scripting firmly entrenched in the mainstream browsers from Microsoft and Netscape, newer browser makers automatically provided support for JavaScript. Therefore, you can count on fundamental scripting services in browsers such as Opera or the Apple Safari

browser (the latter built upon an Open Source browser called KHTML). Not that all browsers work the same way in every detail — a significant challenge for client-side scripting that is addressed throughout this book.

JavaScript: The Right Tool for the Right Job

Knowing how to match an authoring tool to a solution-building task is an important part of being a well-rounded Web site author. A Web designer who ignores JavaScript is akin to a plumber who bruises his knuckles by using pliers instead of the wrench from the bottom of the toolbox.

By the same token, JavaScript won't fulfill every dream. The more you understand about JavaScript's intentions and limitations, the more likely you will be to turn to it immediately when it is the proper tool. In particular, look to JavaScript for the following kinds of solutions:

- ✦ Getting your Web page to respond or react directly to user interaction with form elements (input fields, text areas, buttons, radio buttons, checkboxes, selection lists) and hypertext links
- ✦ Distributing small collections of database-like information and providing a friendly interface to that data
- ✦ Controlling multiple-frame navigation, plug-ins, or Java applets based on user choices in the HTML document
- ✦ Preprocessing data on the client before submission to a server
- ✦ Changing content and styles in modern browsers dynamically and instantly in response to user interaction

At the same time, it is equally important to understand what JavaScript is *not* capable of doing. Scripters waste many hours looking for ways of carrying out tasks for which JavaScript was not designed. Most of the limitations are designed to protect visitors from invasions of privacy or unauthorized access to their desktop computers. Therefore, unless a visitor uses a modern browser and explicitly gives you permission to access protected parts of his or her computer, JavaScript cannot surreptitiously perform any of the following actions:

- ✦ Setting or retrieving the browser's preferences settings, main window appearance features, action buttons, and printing
- ✦ Launching an application on the client computer
- ✦ Reading or writing files or directories on the client or server computer
- ✦ Capturing live data streams from the server for retransmission
- ✦ Sending secret e-mails from Web site visitors to you

Web site authors are constantly seeking tools that will make their sites engaging (if not "cool") with the least amount of effort. This is particularly true when the task is in the hands of people more comfortable with writing, graphic design, and page layout than with hard-core programming. Not every Webmaster has legions of experienced programmers on hand to whip up some special, custom enhancement for the site. Nor does every Web author have control over the Web server that physically houses the collection of HTML and graphics files. JavaScript brings programming power within reach of anyone familiar with HTML, even when the server is a black box at the other end of a telephone line.

✦ ✦ ✦

CHAPTER 2

Authoring Challenges Amid the Browser Wars

In This Chapter

How leapfrogging browser developments hurt Web developers

Separating the core JavaScript language from document objects

The importance of developing a cross-browser strategy

If you are starting to learn JavaScript at this point in the brief history of scriptable browsers, you have both a distinct advantage and disadvantage. The advantage is that you have the wonderful capabilities of mature browser offerings from Microsoft, Netscape, The Mozilla Foundation, Apple, and others at your bidding. The disadvantage is that you have not experienced the painful history of authoring for older browser versions that were buggy and at times incompatible with one another due to a lack of standards. You have yet to learn the anguish of carefully devising a scripted application for the browser version you use only to have site visitors sending you voluminous e-mail messages about how the page triggers all kinds of script errors when run on a different browser brand, generation, or operating system platform.

Welcome to the real world of scripting Web pages with JavaScript. Several dynamics are at work to help make an author's life difficult if the audience for the application uses more than a single type of browser. This chapter introduces you to these challenges before you type your first word of JavaScript code. My fear is that the subjects I raise may dissuade you from progressing further into JavaScript and its powers. But as a developer myself — and as someone who has been using JavaScript since the earliest days of its public prerelease availability — I dare not sugarcoat the issues facing scripters today. Instead, I want to make sure you have an appreciation of what lies ahead to assist you in learning the language. I believe if you understand the big picture of the browser-scripting world as it stands in the year 2004, you will find it easier to target JavaScript usage in your Web application development and be successful at it.

Leapfrog

Browser compatibility has been an issue for authors since the earliest days of the Web gold rush — long before JavaScript. Despite the fact that browser developers and other interested parties voiced their opinions during formative stages of standards development, HTML authors could not produce a document that appeared the same pixel by pixel on all client machines. It may have been one thing to establish a set of standard tags for defining heading levels and line

breaks, but it was rare for the actual rendering of content inside those tags to look identical on different brands of browsers on different operating systems.

Then, as the competitive world heated up — and Web browser development transformed itself from a volunteer undertaking into profit-seeking businesses — creative people defined new features and new tags that helped authors develop more flexible and interesting looking pages. As happens a lot in any computer-related industry, the pace of commercial development easily surpassed the studied progress of standards. A browser maker would build a new HTML feature into a browser and only then propose that feature to the relevant standards body. Web authors were using these features (sometimes for prerelease browser versions) before the proposals were published for review.

When the deployment of content depends almost entirely on an interpretive engine on the client computer receiving the data — the HTML engine in a browser, for example — authors face an immediate problem. Unlike a standalone computer program that can extend and even invent functionality and have it run on everyone's computer (at least for a given operating system), Web content providers must rely on the functionality built into the browser. This led to questions such as, "If not all browsers coming to my site support a particular HTML feature, then should I apply newfangled HTML features for visitors only at the bleeding edge?" and "If I do deploy the new features, what do I do for those with older browsers?"

Authors who developed pages in the earliest days of the Web wrestled with these questions for many HTML features that we today take for granted. Tables and frames come to mind. Eventually, the standards caught up with the proposed HTML extensions — but not without a lot of author woe along the way.

Despite the current dominance of Microsoft's Internet Explorer browser on the dominant Windows operating system, the number of browsers that people use is not shrinking. Several recent browsers, including the modern Netscape 7 and Firebird browsers, are based on an Open Source browser called Mozilla. The Macintosh operating system now includes its own Apple-branded browser, Safari (released in 2003). And the independent Opera browser also has a home on some users' computers. All of these non-Microsoft browser makers obviously believe that they bring improvements to the world to justify their development — building better mousetraps you might say.

Duck and Cover

Today's browser wars are fought on different battlegrounds than in the early days of the Web. The breadth and depth of established Web standards have substantially fattened the browser applications — and books that developers read to exploit those standards for their content. On the one hand, most developers clamor for deeper standards support in new browser versions. On the other hand, everyday users care little about standards. All they want is to have an enjoyable time finding the information they seek on the Web. Most users are slow to upgrade their browsers, holding out until their favorite sites start breaking in their ancient browsers.

Industry standards don't necessarily make the Web developer's job any easier. For one thing, the standards are unevenly implemented across the latest browsers. Some browsers go further in their support than others. Then there are occasional differences in interpretation of vague standards details. And sometimes the standards don't provide any guidance in areas that are vital to content developers. At times we are left to the whims of browser makers who fill the gaps with proprietary features, in the hope that those features will become de facto standards.

As happens in war, civilian casualties mount when the big guns start shooting. The browser battle lines shifted dramatically in only a few years. The huge market share territory once

under Netscape's command now lies in Microsoft hands. The Netscape brand, itself, is fading as a result of corporate dealings; but the spirit continues in the populist, Open Source Mozilla effort. While a fair amount of authoring common ground exists between the latest versions of today's browsers, uneven implementation of the newest features causes the biggest problems for authors wishing to deploy on all browsers. Trying to define the common denominator may be the toughest part of the authoring job.

Compatibility Issues Today

Allow me to describe the current status of the compatibility situation among the top three browser families: Microsoft's Internet Explorer, browsers based on Mozilla, and Apple's Safari. The discussion in the next few sections intentionally does not get into specific scripting technology very deeply — some of you may know very little about programming at this point. In many chapters throughout Parts III and IV, I offer scripting suggestions to accommodate a variety of browsers.

Separating language from objects

Although early JavaScript authors initially treated client-side scripting as one environment that permitted the programming of page elements, the scene has changed as the browsers have matured. Today, a clear distinction exists between specifications for the core JavaScript language and for the elements you script in a document (for example, buttons and fields in a form).

On one level, this separation is a good thing. It means that one specification exists for basic programming concepts and syntax, which could become the programming language in any number of other environments. You can think of the core language as basic wiring. Once you know how electric wires work, you can connect them to all kinds of electrical devices. Similarly, JavaScript today is used to wire together elements in an HTML document. Tomorrow, operating systems could use the core language to enable users to wire together desktop applications that need to exchange information automatically.

At the ends of today's JavaScript wires are the elements on the page. In programming jargon, these items are known as *document objects*. By keeping the specifications for document objects separate from the wires that connect them, you can use other kinds of wires (other languages) to connect them. It's like designing telephones that can work with any kind of wire, including a type of wire that hasn't been invented yet. Today the devices can work with copper wire or fiber optic cable. You get a good picture of this separation in Internet Explorer, whose set of document objects can be scripted with JavaScript or VBScript. They're the same objects, just different wiring.

The separation of core language from document objects enables each concept to have its own standards effort and development pace. But even with recommended standards for each factor, each browser maker is free to extend the standards. Furthermore, authors may have to expend more effort to devise one version of a page or script that plays on multiple browsers unless the script adheres to a common denominator (or uses some other branching techniques to let each browser run its own way).

Core language standard

Keeping track of JavaScript language versions requires a brief history lesson. The first version of JavaScript (in Netscape Navigator 2) was Version 1.0, although that numbering was not part of the language usage. JavaScript was JavaScript. Version numbering became an

issue when Navigator 3 was released. The version of JavaScript associated with that Navigator version was JavaScript 1.1. The first appearance of the Navigator 4 generation increased the language version one more notch with JavaScript 1.2.

Microsoft's scripting effort contributes confusion for scripting newcomers. The first version of Internet Explorer to include scripting was Internet Explorer 3. The timing of Internet Explorer 3 was roughly coincidental to Navigator 3. But as scripters soon discovered, Microsoft's scripting effort was one generation behind. Microsoft did not license the JavaScript name. As a result, the company called its language JScript. Even so, the HTML tag attribute that lets you name the language of the script inside the tags could be either JScript or JavaScript for Internet Explorer. Internet Explorer 3 could understand a JavaScript script written for Navigator 2.

During this period of dominance by Navigator 3 and Internet Explorer 3, scripting newcomers were often confused because they expected the scripting languages to be the same. Unfortunately for the scripters, there were language features in JavaScript 1.1 that were not available in the older JavaScript version in Internet Explorer 3. Microsoft improved JavaScript in IE3 with an upgrade to the .dll file that gives IE its JavaScript syntax. However, it's hard to know which .dll is installed in any given visitor's IE3. The situation smoothed out for Internet Explorer 4. Its core language was essentially up to the level of JavaScript 1.2 as in early releases of Navigator 4. Microsoft still officially called the language JScript. Almost all language features that were new in Navigator 4 were understood when you loaded the scripts into Internet Explorer 4.

While all of this jockeying for JavaScript versions was happening, Netscape, Microsoft, and other concerned parties met to establish a core language standard. The standards body is a Switzerland-based organization originally called the European Computer Manufacturer's Association and now known simply as ECMA (commonly pronounced ECK-ma). In mid-1997, the first formal language specification was agreed on and published (ECMA-262). Due to licensing issues with the JavaScript name, the body created a new name for the language: ECMAScript.

With only minor and esoteric differences, this first version of ECMAScript was essentially the same as JavaScript 1.1 found in Navigator 3. Both Navigator 4 and Internet Explorer 4 officially supported the ECMAScript standard. Moreover, as happens so often when commerce meets standards bodies, both browsers went beyond the ECMAScript standard. Fortunately, the common denominator of this extended core language is broad, lessening authoring headaches on this front.

JavaScript version 1.3 was implemented in Netscape Navigator 4.06 through 4.7x. This language version is also the one supported in IE 5, 5.5, and 6. A few new language features are incorporated in JavaScript 1.5, as implemented in Mozilla-based browsers (including Navigator 6 and later).

In practice, so many browsers in use today support all but a few leading-edge features of the Mozilla browsers that JavaScript version numbers are mostly irrelevant. Other compatibility issues with older browsers will likely get in your way before core language problems. The time has come to forget about elaborate workarounds for inadequacies of the oldest browsers.

Document object model

If prevalent browsers have been close to each other in core JavaScript language compatibility, nothing could be further from the truth when it comes to the document objects. Internet Explorer 3 based its document object model (DOM) on that of Netscape Navigator 2, the same

browser level it used as a model for the core language. When Netscape added a couple of new objects to the model in Navigator 3, the addition caused further headaches for neophyte scripters who expected those objects to appear in Internet Explorer 3. Probably the most commonly missed object in Internet Explorer 3 was the image object, which lets scripts swap the image when a user rolls the cursor atop a graphic — *mouse rollovers*, they're commonly called.

In the Level 4 browsers, however, Internet Explorer's document object model jumped way ahead of the object model that Netscape implemented in Navigator 4. The two most revolutionary aspects of IE4 were the ability to script virtually every element in an HTML document and the instant reflow of a page when the content changed. This opened the way for HTML content to be genuinely dynamic without requiring the browser to fetch a rearranged page from the server. NN4 implemented only a small portion of this dynamism, without exposing all elements to scripts or reflowing the page. It introduced a proprietary layering concept that was abandoned in future Navigator versions. Inline content could not change in NN4 as it could in IE4. Suffice it to say IE4 was an enviable implementation.

At the same time, a DOM standard was being negotiated under the auspices of the World Wide Web Consortium (W3C). The hope among scripters was that once a standard was in place, it would be easier to develop dynamic content for all browsers that supported the standard. The resulting standard — the W3C DOM — formalized the notion of being able to script every element on the page, as in IE4. But it also invented an entirely new object syntax that no browser had used previously. The race was on for browsers to support the W3C DOM standards.

An arm of the Netscape company, called Mozilla.org, was formed to create an all-new browser dedicated to supporting industry standards. The engine for the Mozilla browser became the foundation for the all-new Navigator 6. It incorporated all of the W3C DOM Level 1 and a good chunk of Level 2. Mozilla 1.01 became the basis for the Netscape 7 browser, while Netscape 7.1 was built on the Mozilla 1.4 generation. In the summer of 2003, Netscape's parent company, AOL Time Warner, decided to end further Netscape-branded browser development. The work on the underlying Mozilla browser, however, continues under an independent organization called The Mozilla Foundation. Mozilla-branded browsers, and others using the same engine, continue to be upgraded and released to the public. The Mozilla engine offers arguably the most in-depth support for the W3C DOM Level 2 standard.

Even though Microsoft participated in the W3C DOM standards development, IE5 and 5.5 implemented only some of the W3C DOM standard — in some cases, just enough to allow simple cross-browser scripting that adheres to the standard. Microsoft further filled out W3C DOM support in IE6, but chose to omit several important parts. DOM support in Apple's Safari 1.0 lies in between that of Mozilla and IE6. Of course, the standard is not perfect either, lacking some practical features that IE offered back in IE4.

Despite this seemingly tortuous history of DOM development and browser support, you may wonder how anyone can approach DOM scripting with hope of success. Yet you'd be amazed at how much you can accomplish with today's browsers. You'll certainly encounter compatibility issues along the way, but this book will guide you through the most common problems and equip you to tackle others.

Cascading Style Sheets

Navigator 4 and Internet Explorer 4 were the first browsers to claim compatibility with a W3C recommendation called *Cascading Style Sheets Level 1 (CSS1)*. This specification provided designers an organized way to customize the look and feel of a document (and thus minimized the HTML in each tag). As implementations go, NN4 had a lot of rough edges, especially when

trying to mix stylesheets and tables. But IE4 was no angel, either, especially when comparing the results of stylesheet assignments as rendered in the Windows and Macintosh versions of the browser (developed by two separate teams).

CSS Level 2 adds more style functionality to the standard, and IE6, Mozilla-based browsers, and Safari support a good deal of Level 2 (albeit unevenly). Rendering of styled content is more harmonious among browsers, largely thanks to guidelines about how styles should render. Complex layouts, however, still need careful tweaking from time to time because of different interpretations of the standard.

JavaScript plays a role in stylesheets in IE4+, Mozilla, and Safari because those browsers' object models permit dynamic modification to styles associated with any content on the page. Stylesheet information is part of the object model and is therefore accessible and modifiable from JavaScript.

Dynamic HTML

Perhaps the biggest improvements to the inner workings of the Level 4 browsers from both Netscape and Microsoft revolved around a concept called *Dynamic HTML (DHTML)*. The ultimate goal of DHTML was to enable scripts in documents to control the content, content position, and content appearance in response to user actions. To that end, the W3C organization developed another standard for the precise positioning of HTML elements on a page as an extension of the CSS standards effort. The CSS-Positioning recommendation was later blended into the CSS standard, and both are now part of CSS Level 2. With positioning, you can define an exact location on the page where an element should appear, whether the item should be visible, and what stacking order it should take among all the items that might overlap it.

IE4+ adheres to the positioning standard syntax and makes positionable items subject to script control. Navigator 4 followed the standard from a conceptual point of view, but it implemented an alternative methodology involving an entirely new, and eventually unsanctioned, tag for layers. Such positionable items were scriptable in Navigator 4 as well, although a lot of the script syntax differed from that used in Internet Explorer 4. Fortunately for DHTML authors, Mozilla, by its adherence to the CSS standard, is more syntactically in line with DHTML style properties employed in IE4+. Numerous issues still surround the IE implementation because of Microsoft's shift with IE6 to a more accurate interpretation of certain measurement systems in CSS. The new system can be switched on in IE6 to be more in sync with the CSS standard as implemented in Mozilla and Safari browsers. Cross-browser DHTML scripting can be challenging, yet it is certainly possible if you understand the limitations imposed by following a common denominator.

Developing a Scripting Strategy

Browsers representing the latest generation contain a hodgepodge of standards and proprietary extensions. Even if you try to script to a common denominator among today's browsers, your code probably won't take into account the earlier versions of both the JavaScript core language and the browser document object models.

The true challenge for authors these days is determining the audience for which scripted pages are intended. You learn techniques in Chapter 13 that enable you to redirect users to different paths in your Web site based on their browser capabilities. In Chapter 14, you discover the alternatives you can take depending on the object model version(s) and specific features you need to support. Each new browser generation not only brings with it new and exciting features you are probably eager to employ in your pages, it also adds

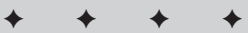
to the fragmentation of the audience visiting a publicly accessible page. With each new browser upgrade, fewer existing users are willing to download megabytes of browser merely to have the latest and greatest browser version. For many pioneers — and certainly for most nontechie users — there is a shrinking imperative to upgrade browsers, unless that browser comes via a new computer or operating system upgrade.

At this stage in the history of scriptable browsers, I take the stand that we should assume that a Web surfer arrives with a browser equipped with support for at least simple W3C DOM and DHTML capabilities. That certainly won't be the case 100 percent of the time, so it is also your obligation to apply scripting in an additive, or value-added manner. By this I mean that your pages should convey their primary information to the most brain-dead browser; but visitors with recent scriptable browsers will have a more enjoyable experience — better interactivity, faster performance, and a more engaging presentation. You will not only be contributing to the state of the art, but also carrying on the original vision of scripting in the browser.

✦ ✦ ✦

CHAPTER 3

Your First JavaScript Script

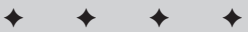


In This Chapter

How to choose basic JavaScript authoring tools

How to set up your authoring environment

How to enter a simple script to a Web page



In this chapter, you set up a productive scriptwriting and previewing environment on your computer, and then you write a simple script whose results you can see in your JavaScript-compatible browser.

Because of differences in the way various personal computing operating systems behave, I present details of environments for two popular variants: Windows (95 through XP) and MacOS X. For the most part, your JavaScript authoring experience is the same regardless of the operating system platform you use — including Linux or UNIX. Although there may be slight differences in font designs depending on your browser and operating system, the information remains the same. Most illustrations of browser output in this book are made from the Windows XP version of Internet Explorer 6. If you run another browser or version, don't fret if every pixel doesn't match with the illustrations in this book.

The Software Tools

The best way to learn JavaScript is to type the HTML and scripting code into documents in a text editor. Your choice of editor is up to you, although I provide you with some guidelines for choosing a text editor in the next section.

Choosing a text editor

For the purposes of learning JavaScript in this book, avoid WYSIWYG (What You See Is What You Get) Web-page authoring tools, such as FrontPage and DreamWeaver, for now. These tools certainly will come in handy afterward when you can productively use those facilities for molding the bulk of your content and layout. But the examples in this book focus more on script content (which you must type in anyway), so there isn't much HTML that you have to type. Files for all complete Web page listings in this book (except for the tutorial chapters) also appear on the companion CD-ROM.

An important factor to consider in your choice of editor is how easy it is to save standard text files with an .html filename extension. In the case of Windows, any program that not only saves the file as text by default but also enables you to set the extension to .htm or .html prevents a great deal of problems. If you use Microsoft Word, for

example, the program tries to save files as binary Word files — something that no Web browser can load. To save the file initially as a text or .html extension file requires mucking around in the Save As dialog box. This requirement is truly a nuisance.

Nothing's wrong with using bare-essentials text editors. In Windows, that includes the WordPad program or a more fully featured product such as the shareware editor called TextPad. For MacOS X, TextEdit is also fine. A favorite among Mac HTML authors and scripters is BBEdit (Bare Bones Software), which includes a number of useful aids for scripters, such as optional line numbers (which help in debugging JavaScript).

Choosing a browser

The other component that is required for learning JavaScript is the browser. You don't have to be connected to the Internet to test your scripts in the browser. You can perform all testing offline. This means you can learn JavaScript and create cool, scripted Web pages with a laptop computer — even on a boat in the middle of an ocean.

The browser brand and version you use is up to you. Because the tutorial chapters in this book teach the W3C DOM syntax, you should be using a recent browser. Any of the following will get you through the tutorial: Internet Explorer 5 or later (Windows or Macintosh); any Mozilla-based browser (including Netscape 7 and later); and Apple Safari.

Many example listings in Parts III and IV of this book demonstrate language or document object model (DOM) features that work on only specific browsers and versions. Check the compatibility listing for that language or DOM feature to make sure you use the right browser to load the page.

Setting Up Your Authoring Environment

To make the job of testing your scripts easier, make sure that you have enough free memory in your computer to let both your text editor and browser run simultaneously. You need to be able to switch quickly between editor and browser as you experiment and repair any errors that may creep into your code. The typical workflow entails the following steps:

1. Enter HTML and script code into the source document in the text editor.
2. Save the latest version to disk.
3. Switch to the browser.
4. Do one of the following: If this is a new document, open the file via the browser's Open menu. If the document is already loaded, reload the file into the browser.

Steps 2 through 4 are the key ones you will follow frequently. I call this three-step sequence the save-switch-reload sequence. You will perform this sequence so often as you script that the physical act quickly will become second nature to you. How you arrange your application windows and effect the save-switch-reload sequence varies according to your operating system.

Windows

You don't have to have either the editor or browser window maximized (at full screen) to take advantage of them. In fact, you may find them easier to work with if you adjust the size

and location of each window so both windows are as large as possible while still enabling you to click a sliver of the other's window. Or, you can leave the taskbar visible so you can click the desired program's button to switch to its window (see Figure 3-1). A monitor that displays more than 800 × 600 pixels certainly helps in offering more screen real estate for the windows and the taskbar.

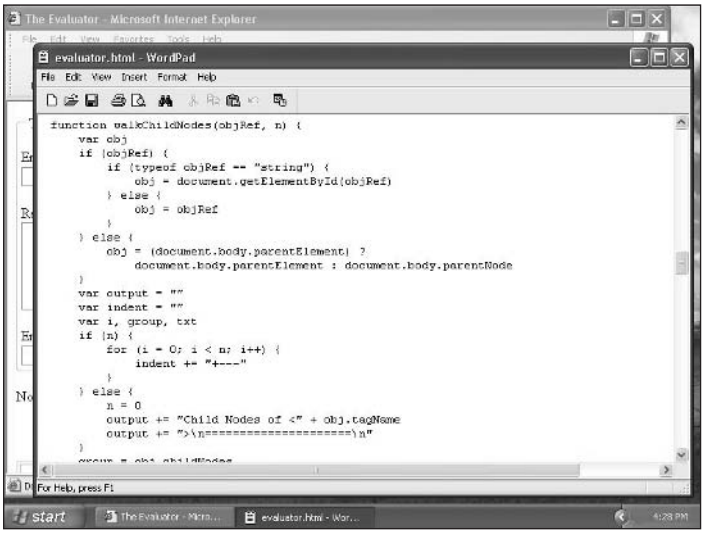


Figure 3-1: Editor and browser window arrangement in Windows XP.

In practice, however, the Windows Alt+Tab task-switching keyboard shortcut makes the job of the save-switch-reload steps outlined earlier a snap. If you run Windows and also use a Windows-compatible text editor (which more than likely has a Ctrl+S file-saving keyboard shortcut), you can effect the save-switch-reload sequence from the keyboard all with the left hand: Ctrl+S (save the source file); Alt+Tab (switch to the browser); Ctrl+R (reload the saved source file).

As long as you keep switching between the browser and text editor via Alt+Tab task switching, either program is always just an Alt+Tab away.

MacOS X

In MacOS X you can change between your text editor and browser applications via the Dock or, more conveniently, by typing ⌘-Tab. As long as you stay strictly in those two applications, the other program is only one ⌘-Tab away (see Figure 3-2).

With this setup, the save-switch-reload sequence is a simple affair:

1. Press ⌘-S (save the source file).
2. Press ⌘-Tab (switch to the browser).
3. Press ⌘-R (reload the saved source file).

To return to editing the source file, press ⌘-Tab again.

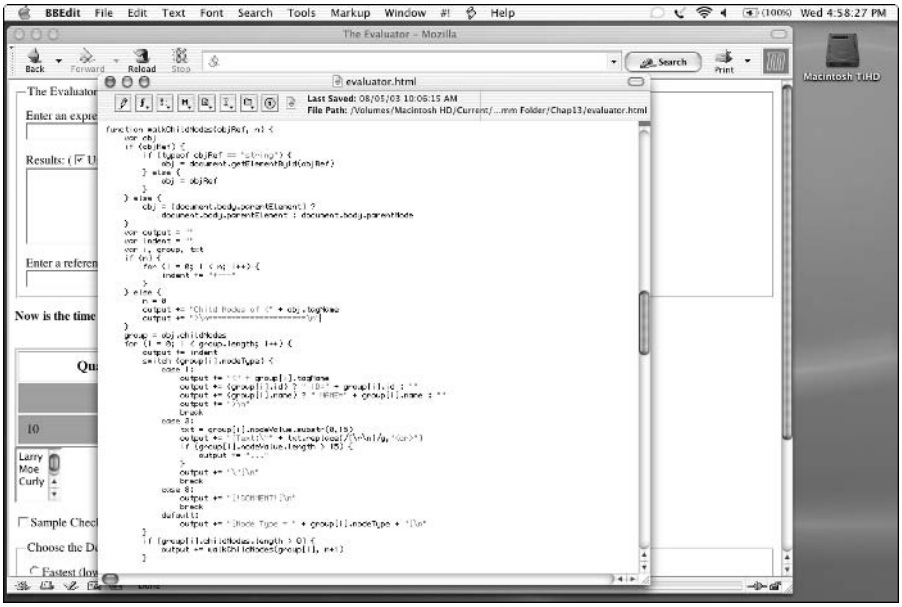


Figure 3-2: Editor and browser window arrangement on the Macintosh screen.

Reloading issues

For the most part, a simple page reload is enough to let you test a revised version of a script right away. But sometimes the browser's cache (with its default settings) can preserve parts of the previous page's attributes when you reload, even though you have changed the source code. To perform a more thorough reload, hold down the Shift key while clicking the browser's Reload/Refresh button. Alternatively, you can turn off the browser's cache in the preferences area, but that setting may negatively affect the overall performance of the browser during your regular Web surfing.

What Your First Script Will Do

For the sake of simplicity, the kind of script you look at in the next section is the kind that runs automatically when the browser loads the HTML page. Although all scripting and browsing work done here is offline, the behavior of the page is identical if you place the source file on a server and someone accesses it via the Web.

Figure 3-3 shows the page as it appears in the browser after you're finished. (The exact wording differs slightly if you run your browser on an operating system platform other than Windows XP or if you use a browser other than Internet Explorer.) The part of the page that is defined in regular HTML contains nothing more than an `<h1>`-level header with a horizontal rule under it. If someone does not use a JavaScript-equipped browser, all he or she sees is the header and horizontal rule (unless that person has a truly outmoded browser, in which case some of the script words appear in the page).

Below the rule, the script displays plain body text that combines static text with information about the browser you use to load the document. The script writes a stream of HTML

information to the browser, including a tag that applies a stylesheet to render a portion of the information in boldface. Even though two lines of code are writing information to the page, the result is rendered as one line — just as it is when all the text is hard-coded in HTML.

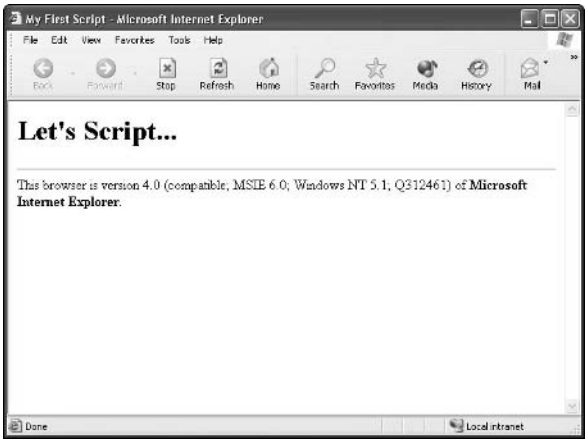


Figure 3-3: The finished page of your first JavaScript script.

Entering Your First Script

It's time to start creating your first JavaScript script. Launch your text editor and browser. If your browser offers to dial your Internet service provider (ISP) or begins dialing automatically, cancel or quit the dialing operation. If the browser's Stop button is active, click it to halt any network searching it may try to do. You may receive a dialog box message or page indicating that the URL for your browser's home page (usually the home page of the browser's publisher — unless you've changed the settings) is unavailable. That's fine. You want the browser open, but you needn't be connected to your ISP. If you're automatically connected to the Internet via a local area network in your office or school, or via cable modem or DSL, that's also fine. However, you don't need the network connection for now. Next, follow these steps to enter and preview your first JavaScript script:

1. Activate your text editor and create a new, blank document.
2. Type the script into the window exactly as shown in Listing 3-1.

Listing 3-1: **Source Code for script1.html**

```
<html>
<head>
<title>My First Script</title>
<style type="text/css">
.highlight {font-weight: bold}
</style>
</head>
```

Continued

Listing 3-1 *(continued)*

```
<body>
<h1>Let's Script...</h1>
<hr>
<script type="text/javascript">
<!-- hide from old browsers
document.write("This browser is version " + navigator.appVersion);
document.write(" of <span class='highlight'>" + navigator.appName + "</span>.");
// end script hiding -->
</script>
</body>
</html>
```

3. Save the document with the name `script1.html`.
4. Switch to your browser.
5. Choose Open (or Open File on some browsers) from the File menu and select `script1.html`. (On some browsers, you have to click a Browse button to reach the File dialog box.)

If you typed all lines as directed, the document in the browser window should look like the one in Figure 3-3 (with minor differences for your computer's operating system and browser version). If the browser indicates that a mistake exists somewhere as the document loads, don't do anything about it for now. (Click the OK button if you see a script error dialog box.) Let's first examine the details of the entire document so you understand some of the finer points of what the script is doing.

Examining the Script

You do not need to memorize any of the commands or syntax discussed in this section. Instead, relax and watch how the lines of the script become what you see in the browser. In Listing 3-1, all of the lines up to the `<script>` tag are very standard HTML with one Cascading Style Sheet (CSS) rule in the head portion.

The <script> tag

Any time you include JavaScript verbiage in an HTML document, you must enclose those lines inside a `<script>...</script>` tag pair. These tags alert the browser program to begin interpreting all the text between these tags as a script. Because other scripting languages (such as Microsoft's VBScript) can take advantage of these script tags, you must specify the kind of language in which the enclosed code is written. Therefore, when the browser receives the signal that your script is of the type `text/javascript`, it employs its built-in JavaScript interpreter to handle the code. You can find parallels to this setup in real life: If you have a French interpreter at your side, you need to know that the person with whom you're conversing also knows French. If you encounter someone from Russia, the French interpreter can't help you. Similarly, if your browser has only a JavaScript interpreter inside, it can't understand code written in VBScript.

Now is a good time to instill an aspect of JavaScript that will be important to you throughout all your scripting ventures: JavaScript is case-sensitive. Therefore, you must enter any item in your script that uses a JavaScript word with the correct uppercase and lowercase letters. Your HTML tags (including the `<script>` tag) can be in the case of your choice, but everything in JavaScript is case-sensitive.[1] When a line of JavaScript doesn't work, look for the wrong case first. Always compare your typed code against the listings printed in this book and against the various vocabulary entries discussed throughout it.

A script for all browsers

The next line after the `<script>` tag in Listing 3-1 appears to be the beginning of an HTML comment tag. It is, but the JavaScript interpreter treats comment tags in a special way. Although JavaScript dutifully ignores a line that begins with an HTML comment start tag, it treats the next line as a full-fledged script line. In other words, the scripting machinery inside the browser begins interpreting the next line after a comment start tag. If you want to put a comment inside JavaScript code, the comment must start with a double slash (`//`). Such a comment may go near the end of a line (such as after a JavaScript statement that is to be interpreted by the browser) or on its own line. In fact, the latter case appears near the end of the script. The comment line starts with two slashes.

Step back for a moment and notice that the entire script (including comments) is contained inside a standard HTML comment tag (`<!--comment-->`). The value of this containment is not clear until you see what happens to your scripted HTML document in a non-JavaScript-compatible browser. Such a browser blows past the `<script>` tag as being an advanced tag it doesn't understand. But it treats a line of script as regular text to be displayed in the page. If you enclose script lines between HTML comment tags, most older browsers don't display the script lines.

Remember, too, that some users don't have access to modern browsers or graphical browsers. (They use the Lynx text-oriented UNIX Web reader software or lightweight browsers in handheld computers.) By embracing your script lines within these comments, your Web pages don't look completely broken in relatively modern, non-JavaScript browsers.

Notice that the comment lines that shield older browsers from your scripts go *inside* the `<script>...</script>` tags. Do not put these comment lines above the `<script>` tag or below the `</script>` tag and expect them to work.

One more issue about the script-hiding comment lines in this book. To save space on the page, most examples do not have comment lines inserted in them. But as you can see in the full-fledged application examples from Chapters 48 through 57 on the CD-ROM, the comment lines are where they should be. For any pages you produce for public consumption, always encase your script lines inside these comments.

Displaying some text

Both script lines in Listing 3-1 use one of the possible actions a script can ask a document to perform (`document.write()`, meaning display text in the current document). You learn more about the `document` object in Chapter 18.

---

[1] XHTML style, if you intend to follow its conventions, requires all lowercase tags and attribute names. This is the style observed throughout this book.

Whenever you ask an object (a document in this case) to perform a task for you, the name of the task is always followed by a set of parentheses. In some cases — the `write()` task, for example — JavaScript needs to know what information it should act on. That information (called a *parameter*) goes inside parentheses after the name of the task. Thus, if you want to write the name of the first U.S. president to a document, the command to do so is

```
document.write("George Washington");
```

The line of text that the script writes starts with some static text ("This browser is version") and adds some evaluated text (the version of the browser) to it. The writing continues with more static text that includes an HTML tag (`" of <span class='highlight'>"`), more evaluated text (the name of the browser application), and an HTML closing tag and the sentence's period (`"</span>."`). JavaScript uses the plus symbol (+) to join (*concatenate*) text components into a larger, single string of text characters to be written by the document. Neither JavaScript nor the + symbol knows anything about words and spaces, so the script is responsible for making sure that the proper spaces are passed along as part of the parameters. Notice, therefore, that an extra space exists after the word "version" in the first `document.write()` parameter, and extra spaces exist on both sides of "of" in the second `document.write()` parameter.

To fetch the information about the browser version and name for your parameters, you call upon JavaScript to extract the corresponding properties from the `navigator` object. You extract a property by appending the property name to the object name (`navigator` in this case) and separating the two names with a period. If you're searching for some English to mentally assign to this scheme as you read it, start from the right side and call the right item a property "of" the left side: the `appVersion` property of the `navigator` object. This dot syntax looks a great deal like the `document.write()` task, but a property name does not have parentheses after it. In any case, the reference to the property in the script tells the JavaScript interpreter to insert the value of that property in the spot where the call is made. For your first attempt at the script, JavaScript substitutes the internal information about the browser as part of the text string that gets written to the document.

Finally, notice the semicolon characters at the end of each JavaScript statement. Trailing semicolons are purely optional. There is no penalty for leaving them out. If you intend to investigate other programming languages, such as Java or C++, for example, you'll find those semicolons are required. Program listings in this book use semicolons.

Have Some Fun

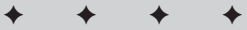
If you encounter an error in your first attempt at loading this document into your browser, go back to the text editor and check the lines of the script section against Listing 3-1, looking carefully at each line in light of the explanations. There may be a single character out of place, a lowercase letter where an uppercase one belongs, or a quote or parenthesis missing. Make necessary repairs, switch to your browser, and click Reload.

To see how dynamic the script in `script1.html` is, go back into the text editor and replace the word "browser" with "client software." Save, switch, and reload to see how the script changes the text in the document. Feel free to substitute other text for the quoted text in the `document.write()` statement. Or, add more text with additional `document.write()` statements. The parameters to `document.write()` are HTML text, so you can even write `"<br>"` to make a line break. Always be sure to save, switch, and reload to see the results of your handiwork.

✦ ✦ ✦

PART

JavaScript Tutorial



In This Part

Chapter 4
Browser and Document Objects

Chapter 5
Scripts and HTML Documents

Chapter 6
Programming Fundamentals, Part I

Chapter 7
Programming Fundamentals, Part II

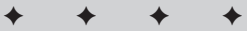
Chapter 8
Window and Document Objects

Chapter 9
Forms and Form Elements

Chapter 10
Strings, Math, and Dates

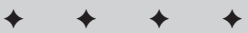
Chapter 11
Scripting Frames and Multiple Windows

Chapter 12
Images and Dynamic HTML



CHAPTER 4

Browser and Document Objects



In This Chapter

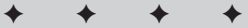
What client-side scripts do

What happens when a document loads

How the browser creates objects

How scripts refer to objects

What distinguishes one object from another



This chapter marks the first of nine tutorial chapters tailored to Web authors who have at least basic grounding in HTML concepts. In particular, you should already be familiar with common HTML tags and their attributes, as well as the fundamentals of Cascading Style Sheets (CSS). In this chapter, you see several practical applications of JavaScript and begin to see how a JavaScript-enabled browser turns familiar HTML elements into objects that your scripts control. This tutorial teaches concepts and terminology that apply to modern browsers with special focus on standards-compatibility to equip you to work with today's and tomorrow's browsers. You should study this tutorial in conjunction with any of the following browsers: Internet Explorer 5 or later (Windows or Macintosh), any Mozilla-based browser, or Apple's Safari.

Scripts Run the Show

If you have authored Web pages with HTML, you are familiar with how HTML tags influence the way content is rendered on a page when viewed in the browser. As the page loads, the browser recognizes angle-bracketed tags as formatting instructions. Instructions are read from the top of the document downward, and elements defined in the HTML document appear onscreen in the same order in which they appear in the document's source code. As an author, you do a little work one time and up front — adding the tags to text content — and the browser does a lot more work every time a visitor loads the page into a browser.

Assume for a moment that one of the elements on the page is a text input field inside a form. The user is supposed to enter some text in the text field and then click the Submit button to send that information back to the Web server. If that information must be an Internet e-mail address, how do you ensure the user includes the "@" symbol in the address?

One way is to have a Common Gateway Interface (CGI) program on the server inspect the submitted form data after the user clicks the Submit button and the form information is transferred to the server. If the user omits or forgets the "@" symbol, the CGI program sends the page back to the browser — but this time with an instruction to include the symbol in the address. Nothing is wrong with this

exchange, but it means a significant delay for the user to find out that the address does not contain the crucial symbol. Moreover, the Web server has to expend some of its resources to perform the validation and communicate back to the visitor. If the Web site is a busy one, the server may try to perform hundreds of these validations at any given moment, probably slowing the response time to the user even more.

Now imagine that the document containing that text input field has some intelligence built into it that makes sure the text field entry contains the "@" symbol before ever submitting one bit (literally!) of data to the server. That kind of intelligence would have to be embedded in the document in some fashion — downloaded with the page's content so it can stand ready to jump into action when called upon. The browser must know how to run that embedded program. Some user action must start the program, perhaps when the user clicks the Submit button. If the program runs inside the browser and detects a lack of the "@" symbol, an alert message should appear to bring the problem to the user's attention. The same program also should be capable of deciding if the actual submission can proceed or if it should wait until a valid e-mail address is entered into the field.

This kind of pre-submission data entry validation is but one of the practical ways JavaScript adds intelligence to an HTML document. Looking at this example, you might recognize that a script must know how to look into what is typed in a text field; a script must also know how to let a submission continue or how to abort the submission. A browser capable of running JavaScript programs conveniently treats elements such as the text field as *objects*. A JavaScript script controls the action and behavior of objects — most of which you see on the screen in the browser window.

JavaScript in Action

By adding lines of JavaScript code to your HTML documents, you control onscreen objects in whatever way your applications require. To give you an idea of the scope of applications you can create with JavaScript, I show you several applications on the CD-ROM (in the Listings folders for Chapters 48 through 57). I strongly suggest you open the applications and play with them in your browser as they are described in the next several pages. You can find links to the application files for these chapters via the `index.html` file located in the Listings folder on the CD-ROM.

Interactive user interfaces

HTML hyperlinks do a fine job, but they're not necessarily the most engaging way to present a table of contents for a large site or document. With a bit of JavaScript, you can create an interactive, expandable table of contents listing that displays the hierarchy of a large body of material (see Figure 4-1). Just like the text listings (or *tree views*) in operating system file management windows, the expandable table of contents lets the user see as much or as little as possible while displaying the big picture of the entire data collection.

Click a gray widget icon to expand the items underneath. An endpoint item has an orange and black widget icon. Items in the outline can be links to other pages or descriptive information. You also maintain the same kind of font control over each entry, as expected from HTML. While such outlines have been created with the aid of server programs in the past, the response time between clicks is terribly slow. By placing all of the smarts behind the outline inside the page, it downloads once and runs quickly after each click.

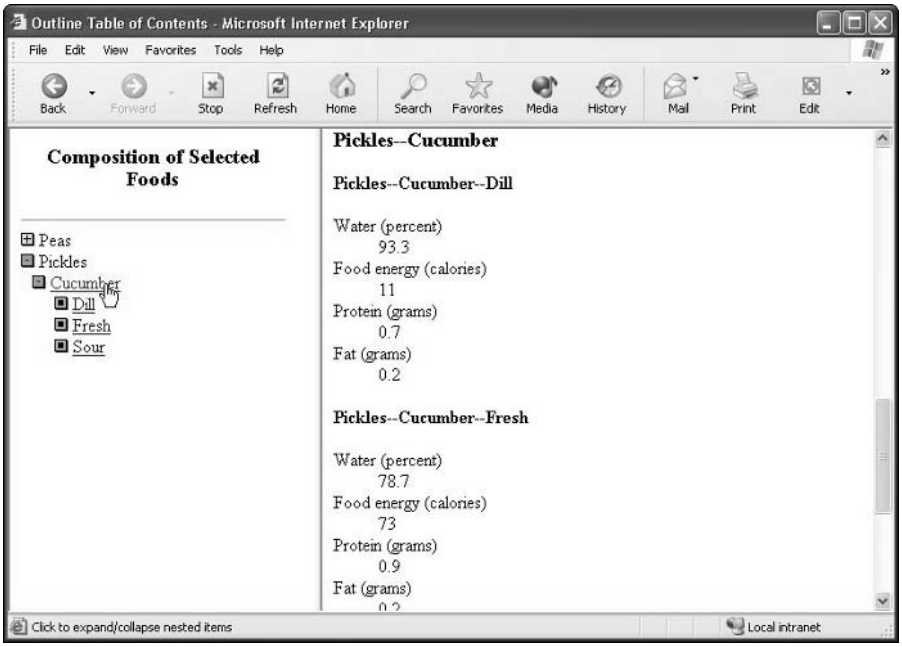


Figure 4-1: An expandable table of contents.

Small data lookup

A common application on the Web is having a server program present a page that visitors use to access large databases on the server. Large data collections are best left on the server where search engines and database technologies are the best fit. But if your page acts as a *front end* to a small data collection lookup, you can consider embedding that data collection in the document (out of view) and letting JavaScript act as the intermediary between user and data.

I do just that in a Social Security prefix lookup system shown in Figure 4-2. I convert a printed table of about 55 entries into a JavaScript list that occupies only a few hundred bytes. When the visitor types the three-character prefix of his or her Social Security number into the field and clicks the Search button, a script behind the scenes compares that number against the 55 or so ranges in the table. When the script finds a match, it displays the corresponding state of registration in a second field.

If the application were stored on the server and the data stored in a server database, each click of the Search button would mean a delay of many seconds as the server processed the request, got the data from the database, and reformulated the page with the result for the user. Built instead as a JavaScript application, once the page downloads the first time, scripts perform all lookups instantaneously.

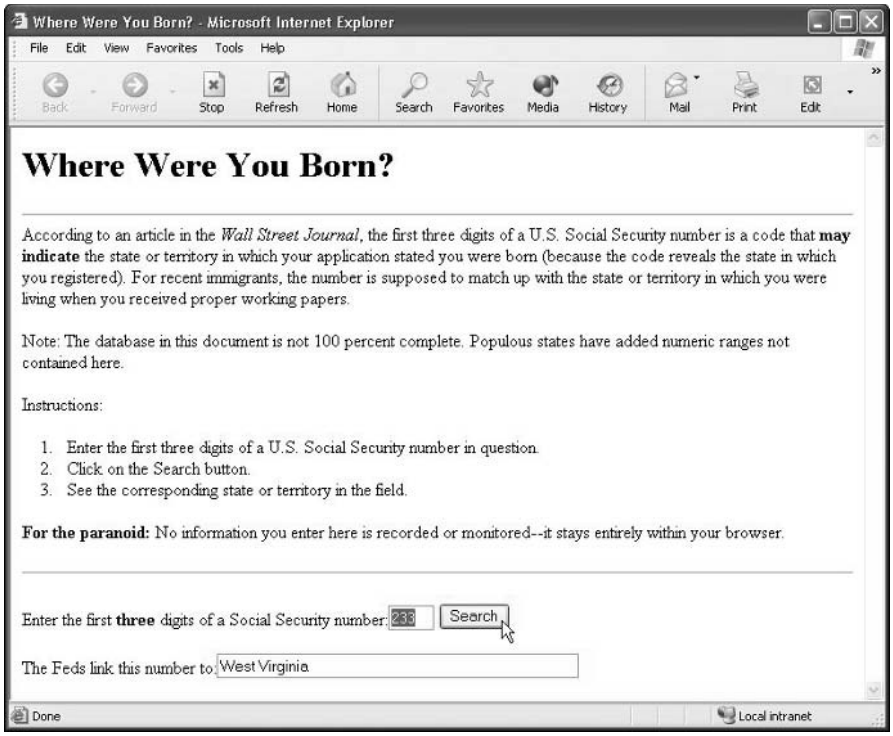


Figure 4-2: Looking up data in a small table.

Forms validation

I've already used data entry form validation as an example of when JavaScript is a good fit. In fact, the data entry field in the Social Security lookup page (see Figure 4-2) includes scripting to check the validity of the entered number. Just as a server program for this task has to verify that the entry is a three-digit number, so, too, must the JavaScript program verify the entered value. If a mistake appears in the entry — perhaps a finger slips and hits a letter key — the visitor is advised of the problem and directed to try another entry. The validation script even preselects the text in the entry field for the visitor so that typing a new value replaces the old one.

Interactive data

JavaScript opens opportunities for turning static information into interactive information. Figure 4-3 shows a graphical calculator for determining the value of an electrical component (called a *resistor*) whose only markings are colored bars.

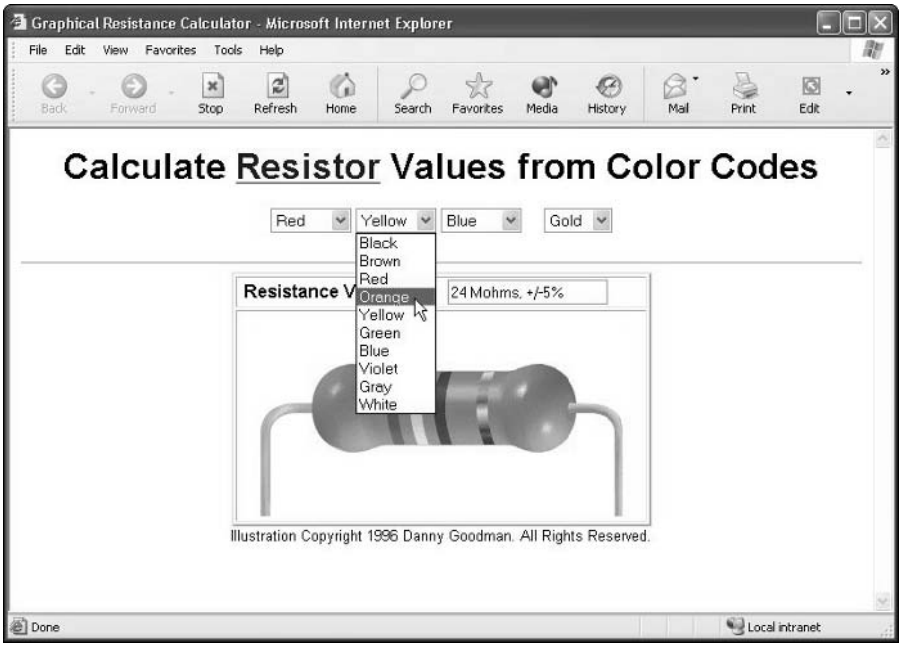


Figure 4-3: An interactive graphical calculator.

The image in the bottom half of the page is composed of seven images in vertical slices all bunched up against each other. Four slices display the colored bands, while the remaining three slices contain the ends of the resistor and the spacer between groups of bands. As the visitor selects a color from a pop-up list near the top, the associated image slice changes to the selected color and the resistance value is calculated and displayed.

Again, once the page is loaded, response time is instantaneous. Conversely, a server-based version of this calculator would take many seconds between color changes. Moreover, JavaScript provides the power to preload all possible images into the browser cache while the main page loads. Therefore, with only a slight extra delay to download all images with the page, no further delay occurs when a visitor chooses a new color. Not only is the application practical (for its intended audience), but it's just plain fun to play with.

Multiple frames

While frames are the domain of HTML, they suddenly become more powerful with some JavaScript behind them. The Decision Helper application shown in Figure 4-4 takes this notion to the extreme.

The Decision Helper is a full-fledged application that includes four input screens and one screen that displays the results of some fairly complex calculations based on the input screens. Results are shown both in numbers and in a bar graph form, as displayed in Figure 4-4.

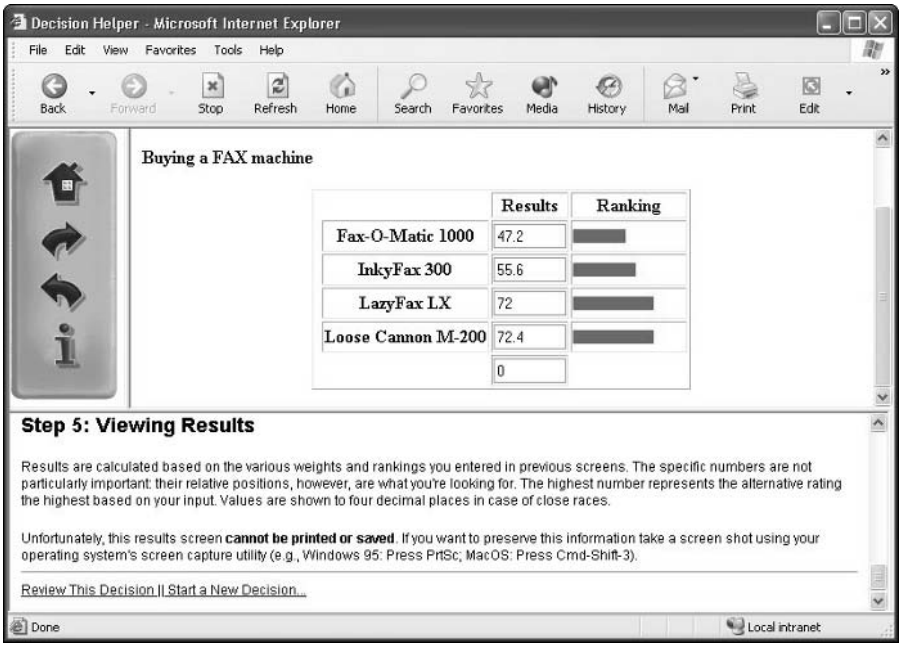


Figure 4-4: The Decision Helper.

Interaction among the three frames requires JavaScript. For example, suppose the user clicks one of the directional arrows in the top-left frame. Not only does the top-right frame change to another document, but the instructions document in the bottom frame also shifts to the anchor point that parallels the content of the input screen. Scripting behind the top-right frame documents uses various techniques to preserve entry information as the user navigates through the sequence of input pages. These are the same techniques you might use to build an online product catalog and shopping cart — accumulating the customer's selections from various catalog pages and then bringing them together in the checkout order form.

Certainly you could fashion this application out of a CGI program on the server. But the high level of interaction and calculation required would turn this now speedy application into a glacially slow exchange of information between user and server.

Dynamic HTML

Starting with the version 4 browsers from both Netscape and Microsoft, you can modify more and more content on the page with the help of client-side scripts. In Figure 4-5, for example, scripts in the page control the dragging of map pieces in the puzzle. Highlighted colors change as you click the state maps, instruction panels fly in from the edge of the screen, and another item appears when you place all the states in their proper positions.

Applying scripts to modify HTML content or position on the fly is commonly called *Dynamic HTML (DHTML)*. JavaScript becomes the vital connection between the user and dynamically repositionable elements on the screen. Not even a program on the server could help this application because you need immediate programmatic control in the page to respond to user mouse motion and instantaneous changes to screen elements.

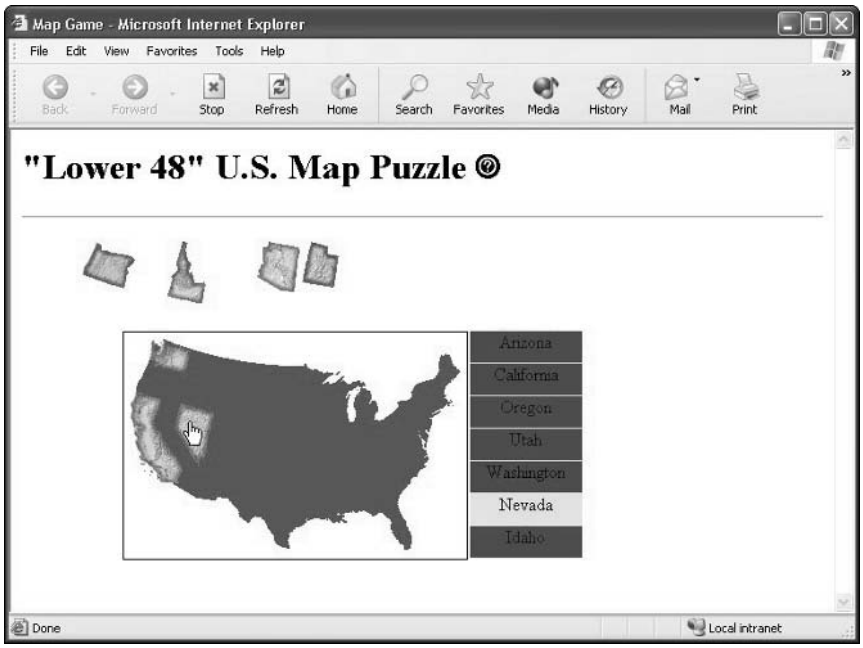


Figure 4-5: A map game in scriptable Dynamic HTML.

When to use JavaScript

The preceding examples demonstrate a wide range of applications for JavaScript, but by no means do they come close to exhausting JavaScript's possibilities. When faced with a Web application task, I look to client-side JavaScript for help with the following requirements:

- **Data entry validation:** If form fields need to be filled out for processing on the server, I let client-side scripts prequalify the data entered by the user.
- **Serverless CGIs:** I use this term to describe processes that, were it not for JavaScript, would be programmed as CGIs on the server, yielding slow performance because of the interactivity required between the program and user. This includes tasks such as small data collection lookup, modification of images, and generation of HTML in other frames and windows based on user input.
- **Dynamic HTML interactivity:** It's one thing to use DHTML's capabilities to position elements precisely on the page—you don't need scripting for that. But if you intend to make the content dance on the page, scripting makes that happen.
- **CGI prototyping:** Sometimes you may want a CGI program to be at the root of your application because it reduces the potential incompatibilities among browser brands and versions. It may be easier to create a prototype of the CGI in client-side JavaScript. Use this opportunity to polish the user interface before implementing the application as a CGI.
- **Offloading a busy server:** If you have a highly trafficked Web site, it may be beneficial to convert frequently used CGI processes to client-side JavaScript scripts. Once a page

is downloaded, the server is free to serve other visitors. Not only does this lighten server load, but users also experience quicker response to the application embedded in the page.

- **Adding life to otherwise dead pages:** HTML by itself is pretty "flat." Adding a blinking chunk of text doesn't help much; animated GIF images more often distract from, rather than contribute to, the user experience at your site. But if you can dream up ways to add some interactive zip to your page, it may engage the user and encourage a recommendation to friends or repeat visits.
- **Creating "Web pages that think":** If you let your imagination soar, you may develop new, intriguing ways to make your pages appear "smart." For example, in the application Intelligent "Updated" Flags (Chapter 54 on the CD-ROM), you see how (without a server CGI or database) an HTML page can "remember" when a visitor last came to the page. Then any items that have been updated since the last visit — regardless of the number of updates you've done to the page — are flagged for that visitor. That's the kind of subtle, thinking Web page that best displays JavaScript's powers.

The Document Object Model

Before you can truly start scripting, you should have a good feel for the kinds of objects you will be scripting. A scriptable browser does a lot of the work of creating software objects that generally represent the visible objects you see in an HTML page in the browser window. Obvious objects include form controls (text boxes and buttons) and images. However, there may be other objects that aren't so obvious by looking at a page but which make perfect sense when you consider the HTML tags used to generate a page's content — paragraph objects or frames of a frameset, for example.

To help scripts control these objects — and to help authors see some method to the madness of potentially dozens of objects on a page — the browser makers define a *document object model (DOM)*. A model is like a prototype or plan for the organization of objects on a page.

Evolution of browser DOMs has caused much confusion and consternation among scripters due to a lack of compatibility across succeeding generations and brands of browsers. Fortunately, the DOM world is stabilizing around a formal specification published by the World Wide Web Consortium (W3C). Today's modern browsers continue to support some of the "old ways" of the earliest DOM because so much existing script code on the Web relies on these traditions continuing to work (you'll see some of these in Chapter 9). But with the vast majority of browsers in use today supporting the basic W3C DOM syntax and terminology, scripters should aim toward standards compatibility whenever possible.

HTML structure and the DOM

An important trend in HTML markup is applying markup solely to define the structure of a document and the context of each piece of content in the document. The days of using HTML tags solely to influence the appearance of a chunk of text are drawing to a close. It is no longer acceptable to enclose a line of text in, say, an `<h1>` tag because you want the line to appear in the text size and weight that browsers automatically apply to text tagged in that way. An `<h1>` element has a special context within a document's structure: a first-level heading. In today's HTML world, if you wish to display a standalone line of text with a particular style, the text would likely be in a simple paragraph (`<p>`) tag; the precise look of that paragraph would be

under the control of a Cascading Style Sheet (CSS) rule. Current practice even frowns upon the application of `<b>` and `<i>` tags to assign boldface and italic styles to a span of text. Instead surround the text in a contextual tag (such as the `<em>` element to signify emphasis) and define the CSS style you wish applied to any emphasized text in the document.

The result of applying strict structural design to your HTML tagging is a document that has a well-defined *hierarchy* of elements based on their nesting within each other. For example, an empty HTML document has the following minimum elements:

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

The `html` element contains two nested elements, `head` and `body`. The hierarchy of elements can be charted like a corporate organizational chart, as shown in Figure 4-6. For the sake of upcoming terminology lessons, however, it is more convenient to visualize the chart in Figure 4-6 as a family tree — except that, unlike most real family trees each point that spawns children is a single-parent. In the empty HTML document, the `html` element is the parent of two child elements: `head` and `body`. The `html` element is, in turn, a child of the document.

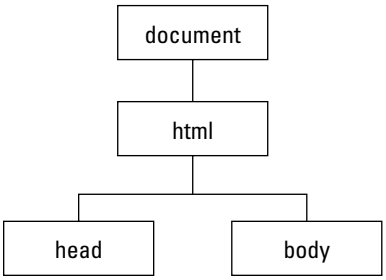


Figure 4-6: Element hierarchy of an empty HTML document.

The DOM in a browser window

As its name implies, the formal Document Object Model focuses primarily on the HTML document and the content nested inside it. From a practical standpoint, however, scripters often need to control the environment that contains the document: the window. The `window` object is the top of the hierarchy that browser scripts work with. The basic structure of the object model in modern browsers (given an empty HTML document) is shown in Figure 4-7.

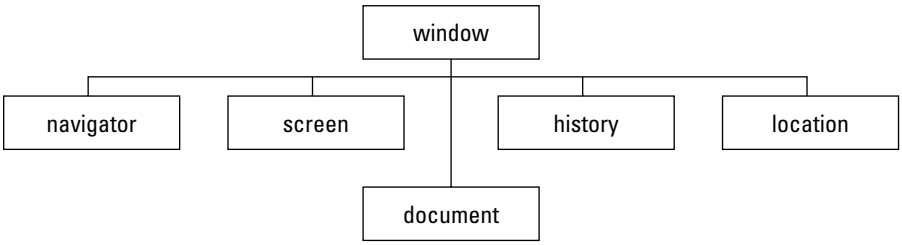


Figure 4-7: Basic object model for all modern browsers.

It's not important to memorize the model. But to give you a sense of the relationships among these top-level objects, the following describes their respective roles:

- **window object:** At the very top of the hierarchy is the window. This object represents the content area of the browser window where HTML documents appear. In a multiple-frame environment, each frame is also a window (but don't concern yourself with this just yet). Because all document action takes place inside the window, the window is the outermost element of the object hierarchy. Its physical borders contain the document.
- **navigator object:** This is the closest your scripts come to accessing the browser program, primarily to read the brand and version of browser that holds the current document. This object is read-only, protecting the browser from inappropriate access by rogue scripts.
- **screen object:** Another read-only object lets scripts learn about the physical environment in which the browser is running. For example, this object reveals the number of pixels high and wide available in the monitor.
- **history object:** While the browser maintains internal details about the browser's recent history (such as the list available under the Back button), scripts have no access to the details. At most this object assists a script in simulating a click of the Back or Forward button.
- **location object:** This object is the primary avenue to loading a different page into the current window or frame. URL information about the window is available under very controlled circumstances so that scripts cannot track access to other Web sites.
- **document object:** Each HTML document that gets loaded into a window becomes a `document` object. The `document` object contains the content that you are likely to script. Except for the `html`, `head`, and `body` element objects that are found in every HTML document, the precise makeup and structure of the element object hierarchy of the document depends on the content you put into the document.

When a Document Loads

Programming languages, such as JavaScript, are convenient intermediaries between your mental image of how a program works and the true inner workings of the computer. Inside the machine, every word of a program code listing influences the storage and movement of bits (the legendary 1s and 0s of the computer's binary universe) from one RAM storage slot to another. Languages and object models are inside the computer (or, in the case of JavaScript and the DOM, inside the browser's area of the computer) to make it easier for programmers to visualize how a program works and what its results will be. The relationship reminds me a lot of knowing how to drive an automobile from point A to point B without knowing exactly how an internal combustion engine, steering linkages, and all that other internal "stuff" works. By controlling high-level objects such as the ignition key, gearshift, gas pedal, brake, and steering wheel, I can get the results I need.

Of course, programming is not exactly like driving a car with an automatic transmission. Even scripting requires the equivalent of opening the hood and perhaps knowing how to check the transmission fluid or change the oil. Therefore, now it's time to open the hood and watch what happens to a document's object model as a page loads into the browser.

A simple document

Figure 4-8 shows the HTML and corresponding object model for a document that I'll be adding to in a moment. The figure shows only the `document` object portion — the `window` object and its other top-level objects (including the `document` object) are always there, even for an empty document. When this page loads, the browser maintains in its memory a map of the objects generated by the HTML tags in the document. At this point, only three objects exist, one for the outermost `html` element and its two nested elements.

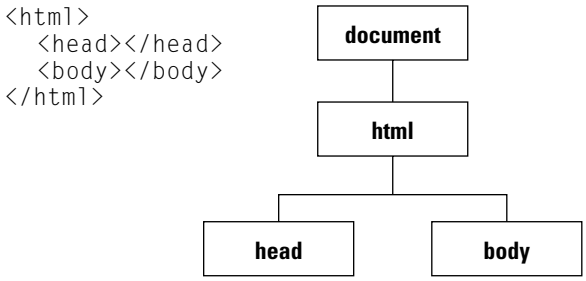


Figure 4-8: Object map of an empty document.

Add a paragraph element

Now, I modify the HTML file to include an empty paragraph element and reload the document. Figure 4-9 shows what happens to both the HTML (changes in boldface) and the object map as constructed by the browser. Even though no content appears in the paragraph, the `<p>` tags are enough to tell the browser to create that `p` element object. Also note that the `p` element object is contained by the `body` element object in the hierarchy of objects in the current map. In other words, the `p` element object is a child of the `body` element object. The object hierarchy matches the HTML tag containment hierarchy.

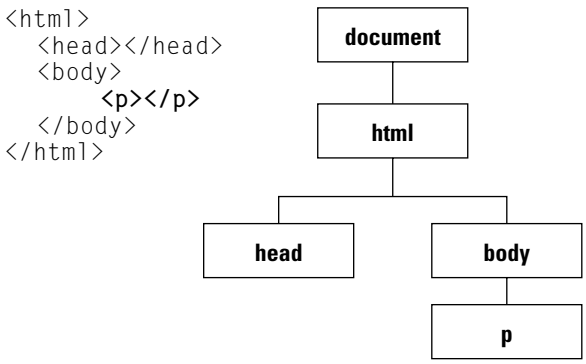


Figure 4-9: Adding an empty paragraph element.

Add paragraph text

I modify and reload the HTML file again, this time inserting the text of the paragraph between the element's start and end tags, as shown in Figure 4-10. A run of text extending between tags is a special kind of object in the DOM, called a *text node*. A text node always has an element acting as its container. Applying the official genealogy metaphor to this structure, the text node is a child of its parent `p` element. We now have a branch of the document object tree that runs several generations: `document->html->body->p->`text node.

```
<html>
  <head></head>
  <body>
      <p>This is the one and
        only paragraph.</p>
  </body>
</html>
```

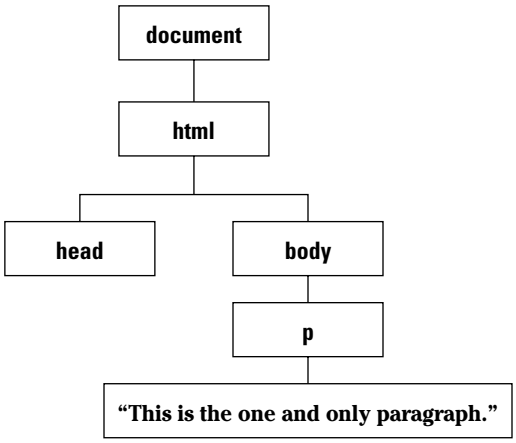


Figure 4-10: Adding a text node to the p element object.

Make a new element

The last modification I make to the file is to wrap a portion of the paragraph text in an `<em>` tag to signify emphasis for the enclosed text. This insertion has a large effect on the hierarchy of the `p` element object, as shown in Figure 4-11. The `p` element goes from having a single (text node) child to having three children: two text nodes with an element between them. In the W3C DOM, a text node cannot have any children and therefore cannot contain an element object. The bit of the text node now inside the `em` element is no longer a child of the `p` element, but rather a child of the `em` element. That text node is now a grandchild of the `p` element object.

Now that you see how objects are created in memory in response to HTML tags, the next step is to figure out how scripts can communicate with these objects. After all, scripting is mostly about controlling these objects.

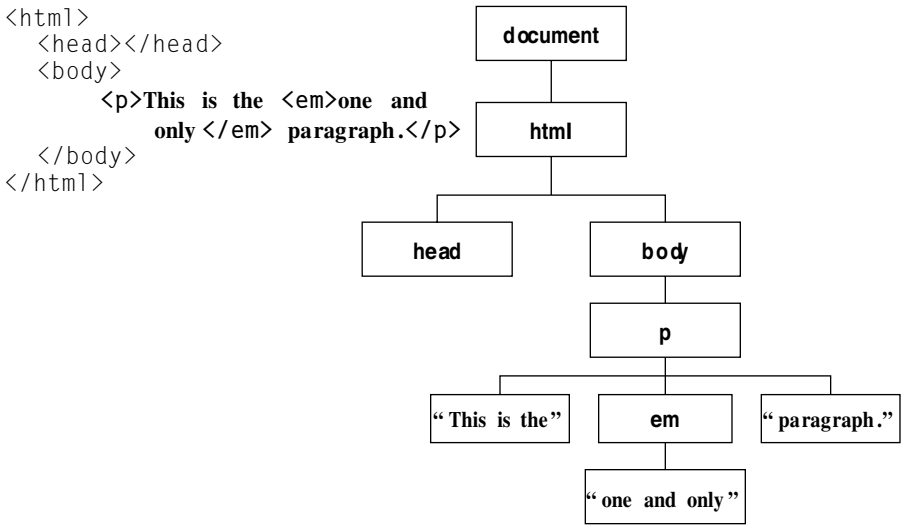


Figure 4-11: Inserting an element into a text node.

Object References

After a document is loaded into the browser, all of its objects are safely stored in memory in the containment hierarchy structure specified by the browser's document object model. For a script to control one of those objects, there must be a way to communicate with an object and find out something about it such as, “Hey, Mr. Text Field, what did the user type?” To let your scripts “talk to” an object, you need a way to refer to that object. That is precisely what an *object reference* in a script does for the browser.

Object naming

The biggest aid in creating script references to objects is assigning a name to every scriptable object in your HTML. In the W3C DOM (and current HTML specification), the way to assign a name to an element is by way of the `id` attribute. This attribute is optional, but if you plan to use scripts to access an element in the page, it is most convenient to assign a name to that element's `id` attribute directly in the HTML code. Here are some examples of `id` attributes added to typical tags:

```
<p id="firstParagraph" >

<img id="logo" src="images/logo.jpg" alt="Corp Logo">

<div class="draggable" id="puzzlePiece">
```

The only rules about object IDs (also called *identifiers*) are that they

- ✦ May not contain spaces
- ✦ Should not contain punctuation except for the underscore character
- ✦ Must be inside quotes when assigned to the `id` attribute
- ✦ Must not start with a numeric character
- ✦ May not occur more than once in the same document

Think of assigning IDs as the same as sticking nametags on everyone attending a conference meeting. To find a particular conference attendee whose name you know, you could wait at the entrance and scan each nametag until you find the name you're looking for; or you could bump around the attendees at random in the hope that you'll find a known name. But it would be more efficient if you had a way to immediately target an attendee by name — like broadcasting the name on the public address system to the whole crowd.

Referencing a particular object

The W3C DOM provides that kind of instant access to any named element in the document. If you haven't programmed before, the syntax for this access command may be intimidating by its length — a hazard when a standard such as the W3C DOM is designed by programmers. Like it or not, we're stuck with this syntax. Here is the syntax you will use frequently in your browser scripting:

```
window.document.getElementById("elementID")
```

You substitute the ID of the element you wish to reference for *elementID*. For example, if you want to reference the paragraph element whose ID is `firstParagraph`, the reference would be:

```
window.document.getElementById("firstParagraph")
```

Be careful! JavaScript is case-sensitive. Be sure you use uppercase for the three uppercase letters in the command, and use a lowercase "d" at the end, and that you capitalize the ID accurately as well.

The `getElementById()` command belongs to the `document` object, meaning that the entire document's collection of elements is subject to this instantaneous search for a matching ID. The dot — a traditional period character — is the JavaScript way of indicating that the item to the left of the dot (the `document` object here) has the item to the right of the dot (`getElementById()` here) as a resource to call upon whenever needed. Each type of object has a list of such resources, as you'll see in a moment (and as summarized in Appendix A).

id versus name Attributes

Prior to the HTML 4.0 specification's introduction of the `id` attribute, scripts could access a handful of elements that also supported the `name` attribute. Elements supporting the `name` attribute are predominantly related to forms, images, and frames. You will see how `name` attributes work in forms in Chapter 9. In fact, most browsers still require the `name` attribute for forms and form controls (text fields, buttons, and select lists) for their data to be submitted to a server. It is permissible to assign the same identifier to both the `id` and `name` attributes of an element.

Node Terminology

W3C DOM terminology uses metaphors to assist programmers in visualizing the containment hierarchy of a document and its content. One concept you should grasp early in your learning is that of a *node*; the other concept is the family relationship among objects in a document.

About nodes

While the English dictionary contains numerous definitions of "node," the one that comes closest to its application in the W3C DOM is the one that implies a knob or bump on a tree branch. Such nodules on a branch usually lead to one of two things: a leaf or another branch. A leaf is a dead end in that no further branches emanate from the leaf; but the branch kind of node leads to a new branch that can, itself, have further nodes, whether they be leaves or more branches. When you define the structure of an HTML document, you also define a node structure (also called a *node tree*) whose placement of branches and leaves depends entirely on your HTML elements and text content.

In the W3C DOM, the fundamental building block is a simple, generic node. But inside an HTML document, we work with special kinds of nodes that are tailored to HTML documents. The two types of nodes that scripts touch most often are element nodes and text nodes. These node types correspond exactly to HTML elements and the text that goes between an element's start and end tags. You've been working with element and text nodes in your HTML authoring, and you didn't even know it.

Look again at the simple document assembled earlier, along with its containment hierarchy diagram in Figure 4-12. All of the boxes representing HTML elements (`html`, `head`, `body`, `p`, and `em`) are element nodes; the three boxes containing actual text that appears in the rendered document are text nodes. You saw in the transition from one long text node (Figure 4-10) to the insertion of the `em` element (Figure 4-11) that the long text node divided into three pieces. Two text node pieces stayed in the same position in the hierarchy relative to the containing `p` element. The new `em` element bullied its way into the tree between the two text nodes, and shifted the third text node one level away from the `p` element.

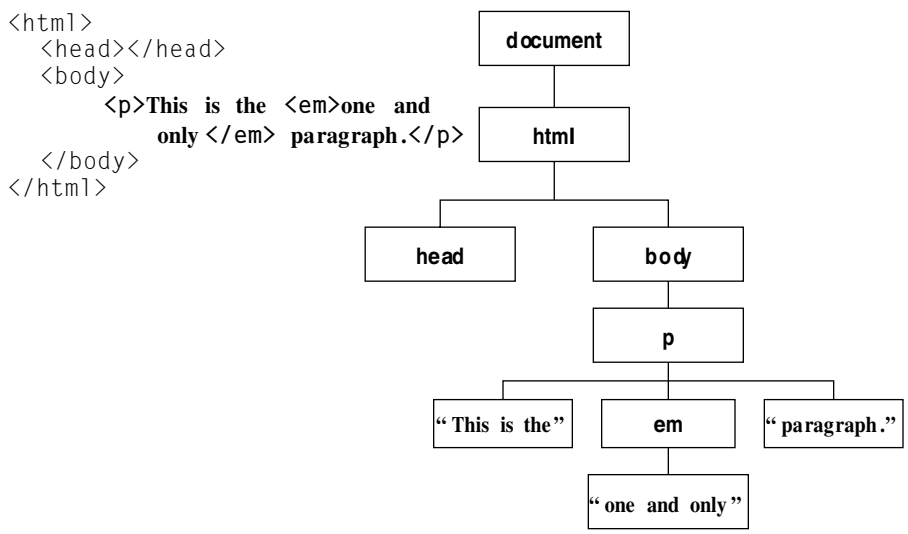


Figure 4-12: A simple HTML document node tree.

Parents and children

Looking more closely at the p element and its content in Figure 4-12, you can see that element has three *child* nodes. The first and last are of the text node type, while the middle one is an element node. When an element contains multiple child nodes, the sequence of child nodes is entirely dependent upon the HTML source code order. Thus, the first child node of the p element is the text node containing the text "This is the ". In the case of the em element, a single child text node is the sole descendant of the element.

Element node children are not always text nodes, nor do branches always end in text nodes. In Figure 4-12, the html element has two child nodes, both of which are element nodes; the body element has one child node, the p element. Even though the head element node appears to be at the end of a branch, it is still an element node because it is capable of containing other nodes (such as a title element). A tag in the HTML indicates an element node, whether or not it has any child nodes. In contrast, a text node can never contain another node — it's one of those dead-end leaf type of nodes.

Notice that a child node is always contained by one element node. That container is the *parent* node of its child or children. For example from the point of view of the em element node, it has both one child (a text node) and one parent (the p element node). A fair amount of W3C DOM terminology (which you'll meet in Chapter 14) concerns itself with assisting scripts to start at any point in a document hierarchy and obtain a reference to a related node if necessary. For instance, if a Dynamic HTML script wants to modify the text inside the em element of Figure 4-12, it would typically do so by starting with a reference to the em element via the document.getElementById() command (assuming the em element has an ID assigned to it) and then modifying the element's child node.

In case you're wondering, the document object at the top of the node tree is, itself, a node. Its place in the tree is special and is called, simply, the *document node*. Each document contains a single document node, and that node becomes the scripter's gateway to the rest of the document's nodes. It's no accident that the syntax for referencing an element node — document.getElementById() — begins with a reference to the document object.

What Defines an Object?

When an HTML tag defines an object in the source code, the browser creates a slot for that object in memory as the page loads. But an object is far more complex internally than, say, a mere number stored in memory. The purpose of an object is to represent some "thing." In the browser and its Document Object Model, the most common objects are those that correspond to elements, such as a text input form field, a table element, or the whole HTML document. Outside of the pared-down world of the DOM, an object can also represent abstract entities, such as a calendar program's appointment entry or a layer of graphical shapes in a drawing program. It is common for your browser scripts to work with both DOM objects and abstract objects of your own design.

Every type of DOM object is unique in some way, even if two or more objects look identical to you in the browser. Three very important facets of an object define what it is, what it looks like, how it behaves, and how scripts control it. Those three facets are properties, methods, and event handlers. They play such key roles in your future DOM scripting efforts that the Object Quick Reference in Appendix A summarizes the properties, methods, and event handlers for each object in the object models implemented in various browser generations.

Properties

Any physical object you hold in your hand has a collection of characteristics that defines it. A coin, for example, has shape, diameter, thickness, color, weight, embossed images on each side, and any number of other attributes that distinguish it from, say, a feather. Each of those features is called a *property*. Each property has a value of some kind attached to it (even if the value is empty or null). For example, the shape property of a coin might be "circle" — in this case, a text value. In contrast, the denomination property is most likely a numeric value.

You may not have known it, but if you've written HTML for use in a scriptable browser, you have set object properties without writing one iota of JavaScript. Tag attributes are the most common way to set an HTML element object's initial properties. For example, the following HTML tag defines an `input` element object that assigns four property values:

```
<input type="button" id="clicker" name="clicker" value="Hit Me...">
```

In JavaScript parlance, then, the `type` property holds the word "button," the `id` and `name` properties hold the same word, "clicker," and the `value` property is the text that appears on the button label, "Hit Me. . . ." In truth, a button input element has more properties than just these, but you don't have to set every property for every object. Most properties have default values that are automatically assigned if nothing special is set in the HTML or later from a script.

The contents of some properties can change after a document has loaded and the user interacts with the page. Consider the following text input tag:

```
<input type="text" id="entry" name="entry" value="User Name?">
```

The `id` and `name` properties of this object are the same word, "entry." When the page loads, the text of the `value` attribute setting is placed in the text field — the automatic behavior of an HTML text field when the `value` attribute is specified. But if a user enters some other text into the text field, the `value` property changes — not in the HTML, but in the memory copy of the object model that the browser maintains. Therefore, if a script queries the text field about the content of the `value` property, the browser yields the current setting of the property — which isn't necessarily the one specified by the HTML.

To gain access to an object's property, you use the same kind of dot notation addressing scheme you saw earlier for objects. A property is a resource belonging to its object, so the reference to it consists of the reference to the object plus one more extension naming the property. Therefore, for the button and text object tags just shown, references to various properties are

```
document.getElementById("clicker").name
document.getElementById("clicker").value
document.getElementById("entry").value
```

You may wonder what happened to the `window` part of the reference. It turns out that there can be only one document contained in a window, so references to objects inside the document can omit the `window` portion and start the reference with `document`. You cannot omit the `document` object, however, from the reference.

Internet Explorer References

Before the W3C DOM came into existence, Microsoft had created its own way of referencing element objects by way of their `id` attributes. You will find many instances of this syntax in existing code that has been written only for Internet Explorer 4 or later. The syntax uses a construction called `document.all`. Although there are a few different ways to use this construction, the most commonly applied way is to continue the dot notation to include the ID of the element. For example, if a paragraph element's ID is `myParagraph`, the IE-only reference syntax is:

```
document.all.myParagraph
```

You can also completely omit the lead-in parts of the reference, and simply refer to the ID of the element:

```
myParagraph
```

Be aware, however, that none of these approaches is supported in the W3C DOM standard. Both the IE-specific and W3C DOM reference syntax styles are implemented in IE5 or later. Going forward, you should migrate existing code to the W3C DOM style to be compatible with more browsers.

Methods

If a property is like a descriptive adjective for an object, then a method is a verb. A *method* is all about action related to the object. A method either does something to the object or with the object that affects other parts of a script or document. They are commands of a sort, but whose behaviors are tied to a particular object.

An object can have any number of methods associated with it (including none at all). To set a method into motion (usually called *invoking a method*), a JavaScript statement must include a reference to it — via its object with a pair of parentheses after the method name — as in the following examples:

```
document.getElementById("orderForm").submit()
document.getElementById("entry").focus()
```

The first is a scripted way of sending a form (named `orderForm`) to a server. The second gives focus to a text field named `entry`.

Sometimes a method requires that you send additional information with it so that it can do its job. Each chunk of information passed with the method is called a *parameter* or *argument* (you can use the terms interchangeably). You saw examples of passing a parameter in your first script in Chapter 3. Two script statements invoked the `write()` method of the `document` object:

```
document.write("This browser is version " + navigator.appVersion)
document.write(" of <B>" + navigator.appName + "</B>.")
```

As the page loaded into the browser, each `document.write()` method sent whatever text was inside the parentheses to the current document. In both cases, the content being sent as a parameter consisted of straight text (inside quotes) and the values of two object properties: the `appVersion` and `appName` properties of the `navigator` object.

Some methods require more than one parameter. If so, the multiple parameters are separated by commas. For example, version 4 and later browsers support a `window` object method that

moves the window to a particular coordinate point on the screen. A coordinate point is defined by two numbers that indicate the number of pixels from the left and top edges of the screen where the top-left corner of the window should be. To move the browser window to a spot 50 pixels from the left and 100 pixels from the top, the method is:

```
window.moveTo(50,100)
```

As you learn more about the details of JavaScript and the document objects you can script, pay close attention to the range of methods defined for each object. They reveal a lot about what an object is capable of doing under script control.

Event handlers

One last characteristic of a DOM object is the *event handler. Events* are actions that take place in a document, usually as the result of user activity. Common examples of user actions that trigger events include clicking a button or typing a character into a text field. Some events, such as the act of loading a document into the browser window or experiencing a network error while an image loads, are not so obvious.

Almost every DOM object in a document receives events of one kind or another — summarized for your convenience in the Object Quick Reference of Appendix A. What determines whether the object does anything in response to the event is an extra attribute you enter into the object's HTML definition. The attribute consists of the event name, an equal sign (just like any HTML attribute), followed by instructions about what to do when the particular event fires. Listing 4-1 shows a very simple document that displays a single button with one event handler defined for it.

Listing 4-1: **A Simple Button with an Event Handler**

```
<html>
<body>
<form>
<input type="button" value="Click Me" onclick="window.alert ('Ouch!')">
</form>
</body>
</html>
```

The form definition contains what, for the most part, looks like a standard `input` element. But notice the last attribute, `onclick="window.alert('Ouch!')"`. Button input objects, as you see in their complete descriptions in Chapter 22, react to mouse clicks. When a user clicks the button, the browser sends a click event to the button. In this button's definition, the attribute says that whenever the button receives that event, it should invoke one of the `window` object's methods, `alert()`. The `alert()` method displays a simple alert dialog box whose content is whatever text is passed as a parameter to the method. Like most arguments to HTML attributes, the attribute setting to the right of the equal sign goes inside quotes. If additional quotes are necessary, as in the case of the text to be passed along with the event handler, those inner quotes can be single quotes. In actuality, JavaScript doesn't distinguish between single or double quotes but does require that each pair be of the same type. Therefore, you can write the attribute this way:

```
onclick='alert("Ouch!")'
```

Exercises

1. Which of the following applications are well suited to client-side JavaScript? Why or why not?

   a. Music jukebox

   b. Web-site visit counter

   c. Chat room

   d. Graphical Fahrenheit-to-Celsius temperature calculator

   e. All of the above

   f. None of the above

2. Which of the following object names are valid in JavaScript? For each one that is invalid, explain why.

   a. `lastName`

   b. `company_name`

   c. `1stLineAddress`

   d. `zip code`

   e. `today's_date`

3. Using the diagram from Figure 4-12 for reference, draw a diagram of the object model containment hierarchy that the browser would create in its memory for the following HTML. Write the script reference to the second paragraph element using W3C DOM syntax.

```
<html>
<head>
<title>Search Form</title>
</head>
<body>
<p id="logoPar"><img src="images/logo.jpg" height="90" width="300"
alt="Logo" /></p>
<p id="formPar">
<form name="searchForm" action="cgi-bin/search.pl" method="POST">
Search for: <input type="text" name="searchText" />
<input type="submit" value="Search" />
</form>
</p>
</body>
</html>
```

4. Describe at least two characteristics that a text node and an element node have in common; describe at least two characteristics that distinguish a text node from an element node.

5. Write the HTML tag for a button input element named “Hi,” whose visible label reads “Howdy” and whose `onclick` event handler displays an alert dialog box that says “Hello to you, too!”

✦ ✦ ✦

CHAPTER 5

Scripts and HTML Documents

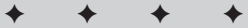
In This Chapter

Where to place scripts in HTML documents

What a JavaScript statement is

What makes a script run

Viewing script errors



In this chapter's tutorial, you begin to see how scripts are embedded within HTML documents and what comprises a script statement. You also see how script statements can run when the document loads or in response to user action. Finally, you find out where script error information is hiding.

Where Scripts Go in Documents

Chapter 4 did not thoroughly cover what scripts look like or how you add them to an HTML document. That's where this lesson picks up the story.

The <script> tag

To assist the browser in recognizing lines of code in an HTML document as belonging to a script, you surround lines of script code with a `<script>...</script>` tag set. This is common usage in HTML where start and end tags encapsulate content controlled by that tag, whether the tag set is for a form or a paragraph.

Depending on the browser, the `<script>` tag has a variety of attributes you can set that govern the script. One attribute, `type`, advises the browser to treat the code within the tag as JavaScript. Some other browsers accept additional languages (such as Microsoft's VBScript in Windows versions of Internet Explorer). The following setting is one that all scriptable browsers accept:

```
<script type="text/javascript">
```

Be sure to include the ending tag for the script. Lines of JavaScript code go between the two tags:

```
<script  type="text/javascript">
  one or more lines of JavaScript code here
</script>
```

If you forget the closing script tag, the script may not run properly and the HTML elsewhere in the page may look strange.

Although you don't work with it in this tutorial, another attribute works with more recent browsers to blend the contents of an external script file into the current document. An `src` attribute (similar to

the src attribute of an <img> tag) points to the file containing the script code. Such files must end with a .js extension. The tag set looks like the following:

```
<script  type="text/javascript" SRC="myscript.js"></script>
```

All script lines are in the external file, so no script lines are included between the start and end script tags in the document.

Tag positions

Where do these tags go within a document? The answer is, anywhere they're needed in the document. Most of the time it makes sense to include the tags nested within the <head>...</head> tag set; other times it is essential that you drop the script into a very specific location in the <body>...</body> section.

In the following four listings, I demonstrate — with the help of a skeletal HTML document — some of the possibilities of <script> tag placement. Later in this lesson, you see why scripts may need to go in different places within a page depending on the scripting requirements.

Listing 5-1 shows the outline of what may be the most common position of a <script> tag set in a document: in the <head> tag section. Typically, the Head is a place for tags that influence noncontent settings for the page — so-called HTML "directive" elements, such as <meta> tags and the document title. It turns out that this is also a convenient place to plant scripts that are called on in response to user action.

Listing 5-1: **Scripts in the Head**

```
<html>
<head>
<title>A Document</title>
<script  type="text/javascript">
   //script statement(s) here
   ...
</script>
</head>
<body>
</body>
</html>
```

The Old language Attribute

Another <script> tag attribute, language, used to be the way to specify the scripting language for the enclosed code. That attribute allowed scripters to specify the language version. For example, if the scripts included code that required JavaScript syntax available only in version 4 browsers (which implemented JavaScript version 1.2), the <script> tag used to be written as follows:

```
<script language="JavaScript1.2">...</script>
```

The language attribute was never part of the HTML 4.0 specification, and is now falling out of favor. If W3C validation is one of your development concerns, the attribute does not validate in strict versions of HTML 4.01 or XHTML 1.0. Older browsers that do not know the type attribute automatically default to JavaScript anyway. Use only the type attribute.

On the other hand, if you need a script to run as the page loads so that the script generates content in the page, the script goes in the <body> portion of the document, as shown in Listing 5-2. If you check the code listing for your first script in Chapter 3, you see that the script tags are in the Body because the script needs to fetch information about the browser and write the results to the page as the page loads.

Listing 5-2: **A Script in the Body**

```
<html>
<head>
<title>A Document</title>
</head>
<body>
<script  type="text/javascript">
   //script statement(s) here
   ...
</script>
</body>
</html>
```

It's also good to know that you can place an unlimited number of <script> tag sets in a document. For example, Listing 5-3 shows a script in both the Head and Body portions of a document. Perhaps this document needs the Body script to create some dynamic content as the page loads, but the document also contains a button that needs a script to run later. That script is stored in the Head portion.

Listing 5-3: **Scripts in the Head and Body**

```
<html>
<head>
<title>A Document</title>
<script  type="text/javascript">
   //script statement(s) here
   ...
</script>
</head>
<body>
<script  type="text/javascript">
   //script statement(s) here
   ...
</script>
</body>
</html>
```

You also are not limited to one <script> tag set in either the Head or Body. You can include as many <script> tag sets in a document as are needed to complete your application. In Listing 5-4, for example, two <script> tag sets are located in the Body portion, with some other HTML between them.

Listing 5-4: **Two Scripts in the Body**

```
<html>
<head>
<title>A Document</title>
</head>
<body>
<script  type="text/javascript">
   //script statement(s) here
   ...
</script>
<more html>
<script  type="text/javascript">
   //script statement(s) here
   ...
</script>
</body>
</html>
```

Handling non-JavaScript browsers

Only browsers that include JavaScript in them know to interpret the lines of code between the `<script>...</script>` tag pair as script statements and not HTML text for display in the browser. This means that a pre-JavaScript browser or a simplified browser in a cell phone not only ignores the tags, but it also treats the JavaScript code as page content. The results can be disastrous to a page.

You can reduce the risk of non-JavaScript browsers displaying the script lines by playing a trick. The trick is to enclose the script lines between HTML comment symbols, as shown in Listing 5-5. Most nonscriptable browsers completely ignore the content between the `<!--` and `-->` comment tags, whereas scriptable browsers ignore those comment symbols when they appear inside a `<script>` tag set.

Listing 5-5: **Hiding Scripts from Most Old Browsers**

```
<script  type="text/javascript">
<!--
   //script statement(s) here
   ...
// -->
</script>
```

The odd construction right before the ending script tag needs a brief explanation. The two forward slashes are a JavaScript comment symbol. This symbol is necessary because JavaScript otherwise tries to interpret the components of the ending HTML symbol (`-->`). Therefore, the forward slashes tell JavaScript to skip the line entirely; a nonscriptable browser simply treats those slash characters as part of the entire HTML comment to be ignored.

Despite the fact that this technique is often called *hiding scripts*, it does not disguise the scripts entirely. All client-side JavaScript scripts are part of the HTML document and download to the browser just like all other HTML. Furthermore, you can view them as part of the document's source code. Do not be fooled into thinking that you can hide your scripts entirely from prying eyes.

JavaScript Statements

Virtually every line of code that sits between a `<script>... </script>` tag pair is a JavaScript *statement*. To be compatible with habits of experienced programmers, JavaScript accepts a semicolon at the end of every statement (the computer equivalent of a period at the end of a sentence). Fortunately for newcomers, this semicolon is optional: The carriage return at the end of a statement suffices for JavaScript to know the statement has ended. It is possible that in the future the semicolon will be required, so it's a good idea to get into the semicolon habit now.

A statement must be in the script for a purpose. Therefore, every statement does "something" relevant to the script. The kinds of things that statements do are

- ✦ Define or initialize a variable
- ✦ Assign a value to a property or variable
- ✦ Change the value of a property or variable
- ✦ Invoke an object's method
- ✦ Invoke a function routine
- ✦ Make a decision

If you don't yet know what all of these mean, don't worry — you will by the end of this tutorial. The point I want to stress is that each statement contributes to the scripts you write. The only statement that doesn't perform any explicit action is the *comment*. A pair of forward slashes (no space between them) is the most common way to include a comment in a script. You add comments to a script for your benefit. They usually explain in plain language what a statement or group of statements does. The purpose of including comments is to remind you six months from now how your script works.

When Script Statements Execute

Now that you know where scripts go in a document, it's time to look at when they run. Depending on what you need a script to do, you have four choices for determining when a script runs:

- ✦ While a document loads
- ✦ Immediately after a document loads
- ✦ In response to user action
- ✦ When called upon by other script statements

The determining factor is how the script statements are positioned in a document.

While a document loads — immediate execution

Your first script in Chapter 3 (reproduced in Listing 5-6) runs while the document loads into the browser. For this application, it is essential that a script inspects some properties of the `navigator` object and includes those property values in the content being rendered for the page as it loads. It makes sense, therefore, to include the `<script>` tags and statements in the Body portion of the document. I call the kind of statements that run as the page loads *immediate statements*.

Listing 5-6: HTML Page with Immediate Script Statements

```
<html>
<head>
<title>My First Script</title>
<style type="text/css">
.highlight {font-weight: bold}
</style>
</head>

<body>
<h1>Let's Script...</h1>
<hr>
<script type="text/javascript">
<!-- hide from old browsers
document.write("This browser is version " + navigator.appVersion);
document.write(" of <span class='highlight'>" + navigator.appName + "</span>.");
// end script hiding -->
</script>
</body>
</html>
```

Deferred scripts

The other three ways that script statements run are grouped together as what I call *deferred scripts*. To demonstrate these deferred script situations, I must introduce you briefly to a concept covered in more depth in Chapter 7: the function. A *function* defines a block of script statements summoned to run some time after those statements load into the browser. Functions are clearly visible inside a `<script>` tag because each function definition begins with the word `function` followed by the function name (and parentheses). Once a function is loaded into the browser (commonly in the Head portion so it loads early), it stands ready to run whenever called upon.

One of the times a function is called upon to run is immediately after a page loads. The `window` object has an event handler called `onload`. Unlike most event handlers, which are triggered in response to user action (for example, clicking a button), the `onload` event handler fires the instant that all of the page's components (including images, Java applets, and embedded multimedia) are loaded into the browser. The `onload` event handler goes in the `<body>` tag, as shown in Listing 5-7. Recall from Chapter 4 (Listing 4-1) that an event handler can run a script statement directly. But if the event handler must run several script statements, it is usually more convenient to put those statements in a function definition and then have the event handler *invoke* that function. That's what happens in Listing 5-7: When the page completes loading, the `onload` event handler triggers the `done()` function. That function (simplified for this example) displays an alert dialog box.

Listing 5-7: **Running a Script from the onload Event Handler**

```
<html>
<head>
<title>An onload script</title>
<script  type="text/javascript">
<!--
function done() {
   alert("The page has finished loading.");
}
// -->
</script>
</head>
<body onload="done()">
Here is some body text.
</body>
</html>
```

Don't worry about the curly braces or other oddities in Listing 5-7 that may cause you concern at this point. Focus instead on the structure of the document and the flow. The entire page loads without running any script statements, although the page loads the `done()` function in memory so that it is ready to run at a moment's notice. After the document loads, the browser fires the `onload` event handler, which causes the `done()` function to run. Then the user sees the alert dialog box.

Getting a script to execute in response to a user action is very similar to the preceding example for running a deferred script right after the document loads. Commonly, a script function is defined in the Head portion, and an event handler in, say, a form element calls upon that function to run. Listing 5-8 includes a script that runs when a user clicks a button.

Listing 5-8: **Running a Script from User Action**

```
<html>
<head>
<title>An onclick script</title>
<script type="text/javascript">
<!--
function alertUser() {
   alert("Ouch!");
}
// -->
</script>
</head>
<body>
Here is some body text.
<form>
   <input type="text" name="entry">
   <input type="button" name="oneButton" value="Press Me!"
onclick="alertUser()">
</form>
</body>
</html>
```

Not every object must have an event handler defined for it in the HTML, as shown in Listing 5-8 — only the ones for which scripting is needed. No script statements execute in Listing 5-8 until the user clicks the button. The `alertUser()` function is defined as the page loads, and it waits to run as long as the page remains loaded in the browser. If it is never called upon to run, there's no harm done.

The last scenario for when script statements run also involves functions. In this case, a function is called upon to run by another script statement. Before you see how that works, it helps to read through the next lesson (Chapter 6). Therefore, I will hold off on this example until later in the tutorial.

Viewing Script Errors

In the early days of JavaScript in browsers, script errors displayed themselves in very obvious dialog boxes. These boxes were certainly helpful for scripters who wanted to debug their scripts. However, if a bug got through to a page served up to a non-technical user, the error alert dialog boxes were not only disruptive, but also scary. To prevent such dialog boxes from disturbing unsuspecting users, the browser makers tried to diminish the visual impact of errors in the browser window. Unfortunately for scripters, it is often easy to overlook the fact that your script contains an error because the error is not so obvious. Recent browser versions have different ways of letting scripters see the errors.

In IE5+, you can set its preferences so that scripts do not generate error dialog boxes (go to Tools ➪ Internet Options ➪ Advanced ➪ Browsing and find the check-box entry that says "Display a notification about every script error"). Even with error dialog boxes turned off, error indications are displayed subtly at the left edge of the browser window's status bar. An alert icon and message ("Error on page.") appear in the status bar. If you double-click the icon, the error dialog box appears (see Figure 5-1). Be sure to expand the dialog box by clicking the Show Details button. Unless you turn on script error dialog boxes and keep them coming, you have to train yourself to monitor the status bar when a page loads and after each script runs.

Figure 5-1: The expanded IE error dialog box.

For Mozilla-based browsers, choose Tools ➪ Web Development ➪ JavaScript Console. The JavaScript console window (a separate window from the Java console) opens to reveal the error message details (see Figure 5-2). You can keep this window open all the time if you like. Unless you clear the window, subsequent error messages are appended to the bottom of the window.

Figure 5-2: The Mozilla 1.4 JavaScript console window.

Safari 1.0 records script errors, but it's not obvious how to read them. You must first enable Safari's Debug menu by typing the following command in the Terminal application:

```
defaults write com.apple.Safari IncludeDebugMenu 1
```

Then, each time you launch Safari, choose the Log JavaScript Exceptions item in the Debug menu. Open the MacOS X Console application window, where JavaScript error messages appear amid other Console logging messages. With luck, future versions will be more developer-friendly.

Understanding error messages and doing something about them is a very large subject, reserved for advanced discussion in Chapter 45 on the CD-ROM. During this tutorial, however, you can use the error messages to see if you have perhaps mistyped a script from a listing in the book.

Scripting versus Programming

You may get the impression that scripting is easier than programming. "Scripting" simply sounds easier or more friendly than "programming." In many respects, this is true. One of my favorite analogies is the difference between a hobbyist who builds model airplanes from scratch and a hobbyist who builds model airplanes from commercial kits. The "from scratch" hobbyist carefully cuts and shapes each piece of wood and metal according to very detailed plans before the model starts to take shape. The commercial kit builder starts with many prefabricated parts and assembles them into the finished product. When both builders are finished, you may not be able to tell which airplane was built from scratch and which one came out of a box of components. In the end, both builders used many of the same techniques to complete the assembly, and each can take pride in the result.

As you've seen with the Document Object Model, the browser gives scripters many prefabricated components with which to work. Without the browser, you'd have to be a pretty good programmer to develop from scratch your own application that served up content and

offered user interaction. In the end, both authors have working applications that look equally professional.

Beyond the DOM, however, "real programming" nibbles its way into the scripting world. That's because scripts (and programs) work with more than just objects. When I said earlier in this lesson that each statement of a JavaScript script does something, that "something" involves *data* of some kind. Data is the information associated with objects or other pieces of information that a script pushes around from place to place with each statement.

Data takes many forms. In JavaScript, the common incarnations of data are numbers; text (called *strings*); objects (both from the object model and others you can create with scripts); and `true` and `false` (called *Boolean values*).

Each programming or scripting language determines numerous structures and limits for each kind of data. Fortunately for newcomers to JavaScript, the universe of knowledge necessary for working with data is smaller than in a language such as Java. At the same time, what you learn about data in JavaScript is immediately applicable to future learning you may undertake in any other programming language — don't believe for an instant that your efforts in learning scripting will be wasted.

Because deep down scripting is programming, you need to have a basic knowledge of fundamental programming concepts to consider yourself a good JavaScript scripter. In the next two lessons, I set aside most discussion about the DOM and focus on the programming principles that will serve you well in JavaScript and future programming endeavors.

Exercises

1. Write the complete script tag set for a script whose lone statement is

   ```
   document.write("Hello, world.");
   ```

2. Build an HTML document and include the answer to the previous question such that the page executes the script as it loads. Open the document in your browser to test the results.

3. Add a comment to the script in the previous answer that explains what the script does.

4. Create an HTML document that displays an alert dialog box immediately after the page loads and displays a different alert dialog box when the user clicks a form button.

5. Carefully study the document in Listing 5-9. Without entering and loading the document, predict

   a. What the page looks like

   b. How users interact with the page

   c. What the script does

   Then type the listing into a text editor as shown (observe all capitalization and punctuation). **Do not type a carriage return after the "=" sign in the `upperMe` function statement; let the line word-wrap as it does in the following listing.** It's okay to use a carriage return between attribute name/value pairs, as shown in the first `<input>` tag. Save the document as an HTML file, and load the file into your browser to see how well you did.

Listing 5-9: **How Does This Page Work?**

```
<html>
<head>
<title>Text Object Value</title>
<script type="text/javascript">
<!--
function upperMe() {
    document.getElementById("output").value =
document.getElementById("input").value.toUpperCase();
}
// -->
</script>
</head>

<body>
Enter lowercase letters for conversion to uppercase:<br>
<form name="converter">
    <input type="text" name="input" id="input"
        value="sample" onchange="upperMe()" /><br />
    <input type="text" name="output" id="output" value="" />
</form>
</body>
</html>
```

✦ ✦ ✦

CHAPTER 6

Programming Fundamentals, Part I

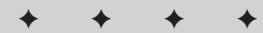
In This Chapter

What variables are and how to use them

Why you must learn how to evaluate expressions

How to convert data from one type to another

How to use basic operators



The tutorial breaks away from HTML and documents for a while as you begin to learn programming fundamentals that apply to practically every scripting and programming language you will encounter. Here, you start learning about variables, expressions, data types, and operators — things that might sound scary if you haven't programmed before. Don't worry. With a little practice, you will become quite comfortable with these terms and concepts.

What Language Is This?

The language you're studying is called JavaScript. But the language has some other names that you may have heard. JScript is Microsoft's name for the language. By leaving out the "ava," the company doesn't have to license the "Java" name from its trademark owner: Sun Microsystems.

A standards body called ECMA (pronounced ECK-ma) now governs the specifications for the language (no matter what you call it). The document that provides all of the details about the language is known as *ECMA-262* (it's the 262nd standard published by ECMA). Both JavaScript and JScript are ECMA-262 compatible. Some earlier browser versions exhibit very slight deviations from ECMA-262 (which came later than the earliest browsers). The most serious discrepancies are noted in the core language reference in Part IV of this book.

Working with Information

With rare exception, every JavaScript statement you write does something with a hunk of information — *data*. Data may be text information displayed on the screen by a JavaScript statement or the on/off setting of a radio button in a form. Each single piece of information in programming is also called a *value*. Outside of programming, the term *value* usually connotes a number of some kind; in the programming world, however, the term is not as restrictive. A string

of letters is a value. A number is a value. The setting of a checkbox (whether it is checked or not) is a value.

In JavaScript, a value can be one of several types. Table 6-1 lists JavaScript's formal data types, with examples of the values you will see displayed from time to time.

Table 6-1: JavaScript Value (Data) Types

Type	*Example*	*Description*
String	`"Howdy"`	A series of characters inside quote marks
Number	`4.5`	Any number not inside quote marks
Boolean	`true`	A logical true or false
Null	`null`	Completely devoid of any content, but a value just the same
Object		A software "thing" that is defined by its properties and methods (arrays are also objects)
Function		A function definition

A language that contains these few data types simplifies programming tasks, especially those involving what other languages consider to be incompatible types of numbers (integers versus real or floating-point values). In some definitions of syntax and parts of objects later in this book, I make specific reference to the type of value accepted in placeholders. When a string is required, any text inside a set of quotes suffices.

You will encounter situations, however, in which the value type may get in the way of a smooth script step. For example, if a user enters a number into a form's text input field, the browser stores that number as a string value type. If the script is to perform some arithmetic on that number, you must convert the string to a number before you can apply the value to any math operations. You see examples of this later in this lesson.

Variables

Cooking up a dish according to a recipe in the kitchen has one advantage over cooking up some data in a program. In the kitchen, you follow recipe steps and work with real things: carrots, milk, or a salmon fillet. A computer, on the other hand, follows a list of instructions to work with data. Even if the data represents something that looks real, such as the text entered into a form's input field, once the value gets into the program, you can no longer reach out and touch it.

In truth, the data that a program works with is merely a collection of bits (on and off states) in your computer's memory. More specifically, data in a JavaScript-enhanced Web page occupies parts of the computer's memory set aside for exclusive use by the browser software. In the olden days, programmers had to know the numeric address in memory (RAM) where a value was stored to retrieve a copy of it for, say, some addition. Although the innards of a program have that level of complexity, programming languages such as JavaScript shield you from it.

The most convenient way to work with data in a script is to first assign the data to a *variable*. It's usually easier to think of a variable as a basket that holds information. How long the variable holds the information depends on a number of factors. But the instant a Web page clears the window (or frame), any variables it knows about are immediately discarded.

Creating a variable

You have a couple of ways to create a variable in JavaScript, but one covers you properly in all cases. Use the `var` keyword, followed by the name you want to give that variable. Therefore, to *declare* a new variable called `myAge`, the JavaScript statement is

```
var myAge;
```

That statement lets the browser know that you can use that variable later to hold information or to modify any of the data in that variable.

To assign a value to a variable, use one of the *assignment operators*. The most common one by far is the equal sign. If I want to assign a value to the `myAge` variable at the same time I declare it (a combined process known as *initializing the variable*), I use that operator in the same statement as the `var` keyword:

```
var myAge = 45;
```

On the other hand, if I declare a variable in one statement and later want to assign a value to it, the sequence of statements is

```
var myAge;
myAge = 45;
```

Use the `var` keyword **only for declaration or initialization —** once for the life of any variable name in a document.

A JavaScript variable can hold any value type. Unlike many other languages, you don't have to tell JavaScript during variable declaration what type of value the variable will hold. In fact, the value type of a variable can change during the execution of a program. (This flexibility drives experienced programmers crazy because they're accustomed to assigning both a data type and a value to a variable.)

Variable names

Choose the names you assign to variables with care. You'll often find scripts that use vague variable names, such as single letters. Other than a few specific times where using letters is a common practice (for example, using `i` as a counting variable in repeat loops in Chapter 7), I recommend using names that truly describe a variable's contents. This practice can help you follow the state of your data through a long series of statements or jumps, especially for complex scripts.

A number of restrictions help instill good practice in assigning names. First, you cannot use any reserved keyword as a variable name. That includes all keywords currently used by the language and all others held in reserve for future versions of JavaScript. The designers of JavaScript, however, cannot foresee every keyword that the language may need in the future. By using the kind of single words that currently appear in the list of reserved keywords (see Appendix B), you always run a risk of a future conflict.

To complicate matters, a variable name cannot contain space characters. Therefore, one-word variable names are fine. Should your description really benefit from more than one word, you can use one of two conventions to join multiple words as one. One convention is to place an underscore character between the words; the other is to start the combination word with a lowercase letter and capitalize the first letter of each subsequent word within the name — I refer to this as the *interCap format*. Both of the following examples are valid variable names:

```
my_age
myAge
```

My preference is for the second version. I find it easier to type as I write JavaScript code and easier to read later. In fact, because of the potential conflict with future one-word keywords, using multiword combinations for variable names is a good idea. Multiword combinations are less likely to appear in the reserved word list.

Variable names have a couple of other important restrictions. Avoid all punctuation symbols except for the underscore character. Also, the first character of a variable name cannot be a numeral. If these restrictions sound familiar, it's because they're identical to those for HTML element identifiers described in Chapter 4.

Expressions and Evaluation

Another concept closely related to the value and variable is *expression evaluation* — perhaps the most important concept of learning how to program a computer.

We use expressions in our everyday language. Remember the theme song of *The Beverly Hillbillies*?

Then one day he was shootin' at some food

And up through the ground came a-bubblin' crude

Oil that is. Black gold. Texas tea.

At the end of the song, you find four quite different references ("crude," "oil," "black gold," and "Texas tea"). They all mean oil. They're all *expressions* for oil. Say any one of them and other people know what you mean. In our minds, we *evaluate* those expressions to mean one thing: oil.

In programming, a variable always evaluates to its contents, or value. For example, after assigning a value to a variable, such as

```
var myAge = 45;
```

any time the variable is used in a statement, its value (45) is automatically applied to whatever operation that statement calls. Therefore, if you're 15 years my junior, I can assign a value to a variable representing your age based on the evaluated value of `myAge`:

```
var yourAge = myAge - 15;
```

The variable, `yourAge`, evaluates to 30 the next time the script uses it. If the `myAge` value changes later in the script, the change has no link to the `yourAge` variable because `myAge` evaluated to 45 when it was used to assign a value to `yourAge`.

Expressions in script1.htm

You probably didn't recognize it at the time, but you saw how expression evaluation came in handy in your first script of Chapter 3. Recall the second `document.write()` statement:

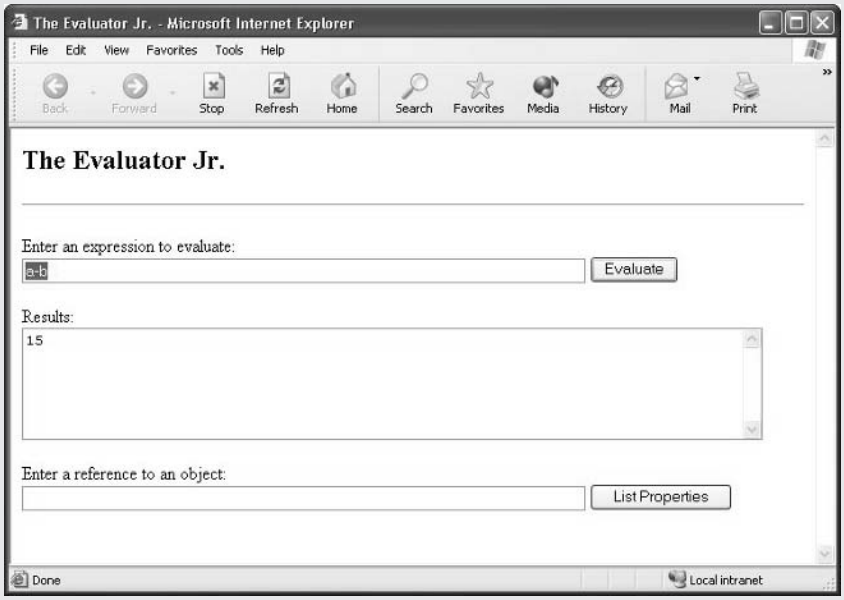
```
document.write(" of " + navigator.appName + ".");
```

The `document.write()` method (remember, JavaScript uses the term *method* to mean *command*) requires a parameter in the parentheses: the text string to be displayed on the Web page. The parameter here consists of one expression that joins three distinct strings:

```
" of "
navigator.appName
"."
```

Testing JavaScript Evaluation

You can begin experimenting with the way JavaScript evaluates expressions with the help of The Evaluator Jr. (seen in the following figure), an HTML page you can find on the companion CD-ROM. (I introduce the Senior version in Chapter 13.) Enter any JavaScript expression into the top text box, and either press Enter/Return or click the Evaluate button.



The Evaluator Jr. for testing expression evaluation.

The Evaluator Jr. has 26 variables (lowercase `a` through `z`) predefined for you. Therefore, you can assign values to variables, test comparison operators, and even do math here. Using the `age` variable examples from earlier in this chapter, type each of the following statements into the upper text box and observe how each expression evaluates in the Results field. Be sure to observe case-sensitivity in your entries. The trailing semicolons are optional in The Evaluator.

```
a = 45;
a;
b = a - 15;
b;
a - b;
a > b;
```

To start over, click the Reload button.

The plus symbol is one of JavaScript's ways of joining strings. Before JavaScript can display this line, it must perform some quick evaluations. The first evaluation is the value of the `navigator.appName` property. This property evaluates to a string of the name of your browser. With that expression safely evaluated to a string, JavaScript can finish the job of joining the three strings in the final evaluation. The evaluated string expression is what ultimately appears on the Web page.

Expressions and variables

As one more demonstration of the flexibility that expression evaluation offers, this section shows you a slightly different route to the `document.write()` statement. Rather than join those strings as the direct parameter to the `document.write()` method, I can gather the strings in a variable and then apply the variable to the `document.write()` method. Here's how that sequence looks, as I simultaneously declare a new variable and assign it a value:

```
var textToWrite = " of " + navigator.appName + ".";
document.write(textToWrite);
```

This method works because the variable, `textToWrite`, evaluates to the combined string. The `document.write()` method accepts that string value and does its display job. As you read a script or try to work through a bug, pay special attention to how each expression (variable, statement, object property) evaluates. I guarantee that as you learn JavaScript (or any language), you will end up scratching your head from time to time because you haven't stopped to examine how expressions evaluate when a particular kind of value is required in a script.

Data Type Conversions

I mentioned earlier that the type of data in an expression can trip up some script operations if the expected components of the operation are not of the right type. JavaScript tries its best to perform internal conversions to head off such problems, but JavaScript cannot read your mind. If your intentions differ from the way JavaScript treats the values, you won't get the results you expect.

A case in point is adding numbers that may be in the form of text strings. In a simple arithmetic statement that adds two numbers together, you get the expected result:

```
3 + 3          // result = 6
```

But if one of those numbers is a string, JavaScript leans toward converting the other value to a string — thus turning the plus sign's action from arithmetic addition to joining strings. Therefore, in the statement

```
3 + "3"        // result = "33"
```

the "string-ness" of the second value prevails over the entire operation. The first value is automatically converted to a string, and the result joins the two strings. Try this yourself in The Evaluator Jr.

If I take this progression one step further, look what happens when another number is added to the statement:

```
3 + 3 + "3"    // result = "63"
```

This might seem totally illogical, but there is logic behind this result. The expression is evaluated from left to right. The first plus operation works on two numbers, yielding a value of 6.

But as the 6 is about to be added to the "3," JavaScript lets the "string-ness" of the "3" rule. The 6 is converted to a string, and two string values are joined to yield "63."

Most of your concern about data types will focus on performing math operations like the ones here. However, some object methods also require one or more parameters of particular data types. While JavaScript provides numerous ways to convert data from one type to another, it is appropriate at this stage of the tutorial to introduce you to the two most common data conversions: string to number and number to string.

Converting strings to numbers

As you saw in the last section, if a numeric value is stored as a string — as it is when entered into a form text field — your scripts may have difficulty applying that value to a math operation. The JavaScript language provides two built-in functions to convert string representations of numbers to true numbers: `parseInt()` and `parseFloat()`.

There is a difference between integers and floating-point numbers in JavaScript. *Integers* are always whole numbers, with no decimal point or numbers to the right of a decimal. *Floating-point numbers*, on the other hand, have fractional values to the right of the decimal. By and large, JavaScript math operations don't differentiate between integers and floating-point numbers: A number is a number. The only time you need to be cognizant of the difference is when a method parameter requires an integer because it can't handle fractional values. For example, parameters to the `scroll()` method of a window require integer values of the number of pixels vertically and horizontally you want to scroll the window. That's because you can't scroll a window a fraction of a pixel on the screen.

To use either of these conversion functions, insert the string value you wish to convert as a parameter to the function. For example, look at the results of two different string values when passed through the `parseInt()` function:

```
parseInt("42")        // result = 42
parseInt("42.33")     // result = 42
```

Even though the second expression passes the string version of a floating-point number to the function, the value returned by the function is an integer. No rounding of the value occurs here (although other math functions can help with that if necessary). The decimal and everything to its right are simply stripped off.

The `parseFloat()` function returns an integer if it can; otherwise, it returns a floating-point number as follows:

```
parseFloat("42")       // result = 42
parseFloat("42.33")    // result = 42.33
```

Because these two conversion functions evaluate to their results, you simply insert the entire function wherever you need a string value converted to a number. Therefore, modifying an earlier example in which one of three values was a string, the complete expression can evaluate to the desired result:

```
3 + 3 + parseInt("3")  // result = 9
```

Converting numbers to strings

You'll have less need for converting a number to its string equivalent than the other way around. As you saw in the previous section, JavaScript gravitates toward strings when faced

with an expression containing mixed data types. Even so, it is good practice to perform data type conversions explicitly in your code to prevent any potential ambiguity. The simplest way to convert a number to a string is to take advantage of JavaScript's string tendencies in addition operations. By adding an empty string to a number, you convert the number to its string equivalent:

```
("" + 2500)           // result = "2500"
("" + 2500).length    // result = 4
```

In the second example, you can see the power of expression evaluation at work. The parentheses force the conversion of the number to a string. A *string* is a JavaScript object that has properties associated with it. One of those properties is the `length` property, which evaluates to the number of characters in the string. Therefore, the length of the string "2500" is 4. Note that the length value is a number, not a string.

Operators

You will use lots of *operators* in expressions. Earlier, you used the equal sign (=) as an assignment operator to assign a value to a variable. In the preceding examples with strings, you used the plus symbol (+) to join two strings. An operator generally performs some kind of calculation (operation) or comparison with two values (the value on each side of an operator is called an *operand*) to reach a third value. In this lesson, I briefly describe two categories of operators — arithmetic and comparison. Chapter 32 covers many more operators, but once you understand the basics here, the others are easier to grasp.

Arithmetic operators

It may seem odd to talk about text strings in the context of "arithmetic" operators, but you have already seen the special case of the plus (+) operator when one or more of the operands is a string. The plus operator instructs JavaScript to *concatenate* (pronounced kon-KAT-en-eight), or join, two strings together precisely where you place the operator. The string concatenation operator doesn't know about words and spaces, so the programmer must make sure that any two strings to be joined have the proper word spacing as part of the strings — even if that means adding a space:

```
firstName = "John";
lastName = "Doe";
fullName = firstName + " " + lastName;
```

JavaScript uses the same plus operator for arithmetic addition. When both operands are numbers, JavaScript knows to treat the expression as an arithmetic addition rather than a string concatenation. The standard math operators for addition, subtraction, multiplication, and division (+, `-`, *, `/`) are built into JavaScript.

Comparison operators

Another category of operator helps you compare values in scripts — whether two values are the same, for example. These kinds of comparisons return a value of the Boolean type — `true` or `false`. Table 6-2 lists the comparison operators. The operator that tests whether two items are equal consists of a pair of equal signs to distinguish it from the single equal sign assignment operator.

Table 6-2: JavaScript Comparison Operators

Symbol	*Description*
==	Equals
!=	Does not equal
>	Is greater than
>=	Is greater than or equal to
<	Is less than
<=	Is less than or equal to

Where comparison operators come into greatest play is in the construction of scripts that make decisions as they run. A cook does this in the kitchen all the time: If the sauce is too watery, add a bit of flour. You see comparison operators in action in the next chapter.

Exercises

1. Which of the following are valid variable declarations or initializations? Explain why each one is or is not valid. If an item is invalid, how do you fix it so that it is?

   a. `my_name = "Cindy";`

   b. `var how many = 25;`

   c. `var zipCode = document.getElementById("zip").value`

   d. `var 1address = document.("address1").value;`

2. Assume that the following statements operate rapidly in sequence, where each statement relies on the result of the one before it. For each of the statements in the sequence, write down how the `someVal` expression evaluates after the statement executes in JavaScript.

```
var someVal = 2;
someVal = someVal + 2;
someVal = someVal * 10;
someVal = someVal + "20";
someVal = "Robert";
```

3. Name the two JavaScript functions that convert strings to numbers. How do you give the function a string value to convert to a number?

4. Type and load the HTML page and script shown in Listing 6-1. Enter a three-digit number into the top two fields and click the Add button. Examine the code and explain what is wrong with the script. How do you fix the script so the proper sum is displayed in the output field?

Listing 6-1: **What's Wrong with This Page?**

```
<html>
<head>
<title>Sum Maker</title>
<script type="text/javascript">
<!--
function addIt() {
   var value1 = document.getElementById("inputA").value;
   var value2 = document.getElementById("inputB").value;
   document.getElementById("output").value = value1 + value2;
}
// -->
</script>
</head>

<body>
<form name="adder">
<input type="text" name="inputA" id="inputA" value="0" size="4" /><br />
<input type="text" name="inputB" id="inputB" value="0" size="4" />
<input type="button" value="Add" onclick="addIt()">
<p>_____</p>
<input type="text" name="output" id="output" size="6" />
</form>
</body>
</html>
```

5. What does the term *concatenate* mean in the context of JavaScript programming?

✦ ✦ ✦

CHAPTER 7

Programming Fundamentals, Part II

In This Chapter

How control structures make decisions

How to define functions

Where to initialize variables efficiently

What those darned curly braces are all about

The basics of data arrays

Your tour of programming fundamentals continues in this chapter with subjects that have more intriguing possibilities. For example, I show you how programs make decisions and why a program must sometimes repeat statements over and over. Before you're finished here, you will learn how to use one of the most powerful information holders in the JavaScript language: the array.

Decisions and Loops

Every waking hour of every day you make decisions of some kind — most of the time you probably don't even realize it. Don't think so? Well, look at the number of decisions you make at the grocery store, from the moment you enter the store to the moment you clear the checkout aisle.

No sooner do you enter the store than you are faced with a decision. Based on the number and size of items you intend to buy, do you pick up a hand-carried basket or attempt to extricate a shopping cart from the metallic conga line near the front of the store? That key decision may have impact later when you see a special offer on an item that is too heavy to put into the hand basket.

Next, you head for the food aisles. Before entering an aisle, you compare the range of goods stocked in that aisle against items on your shopping list. If an item you need is likely to be found in this aisle, you turn into the aisle and start looking for the item; otherwise, you skip the aisle and move to the head of the next aisle.

Later, you reach the produce section in search of a juicy tomato. Standing in front of the bin of tomatoes, you begin inspecting them one by one — picking one up, feeling its firmness, checking the color, looking for blemishes or signs of pests. You discard one, pick up another, and continue this process until one matches the criteria you set in your mind for an acceptable morsel. Your last stop in the store is the checkout aisle. "Paper or plastic?" the clerk asks. One more decision to make. What you choose impacts how you get the groceries from the car to the kitchen as well as your recycling habits.

In your trip to the store, you go through the same kinds of decisions and repetitions that your JavaScript programs also encounter. If you understand these frameworks in real life, you can now look into the JavaScript equivalents and the syntax required to make them work.

Control Structures

In the vernacular of programming, the kinds of statements that make decisions and loop around to repeat themselves are called *control structures*. A control structure directs the execution flow through a sequence of script statements based on simple decisions and other factors.

An important part of a control structure is the *condition*. Just as you may travel different routes to work depending on certain conditions (for example, nice weather, nighttime, attending a soccer game), so, too, does a program sometimes have to branch to an execution route if a certain condition exists. Each condition is an expression that evaluates to `true` or `false` — one of those Boolean data types mentioned in Chapter 6. The kinds of expressions commonly used for conditions are expressions that include a comparison operator. You do the same in real life: If it is true that the outdoor temperature is less than freezing, you put on a coat before going outside. In programming, however, the comparisons are strictly comparisons of values.

JavaScript provides several kinds of control structures for different programming situations. Three of the most common control structures you'll use are `if` constructions, `if...else` constructions, and `for` loops.

Chapter 31 covers in great detail other common control structures you should know. For this tutorial, however, you need to learn about the three common ones just mentioned.

if constructions

The simplest program decision is to follow a special branch or path of the program if a certain condition is true. Formal syntax for this construction follows. Items in italics get replaced in a real script with expressions and statements that fit the situation.

```
if (condition) {
   statement[s] if true

}
```

Don't worry about the curly braces yet. Instead, get a feel for the basic structure. The keyword, `if`, is a must. In the parentheses goes an expression that evaluates to a Boolean value. This is the condition being tested as the program runs past this point. If the condition evaluates to `true`, one or more statements inside the curly braces execute before continuing on with the next statement after the closing brace. If the condition evaluates to `false`, the statements inside the curly braces are ignored and processing continues with the next statement after the closing brace.

The following example assumes that a variable, `myAge`, has had its value set earlier in the script (exactly how is not important for this example). The condition expression compares the value `myAge` against a numeric value of 18.

```
if (myAge < 18) {
   alert("Sorry, you cannot vote.");
}
```

In this example, the data type of the value inside `myAge` must be a number so that the proper comparison (via the `<` comparison operator) does the right thing. For all instances of `myAge` less than 18, the nested statement inside the curly braces runs and displays the alert to the user. After the user closes the alert dialog box, the script continues with whatever statement follows the entire `if` construction.

if . . . else constructions

Not all program decisions are as simple as the one shown for the `if` construction. Rather than specifying one detour for a given condition, you might want the program to follow either of two branches depending on that condition. It is a fine, but important, distinction. In the plain `if` construction, no special processing is performed when the condition evaluates to `false`. But if processing must follow one of two special paths, you need the `if...else` construction. The formal syntax definition for an `if...else` construction is as follows:

```
if (condition) {
   statement[s] if true
} else {
   statement[s] if false
}
```

Everything you know about the condition for an `if` construction applies here. The only difference is the `else` keyword, which provides an alternate path for execution to follow if the condition evaluates to `false`.

As an example, the following `if...else` construction determines how many days are in February for a given year. To simplify the demo, the condition simply tests whether the year divides equally by 4. (True testing for this value includes special treatment of end-of-century dates, but I'm ignoring that for now.) The `%` operator symbol is called the *modulus operator* (covered in more detail in Chapter 32). The result of an operation with this operator yields the remainder of division of the two values. If the remainder is zero, the first value divides evenly by the second.

```
var febDays;
var theYear = 2004;
if (theYear % 4 == 0) {
   febDays = 29;
} else {
   febDays = 28;
}
```

The important point to see from this example is that by the end of the `if...else` construction, the `febDays` variable is set to either 28 or 29. No other value is possible. For years evenly divisible by 4, the first nested statement runs. For all other cases, the second statement runs. Processing then picks up with the next statement after the `if...else` construction.

About Repeat Loops

Repeat loops in real life generally mean the repetition of a series of steps until some condition is met, thus enabling you to break out of that loop. Such was the case earlier in this chapter when you looked through a bushel of tomatoes for the one that came closest to your ideal tomato. The same can be said for driving around the block in a crowded neighborhood until a parking space opens up.

A *repeat loop* lets a script cycle through a sequence of statements until some condition is met. For example, a JavaScript data validation routine might inspect every character that you enter into a form text field to make sure that each one is a number. Or if you have a collection of data stored in a list, the loop can check whether an entered value is in that list. Once that condition is met, the script can then break out of the loop and continue with the next statement after the loop construction.

The most common repeat loop construction used in JavaScript is called the `for` loop. It gets its name from the keyword that begins the construction. A `for` loop is a powerful device because you can set it up to keep track of the number of times the loop repeats itself. The formal syntax of the `for` loop is as follows:

```
for ([initial expression]; [condition]; [update expression]) {
   statement[s] inside loop
}
```

The square brackets mean that the item is optional. However, until you get to know the `for` loop better, I recommend designing your loops to utilize all three items inside the parentheses. The *initial expression* portion usually sets the starting value of a counter variable. The *condition* — the same kind of condition you saw for `if` constructions — defines the condition that forces the loop to stop going around and around. Finally, the *update expression* is a statement that executes each time all of the statements nested inside the construction complete running.

A common implementation initializes a counting variable, `i`, increments the value of `i` by one each time through the loop, and repeats the loop until the value of `i` exceeds some maximum value, as in the following:

```
for (var i = startValue; i <= maxValue; i++) {
   statement[s] inside loop
}
```

Placeholders `startValue` and `maxValue` represent any numeric values, including explicit numbers or variables holding numbers. In the update expression is an operator you have not seen yet. The ++ operator adds 1 to the value of `i` each time the update expression runs at the end of the loop. If `startValue` is 1, the value of `i` is 1 the first time through the loop, 2 the second time through, and so on. Therefore, if `maxValue` is 10, the loop repeats itself 10 times (in other words, as long as `i` is less than or equal to 10). Generally speaking, the statements inside the loop use the value of the counting variable in their execution. Later in this lesson, I show how the variable can play a key role in the statements inside a loop. At the same time, you will see how to break out of a loop prematurely and why you may need to do this in a script.

Functions

In Chapter 5, you saw a preview of the JavaScript function. A *function* is a definition of a set of deferred actions. Functions are invoked by event handlers or by statements elsewhere in the script. Whenever possible, good functions are designed for reuse in other documents. They can become building blocks you use over and over again.

If you have programmed before, you can see parallels between JavaScript functions and other languages' subroutines. But unlike some languages that distinguish between procedures (which carry out actions) and functions (which carry out actions and return values), only one

classification of routine exists for JavaScript. A function is capable of returning a value to the statement that invoked it, but this is not a requirement. However, when a function does return a value, the calling statement treats the function call like any expression — plugging in the returned value right where the function call is made. I will show some examples in a moment.

Formal syntax for a function is as follows:

```
function functionName ( [parameter1]...[,parameterN] ) {
   statement[s]
}
```

Names you assign to functions have the same restrictions as names you assign to HTML elements and variables. You should devise a name that succinctly describes what the function does. I tend to use multiword names with the interCap (internally capitalized) format that start with a verb because functions are action items, even if they do nothing more than get or set a value.

Another practice to keep in mind as you start to create functions is to keep the focus of each function as narrow as possible. It is possible to generate functions that are literally hundreds of lines long. Such functions are usually difficult to maintain and debug. Chances are that you can divide the long function into smaller, more tightly focused segments.

Function parameters

In Chapter 5, you saw how an event handler invokes a function by calling the function by name. Any call to a function, including one that comes from another JavaScript statement, works the same way: a set of parentheses follows the function name.

You also can define functions so they receive parameter values from the calling statement. Listing 7-1 shows a simple document that has a button whose `onclick` event handler calls a function while passing text data to the function. The text string in the event handler call is in a *nested string* — a set of single quotes inside the double quotes required for the entire event handler attribute.

Listing 7-1: **Calling a Function from an Event Handler**

```
<html>
<head>
<script type="text/javascript">
function showMsg(msg) {
   alert("The button sent: " + msg);
}
</script>
</head>
<body>
<form>
   <input type="button" value="Click Me"
   onclick="showMsg('The button has been clicked!')">
</form>
</body>
</html>
```

Parameters (also known as *arguments*) provide a mechanism for "handing off" a value from one statement to another by way of a function call. If no parameters occur in the function definition, both the function definition and call to the function have only empty sets of parentheses (as shown in Chapter 5, Listing 5-8).

When a function receives parameters, it assigns the incoming values to the variable names specified in the function definition's parentheses. Consider the following script segment:

```
function sayHiToFirst(a, b, c) {
   alert("Say hello, " + a);
}
sayHiToFirst("Gracie", "George", "Harry");
sayHiToFirst("Larry", "Moe", "Curly");
```

After the function is defined in the script, the next statement calls that very function, passing three strings as parameters. The function definition automatically assigns the strings to variables `a`, `b`, and `c`. Therefore, before the `alert()` statement inside the function ever runs, `a` evaluates to "Gracie," `b` evaluates to "George," and `c` evaluates to "Harry." In the `alert()` statement, only the `a` value is used and the alert reads

```
Say hello, Gracie
```

When the user closes the first alert, the next call to the function occurs. This time through, different values are passed to the function and assigned to `a`, `b`, and `c`. The alert dialog box reads

```
Say hello, Larry
```

Unlike other variables that you define in your script, function parameters do not use the `var` keyword to initialize them. They are automatically initialized whenever the function is called.

Variable scope

Speaking of variables, it's time to distinguish between variables that are defined outside and those defined inside of functions. Variables defined outside of functions are called *global variables*; those defined inside functions with the `var` keyword are called *local variables*.

A global variable has a slightly different connotation in JavaScript than it has in most other languages. For a JavaScript script, the "globe" of a global variable is the current document loaded in a browser window or frame. Therefore, when you initialize a variable as a global variable, it means that all script statements in the page (including those inside functions) have direct access to that variable value. Statements can retrieve and modify global variables from anywhere in the page. In programming terminology, this kind of variable is said to have *global scope* because everything on the page can "see" it.

It is important to remember that the instant a page unloads itself, all global variables defined in that page disappear from memory forever. If you need a value to persist from one page to another, you must use other techniques to store that value (for example, as a global variable in a framesetting document, as described in Chapter 16; or in a cookie, as described in Chapter 18). While the `var` keyword is usually optional for initializing global variables, I strongly recommend you use it for all variable initializations to guard against future changes to the JavaScript language.

In contrast to the global variable, a local variable is defined inside a function. You already saw how parameter variables are defined inside functions (without `var` keyword initializations). But you can also define other variables with the `var` keyword (absolutely required for local

variables or they become recognized as global variables). The scope of a local variable is only within the statements of the function. No other functions or statements outside of functions have access to a local variable.

Local scope allows for the reuse of variable names within a document. For most variables, I strongly discourage this practice because it leads to confusion and bugs that are difficult to track down. At the same time, it is convenient to reuse certain kinds of variable names, such as `for` loop counters. These are safe because they are always reinitialized with a starting value whenever a `for` loop starts. You cannot, however, nest one `for` loop inside another without specifying a different loop counting variable in the nested loop.

To demonstrate the structure and behavior of global and local variables — and show you why you shouldn't reuse most variable names inside a document — Listing 7-2 defines two global and two local variables. I intentionally use bad form by initializing a local variable that has the same name as a global variable.

Listing 7-2: **Global and Local Variable Scope Demonstration**

```
<html>
<head>
<script type="text/javascript">
var aBoy = "Charlie Brown";   // global
var hisDog = "Snoopy";        // global
function demo() {
   // using improper design to demonstrate a point
   var hisDog = "Gromit";    // local version of hisDog
   var output = hisDog + " does not belong to " + aBoy + ".<br>";
   document.write(output);
}
</script>
</head>
<body>
<script type="text/javascript">
demo();        // runs as document loads
document.write(hisDog + " belongs to " + aBoy + ".");
</script>
</body>
</html>
```

When the page loads, the script in the Head portion initializes the two global variables (`aBoy` and `hisDog`) and defines the `demo()` function in memory. In the Body, another script begins by invoking the function. Inside the function, a local variable is initialized with the same name as one of the global variables — `hisDog`. In JavaScript, such a local initialization overrides the global variable for all statements inside the function. (But note that if the `var` keyword is left off of the local initialization, the statement reassigns the value of the global version to "Gromit.")

Another local variable, `output`, is merely a repository for accumulating the text that is to be written to the screen. The accumulation begins by evaluating the local version of the `hisDog` variable. Then it concatenates some hard-wired text (note the extra spaces at the edges of the string segment). Next comes the evaluated value of the `aBoy` global variable — any global

not overridden by a local is available for use inside the function. The expression is accumulating HTML to be written to the page, so it ends with a period and a <br> tag. The final statement of the function writes the content to the page.

After the function completes its task, the next statement in the Body script writes another string to the page. Because this script statement is executing in global space (that is, not inside any function), it accesses only global variables — including those defined in another <script> tag set in the document. By the time the complete page finishes loading, it contains the following text lines:

```
Gromit does not belong to Charlie Brown.
Snoopy belongs to Charlie Brown.
```

About Curly Braces

Despite the fact that you probably rarely — if ever — use curly braces ({ }) in your writing, there is no mystery to their usage in JavaScript (and many other languages). Curly braces enclose blocks of statements that belong together. While they do assist humans who are reading scripts in knowing what's going on, curly braces also help the browser to know which statements belong together. You always must use curly braces in matched pairs.

You use curly braces most commonly in function definitions and control structures. In the function definition in Listing 7-2, curly braces enclose four statements that make up the function definition (including the comment line). The closing brace lets the browser know that whatever statement comes next is a statement outside of the function definition.

Physical placement of curly braces is not critical (nor is the indentation style you see in the code I provide). The following function definitions are treated identically by scriptable browsers:

```
function sayHiToFirst(a, b, c) {
   alert("Say hello, " + a);
}

function sayHiToFirst(a, b, c)
{
   alert("Say hello, " + a);
}

function sayHiToFirst(a, b, c) {alert("Say hello, " + a);}
```

Throughout this book, I use the style shown in the first example because I find that it makes lengthy and complex scripts easier to read — especially scripts that have many levels of nested control structures.

Arrays

The JavaScript array is one of the most useful data constructions you have available to you. You can visualize the structure of a basic array as if it were a single-column spreadsheet. Each row of the column holds a distinct piece of data, and each row is numbered. Numbers assigned to rows are in strict numerical sequence, starting with zero as the first row (programmers tend to start counting with zero). This row number is called an *index*. To access an

item in an array, you need to know the name of the array and the index for the row. Because index values start with zero, the total number of items of the array (as determined by the array's `length` property) is always one more than the highest index value of the array. More advanced array concepts enable you to create the equivalent of an array with multiple columns (described in Chapter 30). For this tutorial, I stay with the single-column basic array.

Data elements inside JavaScript arrays can be any data type, including objects. And, unlike a lot of other programming languages, different rows of the same JavaScript array can contain different data types.

Creating an array

An array is stored in a variable, so when you create an array you assign the new array object to the variable. (Yes, arrays are objects, but they belong to the core JavaScript language rather than the document object model.) A special keyword — `new` — preceding a call to the JavaScript function that generates arrays creates space in memory for the array. An optional parameter to the `Array()` function enables you to specify at the time of creation how many elements (rows) of data eventually will occupy the array. JavaScript is very forgiving about this because you can change the size of an array at any time. Therefore, if you omit a parameter when generating a new array, your script incurs no penalty.

To demonstrate the array creation process, I create an array that holds the names of the 50 states plus the District of Columbia (a total of 51). The first task is to create that array and assign it to a variable of any name that helps me remember what this collection of data is about:

```
var USStates = new Array(51);
```

At this point, the `USStates` array is sitting in memory like a 51-row table with no data in it. To fill the rows, I must assign data to each row. Addressing each row of an array requires a special way of indicating the index value of the row: square brackets after the name of the array. The first row of the `USStates` array is addressed as

```
USStates[0]
```

To assign the string name of the first state of the alphabet to that row, I use a simple assignment operator:

```
USStates[0] = "Alabama";
```

To fill in the rest of the rows, I include a statement for each row:

```
USStates[1] = "Alaska";
USStates[2] = "Arizona";
USStates[3] = "Arkansas";
...
USStates[50] = "Wyoming";
```

Therefore, if you want to include a table of information in a document from which a script can look up information without accessing the server, you include the data in the document in the form of an array creation sequence. When the statements run as the document loads, by the time the document finishes loading into the browser, the data collection array is built and ready to go. Despite what appears to be the potential for a lot of statements in a document for such a data collection, the amount of data that must download for typical array

collections is small enough not to severely impact page loading — even for dial-up users at 28.8 Kbps. In Chapter 30, you also see some syntax shortcuts for creating arrays that reduce source code character counts.

Accessing array data

The array index is the key to accessing an array element. The name of the array and an index in square brackets evaluates to the content of that array location. For example, after the `USStates` array is built, a script can display an alert with Alaska's name in it with the following statement:

```
alert("The largest state is " + USStates[1] + ".");
```

Just as you can retrieve data from an indexed array element, so can you change the element by reassigning a new value to any indexed element in the array.

Parallel arrays

Now I show you why the numeric index methodology works well in JavaScript. To help with the demonstration, I generate another array that is parallel with the `USStates` array. This new array is also 51 elements long, and it contains the year in which the state in the corresponding row of `USStates` entered the Union. That array construction looks like the following:

```
var stateEntered = new Array(51);
stateEntered [0] = 1819;
stateEntered [1] = 1959;
stateEntered [2] = 1912;
stateEntered [3] = 1836;
...
stateEntered [50] = 1890;
```

In the browser's memory, then, are two data tables that you can visualize as looking like the model in Figure 7-1. I can build more arrays that are parallel to these for items such as the postal abbreviation and capital city. The important point is that the zeroth element in each of these tables applies to Alabama, the first state in the `USStates` array.

USStates		stateEntered
"Alabama"	[0]	1819
"Alaska"	[1]	1959
"Arizona"	[2]	1912
"Arkansas"	[3]	1836
⋮	⋮	⋮
"Wyoming"	[50]	1890

Figure 7-1: Visualization of two related parallel data tables.

If a Web page included these data tables and a way for a user to look up the entry date for a given state, the page would need a way to look through all of the `USStates` entries to find the index value of the one that matches the user's entry. Then, that index value could be applied to the `stateEntered` array to find the matching year.

For this demo, the page includes a text entry field in which the user types the name of the state to look up. In a real application, this methodology is fraught with peril unless the script performs some error checking in case the user makes a mistake. But for now, I assume that the user always types a valid state name. (Don't ever make this assumption in your Web site's pages.) An event handler from either the text field or a clickable button calls a function that looks up the state name, fetches the corresponding entry year, and displays an alert message with the information. The function is as follows:

```
function getStateDate() {
   var selectedState = document.getElementById("entry").value;
   for ( var i = 0; i < USStates.length; i++) {
      if (USStates[i] == selectedState) {
         break;
      }
   }
   alert("That state entered the Union in " + stateEntered[i] + ".");
}
```

In the first statement of the function, I grab the value of the text box and assign the value to a variable, `selectedState`. This is mostly for convenience because I can use the shorter variable name later in the script. In fact, the usage of that value is inside a `for` loop, so the script is marginally more efficient because the browser doesn't have to evaluate that long reference to the text field each time through the loop.

The key to this function is in the `for` loop. Here is where I combine the natural behavior of incrementing a loop counter with the index values assigned to the two arrays. Specifications for the loop indicate that the counter variable, `i`, is initialized with a value of zero. The loop is directed to continue as long as the value of `i` is less than the length of the `USStates` array. Remember that the length of an array is always one more than the index value of the last item. Therefore, the last time the loop runs is when `i` is 50, which is both less than the length of 51 and equal to the index value of the last element. Each time after the loop runs, the counter increments by one.

Nested inside the `for` loop is an `if` construction. The condition tests the value of an element of the array against the value typed in by the user. Each time through the loop, the condition tests a different row of the array starting with row zero. In other words, this `if` construction can be performed dozens of times before a match is found, but each time the value of `i` is one larger than the previous try.

The equality comparison operator (`==`) is strict when it comes to comparing string values. Such comparisons respect the case of each letter. In our example, the user must type the state name exactly as it is stored in the `USStates` array for the match to be found. In Chapter 10, you learn about some helper methods that eliminate case and sensitivity in string comparisons.

When a match is found, the statement nested inside the `if` construction runs. The `break` statement is designed to help control structures bail out if the program needs it. For this application, it is imperative that the `for` loop stop running when a match for the state name

is found. When the `for` loop breaks, the value of the `i` counter is fixed at the row of the `USStates` array containing the entered state. I need that index value to find the corresponding entry in the other array. Even though the counting variable, `i`, is initialized in the `for` loop, it is still "alive" and in the scope of the function for all statements after the initialization. That's why I can use it to extract the value of the row of the `stateEntered` array in the final statement that displays the results in an alert message.

This application of a `for` loop and array indexes is a common one in JavaScript. Study the code carefully and be sure you understand how it works. This way of cycling through arrays plays a role not only in the kinds of arrays you create in your code, but also with the arrays that browsers generate for the document object model.

Document objects in arrays

If you look at the `document` object portions of the Quick Reference in Appendix A, you can see that the properties of some objects are listed with square brackets after them. These are, indeed, the same kind of square brackets you just saw for array indexes. That's because when a document loads, the browser creates arrays of like objects in the document. For example, if your page includes two `<form>` tag sets, then two forms appear in the document. The browser maintains an array of form objects for that document. References to those forms are

```
document.forms[0]
document.forms[1]
```

Index values for document objects are assigned according to the loading order of the objects. In the case of `form` objects, the order is dictated by the order of the `<form>` tags in the document. This indexed array syntax is another way to reference forms in an object reference. You can still use a form's identifier if you prefer — and I heartily recommend using object names wherever possible because even if you change the physical order of the objects in your HTML, references that use names still work without modification. But if your page contains only one form, you can use the reference types interchangeably, as in the following examples of equivalent references to the length property of a form's `elements` array (the `elements` array contains all the form controls in the form):

```
document.getElementById("entryForm").elements.length
document.forms[0].elements.length
```

In examples throughout this book, you can see that I often use the array type of reference to simple forms in simple documents. But in my production pages, I almost always use named references.

Exercises

1. With your newly acquired knowledge of functions, event handlers, and control structures, use the script fragments from this chapter to complete the page that has the lookup table for all of the states and the years they entered into the Union. If you do not have a reference book for the dates, use different year numbers starting with 1800 for each entry. In the page, create a text entry field for the state and a button that triggers the lookup in the arrays.

2. Examine the following function definition. Can you spot any problems with the definition? If so, how can you fix the problems?

```
function format(ohmage) {
   var result;
   if ohmage >= 1e6 {
      ohmage = ohmage / 1e6;
      result = ohmage + " Mohms";
   } else {
      if (ohmage >= 1e3)
         ohmage = ohmage / 1e3;
         result = ohmage + " Kohms";
      else
         result = ohmage + " ohms";
}
alert(result);
```

3. Devise your own syntax for the scenario of looking for a ripe tomato at the grocery store, and write a `for` loop using that object and property syntax.

4. Modify Listing 7-2 so it does not reuse the `hisDog` variable inside the function.

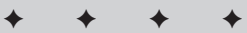
5. Given the following table of data about several planets of our solar system, create a Web page that enables users to enter a planet name and, at the click of a button, have the distance and diameter appear either in an alert box or (as extra credit) in separate fields of the page.

Planet	*Distance from the Sun*	*Diameter*
Mercury	36 million miles	3,100 miles
Venus	67 million miles	7,700 miles
Earth	93 million miles	7,920 miles
Mars	141 million miles	4,200 miles

✦ ✦ ✦

CHAPTER 8

Window and Document Objects



In This Chapter

What the `window` object does

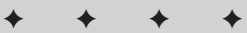
How to access key `window` object properties and methods

How to trigger script actions after a document loads

The purposes of the `location` and `history` objects

How the `document` object is created

How to access key `document` object properties and methods



Now that you have exposure to programming fundamentals, it is easier to demonstrate how to script objects in documents. Starting with this lesson, the tutorial turns back to the document object model, diving more deeply into each of the objects you will place in many of your documents.

Top-Level Objects

As a refresher, study the hierarchy of top-level objects in Figure 8-1. This chapter focuses on objects of this level that you'll frequently encounter in your scripting: `window`, `location`, `navigator`, and `document`. The goal is not only to equip you with the basics so you can script simple tasks, but also to prepare you for in-depth examinations of each object and its properties, methods, and event handlers in Part III of this book. I introduce only the basic properties, methods, and event handlers for objects in this tutorial—you can find far more in Part III. Examples in that part of the book assume you know the programming fundamentals covered in previous chapters.

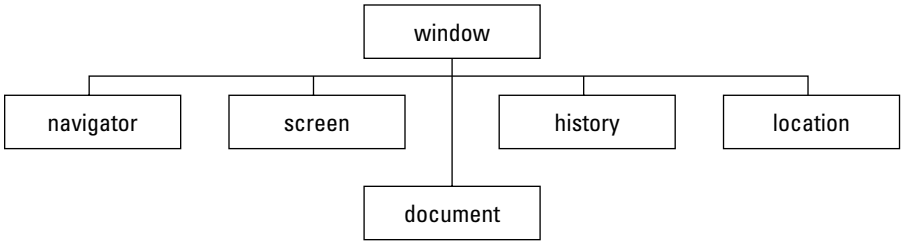


Figure 8-1: The top-level browser object model for all scriptable browsers.

The window Object

At the very top of the object hierarchy is the `window` object. This object gains that exalted spot in the object food chain because it is the master container for all content you view in the Web browser. As long as a browser window is open — even if no document is loaded in the window — the `window` object is defined in the current model in memory.

In addition to the content part of the window where documents go, a window's sphere of influence includes the dimensions of the window and all of the "stuff" that surrounds the content area. The area where scrollbars, toolbars, the status bar, and (non-Macintosh) menu bar live is known as a window's *chrome*. Not every browser has full scripted control over the chrome of the main browser window, but you can easily script the creation of additional windows sized the way you want and that have only the chrome elements you wish to display in the subwindow.

Although the discussion about frames comes in Chapter 11, I can safely say now that each frame is also considered a `window` object. If you think about it, that makes sense because each frame can hold a different document. When a script runs in one of those documents, it regards the frame that holds the document as the `window` object in its view of the object hierarchy.

As you learn in this chapter, the `window` object is a convenient place for the document object model to attach methods that display modal dialog boxes and adjust the text that displays in the status bar at the bottom of the browser window. A `window` object method enables you to create a separate window that appears on the screen. When you look at all of the properties, methods, and event handlers defined for the `window` object (see Chapter 16), it should be clear why they are attached to window objects — visualize their scope and the scope of a browser window.

Accessing window properties and methods

You can word script references to properties and methods of the `window` object in several ways, depending more on whim and style than on specific syntactical requirements. The most logical and common way to compose such references includes the `window` object in the reference:

```
window.propertyName
window.methodName([parameters])
```

A window object also has a synonym when the script doing the referencing points to the window that houses the document. The synonym is self. Reference syntax then becomes

```
self.propertyName
self.methodName([parameters])
```

You can use these initial reference object names interchangeably, but I tend to reserve the use of self for more complex scripts that involve multiple frames and windows. The self moniker more clearly denotes the current window holding the script's document. It makes the script more readable — by me and by others.

Back in Chapter 4, I indicated that because the window object is always "there" when a script runs, you could omit it from references to any objects inside that window. Therefore, the following syntax models assume properties and methods of the current window:

```
propertyName
methodName([parameters])
```

In fact, as you will see in a few moments, some methods may be more understandable if you omit the window object reference. The methods run just fine either way.

Creating a window

A script does not create the main browser window. A user does that by virtue of launching the browser or by opening a URL or file from the browser's menus (if the window is not already open). But a script can generate any number of subwindows once the main window is open (and that window contains a document whose script needs to open subwindows).

The method that generates a new window is window.open(). This method contains up to three parameters that define window characteristics, such as the URL of the document to load, its name for target attribute reference purposes in HTML tags, and physical appearance (size and chrome contingent). I don't go into the details of the parameters here (they're covered in great depth in Chapter 16), but I do want to expose you to an important concept involved with the window.open() method.

Consider the following statement that opens a new window to a specific size and with an HTML document from the same server directory that holds the current page:

```
var subWindow = window.open("define.html","def","height=200,width=300");
```

The important thing to note about this statement is that it is an assignment statement. Something gets assigned to that variable subWindow. What is it? It turns out that when the window.open() method runs, it not only opens up that new window according to specifications set as parameters, but it also evaluates to a reference to that new window. In programming parlance, the method is said to *return a value* — in this case, a genuine object reference. The value returned by the method is assigned to the variable.

Your script can now use that variable as a valid reference to the second window. If you need to access one of its properties or methods, you must use that reference as part of the complete reference. For example, to close the subwindow from a script in the main window, use this reference to the close() method for that subwindow:

```
subWindow.close();
```

If you issue `window.close()`, `self.close()`, or just `close()` in the main window's script, the method closes the main window (after confirming with the user) and not the subwindow. To address another window, then, you must include a reference to that window as part of the complete reference. This has an impact on your code because you probably want the variable holding the reference to the subwindow to be valid as long as the main document is loaded into the browser. For that to happen, the variable has to be initialized as a global variable, rather than inside a function (although you can set its value inside a function). That way, one function can open the window while another function closes it.

Listing 8-1 is a page that has a button for opening a blank, new window and closing that window from the main window. To view this demonstration, shrink your main browser window to less than full screen. Then when the new window is generated, reposition the windows so you can see the smaller, new window when the main window is in front. (If you "lose" a window behind another, use the browser's Window menu to choose the hidden window.) The key point of Listing 8-1 is that the `newWindow` variable is defined as a global variable so that both the `makeNewWindow()` and `closeNewWindow()` functions have access to it. When a variable is declared with no value assignment, its initial value is `null`. A `null` value is interpreted to be the same as `false` in a condition, while the presence of any non-zero value is the same as `true` in a condition. Therefore, in the `closeNewWindow()` function, the condition tests whether the window has been created before issuing the subwindow's `close()` method. Then, to clean up, the function sets the `newWindow` variable to `null` so that another click of the Close button doesn't try to close a nonexistent window.

Listing 8-1: **References to Window Objects**

```
<html>
<head>
<title>Window Opener and Closer</title>
<script type="text/javascript">
var newWindow
function makeNewWindow() {
   newWindow = window.open("","","height=300,width=300");
}
function closeNewWindow() {
   if (newWindow) {
      newWindow.close();
      newWindow = null;
   }
}
</script>
</head>

<body>
<form>
<input type="button" value="Create New Window" onclick="makeNewWindow()">
<input type="button" value="Close New Window" onclick="closeNewWindow()">
</form>
</body>
</html>
```

Window Properties and Methods

The one property and three methods for the `window` object described in this section have an immediate impact on user interaction. They work with all scriptable browsers. You can find extensive code examples in Part III for each property and method. You can also experiment with the one-statement script examples by entering them in the top text box of The Evaluator Jr. (from Chapter 6).

window.status property

The status bar at the bottom of the browser window normally displays the URL of a link when you roll the mouse pointer atop it. Other messages also appear in that space during document loading, Java applet initialization, and the like. However, you can use JavaScript to display your own messages in the status bar at times that may be beneficial to your users. For example, rather than display the URL of a link, you can display a friendlier, plain-language description of the page at the other end of the link (or a combination of both to accommodate both newbies and geeks).

You can assign the `window.status` property some other text at any time. To change the status bar text of a link as the cursor hovers atop the link, you trigger the action with an `onmouseover` event handler of a link object.

Due to the simplicity of setting the `window.status` property, it is most common for the script statements to run as inline scripts in the event handler definition. This is handy for short scripts because you don't have to specify a separate function or add `<script>` tags to your page. You simply add the script statements to the `<a>` tag:

```
<a href="http://www.microsoft.com" onmouseover=
"window.status='Visit the Microsoft Home page (microsoft.com)'"> Microsoft</a>
```

Look closely at the script statement assigned to the `onmouseover` event handler:

```
window.status='Visit the Microsoft Home page (microsoft.com)'
```

The entire statement is surrounded by double quotes (" . . . "). To nest the string being assigned to the `window.status` property inside the double-quoted script, you surround the string with single quotes (' . . . '). You get a big payoff for a little bit of script when you set the status bar. The downside is that scripting this property is how those awful status bar scrolling banners are created. Yech!

window.alert() method

I have already used the `alert()` method many times so far in this tutorial. This window method generates a dialog box that displays whatever text you pass as a parameter (see Figure 8-2). A single OK button (whose label you cannot change) enables the user to dismiss the alert.

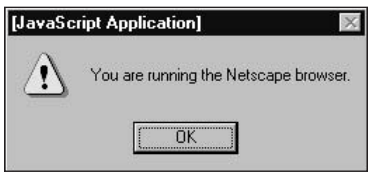


Figure 8-2: A JavaScript alert dialog box (old style).

The appearance of this and two other JavaScript dialog boxes (described next) has changed since the first scriptable browsers. In older browser versions (as shown in Figure 8-2), the browser inserted words clearly indicating that the dialog box was a "JavaScript Alert." Different browsers display different title bars whose content cannot be altered by script. You can change only the other message content.

All three dialog box methods are good cases for using a `window` object's methods without the reference to the window. Even though the `alert()` method is technically a `window` object method, no special relationship exists between the dialog box and the window that generates it. In production scripts, I usually use the shortcut reference:

```
alert("This is a JavaScript alert dialog.");
```

window.confirm() method

The second style of dialog box presents two buttons (Cancel and OK in most versions on most platforms) and is called a confirm dialog box (see Figure 8-3). More importantly, this is one of those methods that returns a value: `true` if the user clicks OK, `false` if the user clicks Cancel. You can use this dialog box and its returned value as a way to have a user make a decision about how a script progresses.

Figure 8-3: A JavaScript confirm dialog box (IE6/WinXP style).

Because the method always returns a Boolean value, you can use the evaluated value of the entire method as a condition statement in an `if` or `if...else` construction. For example, in the following code fragment, the user is asked about starting the application over. Doing so causes the default page of the site to load into the browser.

```
if (confirm("Are you sure you want to start over?")) {
    location.href = "index.html";
}
```

window.prompt() method

The final dialog box of the `window` object, the prompt dialog box (see Figure 8-4), displays a message that you set and provides a text field for the user to enter a response. Two buttons, Cancel and OK, enable the user to dismiss the dialog box with two opposite expectations: canceling the entire operation or accepting the input typed into the dialog box.

Figure 8-4: A JavaScript prompt dialog box (IE6/WinXP style).

The window.prompt() method has two parameters. The first is the message that acts as a prompt to the user. You can suggest a default answer in the text field by including a string as the second parameter. If you don't want any default answer to appear, include an empty string (two double quotes without any space between them).

This method returns one value when the user clicks either button. A click of the Cancel button returns a value of null, regardless of what the user types into the field. A click of the OK button returns a string value of the typed entry. Your scripts can use this information in conditions for if and if...else constructions. A value of null is treated as false in a condition. It turns out that an empty string is also treated as false. Therefore, a condition can easily test for the presence of real characters typed into the field to simplify a condition test, as shown in the following fragment:

```
var answer = prompt("What is your name?","");
if (answer) {
   alert("Hello, " + answer + "!");
}
```

The only time the alert() method is called is when the user enters something into the prompt dialog box and clicks the OK button.

onload event handler

The window object reacts to several system and user events, but the one you will probably use most often is the event that fires as soon as everything in a page finishes loading. This event waits for images, Java applets, and data files for plug-ins to download fully to the browser. It can be dangerous to script access to elements of a document object while the page loads because if the object has not loaded yet (perhaps due to a slow network connection or server), a script error results. The advantage of using the onload event to invoke functions is that you are assured that all document objects are in the browser's document object model. Window event handlers are placed inside the <body> tag. Even though you will come to associate the <body> tag's attributes with the document object's properties, it is the window object's event handlers that go inside the tag.

The location Object

Sometimes an object in the hierarchy represents something that doesn't seem to have the kind of physical presence that a window or a button does. That's the case with the location object. This object represents the URL loaded into the window. This differs from the document object (discussed later in this lesson) because the document is the real content; the location is simply the URL.

Unless you are truly Web-savvy, you may not realize a URL consists of many components that define the address and method of data transfer for a file. Pieces of a URL include the protocol (such as http:) and the hostname (such as www.example.com). You can access all of these items as properties of the location object. For the most part, though, your scripts will be interested in only one property: the href property, which defines the complete URL.

Setting the location.href property is the primary way your scripts navigate to other pages:

```
location.href = "http://www.dannyg.com";
```

You can generally navigate to a page in your own Web site by specifying a relative URL (that is, relative to the currently loaded page) rather than the complete URL with protocol and host information. For pages outside of the domain of the current page, you need to specify the complete URL.

If the page to be loaded is in another window or frame, the window reference must be part of the statement. For example, if your script opens a new window and assigns its reference to a variable named `newWindow`, the statement that loads a page into the subwindow is

```
newWindow.location.href = "http://www.dannyg.com";
```

The navigator Object

Despite a name reminiscent of the Netscape Navigator branded browser, the `navigator` object is implemented in all scriptable browsers. All browsers also implement a handful of properties that reveal the same kind of information that browsers send to servers with each page request. Thus, the `navigator.userAgent` property returns a string text with numerous details about the browser and operating system. For example, a script running in Internet Explorer 6 in Windows XP receives the following value for the `navigator.userAgent` property:

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; Q312461)
```

The same script running in Mozilla 1.4 on a Macintosh reveals the following `userAgent` details:

```
Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/20030624
```

You have already used two other navigator properties (`navigator.appVersion` and `navigator.appName`) in your first script of Chapter 3. See Chapter 38 on the CD-ROM for more details about this object and the meaning of the values returned by its properties. It was once used extensively to branch script execution according to various browser versions. Chapter 14 describes more modern ways to accomplish browser version detection.

The document Object

The `document` object holds the real content of the page. Properties and methods of the `document` object generally affect the look and content of the document that occupies the window. All W3C DOM-compatible browsers (and IE4) allow script access to the text con-tents of a page once the document has loaded. However, as you saw in your first script of Chapter 3, the `document.write()` method lets a script dynamically create content as the page loads on any browser. Many `document` object properties are arrays of other objects in the document, which provide additional ways to reference these objects (over and above the `document.getElementById()` method).

Accessing a `document` object's properties and methods is straightforward, as shown in the following syntax examples:

```
[window.]document.propertyName
[window.]document.methodName([parameters])
```

The `window` reference is optional when the script is accessing the `document` object that contains the script. If you want a preview of the long list of `document` object properties of the browser you're using, enter `document` into the bottom text box of The Evaluator Jr. and press Enter/Return. The object's properties and current values appear in the Results box. Following are some of the most commonly used properties and methods of the `document` object.

document.forms[] property

It is convenient that the document object contains a property — `document.forms` — whose value is an array of all form element objects in the document. As you recall from the discussion of arrays in Chapter 7, an index number inside an array's square brackets points to one of the elements in the array. To find out how many `form` objects are in the current document, use

```
document.forms.length
```

To access the first form in a document, for example, the reference is

```
document.forms[0]
```

As a further convenience, all scriptable browsers let you reference a form more directly by its name (that is, the identifier assigned to the `name` attribute of the `<form>` tag) in one of two ways. The first way is via array syntax, applying the form's name as a string index value of the array:

```
document.forms["formName"]
```

You will see in future chapters that scripts sometimes have only the string name of the form to work with. To derive a valid reference to the `form` object indicated by that name, use this string index form with the array.

The second, even shorter way to reference a `form` object by name is to append the name as a property of the `document` object, as in

```
document.formName
```

Either methodology reaches the same object. You will see many instances of the shortcut approach in form-related example scripts throughout this book (including in the next chapter when working with form controls). Although this syntax dates back to the earliest scriptable browsers, it is still valid in the most modern versions.

document.images[] property

Just as a document keeps track of forms in an array property, so does the `document` object maintain a collection (array) of images inserted into the document by way of `<img>` tags. Images referenced through the `document.images` array may be reached either by numeric or string index of the `img` element's name. Just like forms, the `name` attribute value is the identifier you use for a string index.

The presence of the `document.images` property indicates that the browser supports image swapping. You can therefore use the existence of the property as a controller to make sure the browser supports images as objects before attempting to perform any script action on an image. To do so, surround statements that deal with images with an `if` construction that verifies the property's existence, as follows:

```
if (document.images) {
    // statements dealing with img objects
}
```

Older browsers skip over the nested statements, preventing them from displaying error messages to their users.

document.write() method

The `document.write()` method operates in both immediate scripts to create content in a page as it loads and in deferred scripts that create new content in the same or different window. The method requires one string parameter, which is the HTML content to write to the window or frame. Such string parameters can be variables or any other expressions that evaluate to a string. Very often, the written content includes HTML tags.

Bear in mind that after a page loads, the browser's *output stream* automatically closes. After that, any `document.write()` method issued to the current page opens a new stream that immediately erases the current page (along with any variables or other values in the original document). Therefore, if you wish to replace the current page with script-generated HTML, you need to accumulate that HTML in a variable and perform the writing with just one `document.write()` method. You don't have to explicitly clear a document and open a new data stream; one `document.write()` call does it all.

One last piece of housekeeping advice about the `document.write()` method involves its companion method, `document.close()`. Your script must close the output stream when it finishes writing its content to the window (either the same window or another). After the last `document.write()` method in a deferred script, be sure to include a `document.close()` method. Failure to do this may cause images and forms not to appear. Also, any `document.write()` method invoked later will only append to the page, rather than clear the existing content to write anew. To demonstrate the `document.write()` method, I show two versions of the same application. One writes to the same document that contains the script; the other writes to a separate window. Type in each document in a new text editor document, save it with an `.html` filename extension, and open it in your browser.

Listing 8-2 creates a button that assembles new HTML content for a document, including HTML tags for a new document title and color attribute for the `<BODY>` tag. An operator in the listing that may be unfamiliar to you is `+=`. It appends a string on its right side to whatever string is stored in the variable on its left side. This operator is a convenient way to accumulate a long string across several separate statements. With the content gathered in the `newContent` variable, one `document.write()` statement blasts the entire new content to the same document, obliterating all vestiges of the content of Listing 8-2. The `document.close()` statement, however, is required to close the output stream properly. When you load this document and click the button, notice that the document title in the browser's title bar changes accordingly. As you click back to the original and try the button again, notice that the dynamically written second page loads much faster than even a reload of the original document.

Listing 8-2: **Using document.write() on the Current Window**

```
<html>
<head>
<title>Writing to Same Doc</title>
<script type="text/javascript">
function reWrite() {
   // assemble content for new window
   var newContent = "<html><head><title>A New Doc</title></head>";
   newContent += "<body bgcolor='aqua'><h1>This document is brand new.</h1>";
   newContent += "Click the Back button to see original document.";
   newContent += "</body></html>";
   // write HTML to new window document
   document.write(newContent);
   document.close(); // close layout stream
}
```

```
</script>
</head>
<body>
<form>
<input type="button" value="Replace Content" onclick="reWrite()">
</form>
</body>
</html>
```

In Listing 8-3, the situation is a bit more complex because the script generates a subwindow to which is written an entirely script-generated document. To keep the reference to the new window alive across both functions, the `newWindow` variable is declared as a global variable. As soon as the page loads, the `onload` event handler invokes the `makeNewWindow()` function. This function generates a blank subwindow. I added a property to the third parameter of the `window.open()` method that instructs the status bar of the subwindow to appear.

A button in the page invokes the `subWrite()` method. The first task it performs is to check the `closed` property of the subwindow. This property (which exists only in newer browser versions) returns `true` if the referenced window is closed. If that's the case (if the user manually closed the window), the function invokes the `makeNewWindow()` function again to reopen that window.

With the window open, new content is assembled as a string variable. As with Listing 8-2, the content is written in one blast (although that isn't necessary for a separate window), followed by a `close()` method. But notice an important difference: both the `write()` and `close()` methods explicitly specify the subwindow.

Listing 8-3: **Using document.write() on Another Window**

```
<html>
<head>
<title>Writing to Subwindow</title>
<script type="text/javascript">
var newWindow;
function makeNewWindow() {
   newWindow = window.open("","","status,height=200,width=300");
}

function subWrite() {
   // make new window if someone has closed it
   if (newWindow.closed) {
      makeNewWindow();
   }
   // bring subwindow to front
   newWindow.focus();
   // assemble content for new window
   var newContent = "<html><head><title>A New Doc</title></head>";
   newContent += "<body bgcolor='coral'><h1>This document is brand new.</h1>";
   newContent += "</body></html>";
   // write HTML to new window document
   newWindow.document.write(newContent);
   newWindow.document.close(); // close layout stream
```

Continued

Listing 8-3 *(continued)*

```
</script>
</head>
<body onload="makeNewWindow()">
<form>
<input type="button" value="Write to Subwindow" onclick="subWrite()">
</form>
</body>
</html>
```

document.createElement() and document.createTextNode() methods

The `document.write()` method works on a piece of a Web page only while the page is loading into the browser the first time. Any subsequent invocation of the method erases the page and writes a new page. But if you want to add to or modify a page that has already loaded, you need to call upon the Dynamic HTML capabilities of W3C DOM-compatible browsers. Your goal will be to add to, delete from, or replace sections of the node hierarchy of the document. Most element objects have methods to perform those actions (see more in-depth discussion in Chapter 14). But if you need to add content, you'll have to create new element or text nodes. The `document` object has the methods to do that.

The `document.createElement()` method lets you create in the browser's memory a brand new element object. To specify the precise element you wish to create, pass the tag name of the element as a string parameter of the method:

```
var newElem = document.createElement("p");
```

You may also wish to add some attribute values to the element, which you may do by assigning values to the newly created object's properties, even before the element becomes part of the document.

As you saw in Chapter 4's object hierarchy illustrations, an element object frequently needs text content between its start and end tags. The W3C DOM way to create that text is to generate a brand new text node via the `document.createTextNode()` method, and populate the node with the desired text. For example:

```
var newText = document.createTextNode("Greetings to all.");
```

The act of creating an element or text node does not, by itself, influence the document node tree. You must invoke one of the various insertion or replacement methods (see Chapter 14) to place the new text node into its element and place the element into the document. You learn how to do this in the last tutorial chapter (Chapter 12).

document.getElementById() method

You met the document.getElementById() method in Chapter 4 when learning about the syntax for referencing element objects. This W3C DOM method is one you will use a lot. Get to know its finger-twisting name well. Be sure to honor the upper- and lowercase spelling of this all-important method.

The sole parameter of this method is a quoted string containing the ID of the element you wish to reference. The method returns a value, which you typically preserve in a variable for use by subsequent script statements:

```
var oneTable = document.getElementById("salesResults");
```

After the assignment statement, the variable represents the element object, allowing you to get and set its properties or invoke whatever methods belong to that type of object.

The next logical step past the document level in the object hierarchy is the form. That's where you will spend the next lesson.

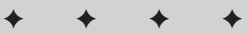
Exercises

1. Which of the following references are valid and which are not? Explain what is wrong with the invalid references.
   a. `window.document.form[0]`
   b. `self.entryForm.submit()`
   c. `document.forms[2].name`
   d. `document.getElementByID("firstParagraph")`
   e. `newWindow.document.write("Howdy")`
2. Write the JavaScript statement that displays a message in the status bar welcoming visitors to your Web page.
3. Write the JavaScript statement that executes while the page loads to display the same message from question 2 to the document as an `<h1>`-level headline on the page.
4. Create a page that prompts the user for his or her name as the page loads (via a dialog box) and then welcomes the user by name in the body of the page.
5. Create a page with any content you like, but one that automatically displays a dialog box after the page loads to show the user the URL of the current page.

✦ ✦ ✦

CHAPTER 9

Forms and Form Elements



In This Chapter

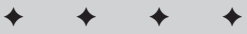
What the `form` object represents

How to access key `form` object properties and methods

How text, button, and `select` objects work

How to submit forms from a script

How to pass information from form elements to functions



Most interactivity between a Web page and the user takes place inside a form. That's where a lot of the interactive HTML stuff lives for every browser: text fields, buttons, checkboxes, option lists, and so on.

As described in earlier chapters, you may use the modern DOM `document.getElementById()` method to reference any element, including forms and form controls. But this chapter focuses on an older, yet equally valid way of referencing forms and controls. It's important to be familiar with this widely used syntax so that you can understand existing JavaScript source code written according to the original (and fully backward-compatible) form syntax — the so-called DOM Level 0 syntax.

The form Object

Using the original DOM Level 0 syntax, you can reference a `form` object either by its position in the array of forms contained by a document or by name (if you assign an identifier to the `name` attribute inside the `<form>` tag). If only one form appears in the document, it is still a member of an array (a one-element array) and is referenced as follows:

```
document.forms[0]
```

Or use the string of the element's name as the array index:

```
document.forms[formName]
```

Notice that the array reference uses the plural version of the word, followed by a set of square brackets containing the index number (zero is always first) or name of the element. Alternatively, you can use the form's name (not as a quoted string) as if it were a property of the `document` object:

```
document.formName
```

Form as object and container

Unlike the modern DOM's ID reference model — which lets a script dive anywhere into a document to grab an element object reference — DOM Level 0 form syntax imposes a hierarchical approach. It treats the `form` object as a container whose contents consist of the form control element objects (`input`, `select`, and `textarea` elements). Figure 9-1 shows the structure of this hierarchy and its place relative to the `document` object. You'll see the effect this structure has on the way you reference form control elements in a moment. This structure echoes perfectly the HTML tag organization within the `<form>` and `</form>` tag "bookends."

In addition to a large collection of properties and methods it has in common with all HTML element objects, the `form` object features a number of items that are unique to this object. Almost all of these unique properties are scripted representations of the `form` element's attributes. Scriptable browsers allow scripts to change these properties under script control, which gives your scripts potentially significant power to direct the behavior of a form submission in response to user selections on the page.

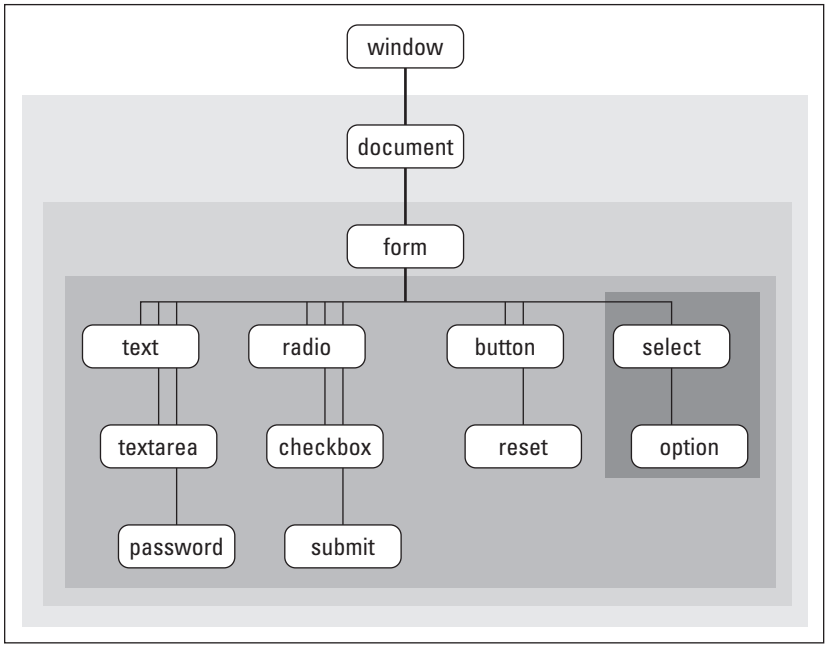


Figure 9-1: DOM Level 0 hierarchy for forms and controls.

Accessing form properties

Forms are created entirely from standard HTML tags in the page. You can set attributes for `name`, `target`, `action`, `method`, and `enctype`. Each of these is a property of a `form` object, accessed by all lowercase versions of those words, as in

```
document.forms[0].action
document.formName.action
```

To change any of these properties, simply assign new values to them:

```
document.forms[0].action = "http://www.giantco.com/cgi/login.pl";
```

form.elements[] property

In addition to keeping track of each type of element inside a form, the browser also maintains a list of all control elements within a form. This list is another array, with items listed according to the order in which their HTML tags appear in the source code. It is generally more efficient to create references to elements directly, using their names. However, sometimes a script needs to look through all of the elements in a form. This is especially true if the content of a form changes with each loading of the page because the number of text fields changes based on the user's browser type. (For example, a script on the page uses `document.write()` to add an extra text box for information required only from Windows users.)

The following code fragment shows the `form.elements[]` property at work in a `for` repeat loop that looks at every element in a form to set the contents of text fields to an empty string. The script cannot simply barge through the form and set every element's content to an empty string because some elements may be types (for example, a button) whose `value` properties have different purposes.

```
var form = window.document.forms[0];
for (var i = 0; i < form.elements.length; i++) {
   if (form.elements[i].type == "text") {
      form.elements[i].value = "";
   }
}
```

In the first statement, I create a variable — `form` — that holds a reference to the first form of the document. I do this so that when I make many references to form elements later in the script, the typical length of each reference is much shorter (and marginally faster). I can use the `form` variable as a shortcut to building references to items more deeply nested in the form.

Next, I start looping through the items in the `elements` array for the form. Each form element has a `type` property, which reveals what kind of form control it is: text, button, radio, checkbox, and so on. I'm interested in finding elements whose type is `text`. For each of those, I set the `value` property to an empty string.

I return to forms later in this chapter to show you how to submit a form without a Submit button and how client-side form validation works.

Form Controls as Objects

Three kinds of HTML elements nested inside a <form> tag become scriptable objects in all browser document object models. Most of the objects owe their existence to the <input> tag in the page's source code. Only the value assigned to the type attribute of an <input> tag determines whether the element is a text box, password entry field, hidden field, button, checkbox, or radio button. The other two kinds of form controls, textarea and select, have their own tags.

To reference a particular form control as an object in DOM Level 0 syntax, you build the reference as a hierarchy starting with the document, through the form, and then to the control. You've already seen how many ways you can reference merely the form part — all of which are valid for building form control references. But if you were using only the identifiers assigned to the form and form control elements (that is, none of the associated arrays of elements), the syntax is as follows:

```
document.formName.controlName
```

For example, consider the following simple form:

```
<form name="searchForm" action="cgi-bin/search.pl">
   <input type="text" name="entry">
   <input type="submit" name="sender" value="Search">
</form>
```

The following sample references to the text input control are all valid:

```
document.searchForm.entry
document.searchForm.elements[0]
document.forms["searchForm"].elements["entry"]
document.forms["searchForm"].entry
```

While form controls have several properties in common, some properties are unique to a particular control type or related types. For example, only a select object offers a property that reveals which item in its list is currently selected. But checkboxes and radio buttons both have a property that indicates whether the control is currently set to "on." Similarly, all text-oriented controls operate the same way for reading and modifying their content.

Having a good grasp of the scriptable features of form control objects is important to your success with JavaScript. In the next sections, you meet the most important form control objects and see how scripts interact with them.

Text-related objects

Each of the four text-related HTML form elements — input elements of the text, password, and hidden types, plus the textarea element — is an element in the document object hierarchy. All but the hidden object display themselves in the page, enabling users to enter information. These objects also display text information that changes in the course of using a page (although browsers capable of modern Dynamic HTML also allow the scripted change of other body text in a document).

To make these form control objects scriptable in a page, you do nothing special to their normal HTML tags — with the possible exception of assigning a name attribute. I strongly recommend assigning unique names to every text-related form control element if your scripts will be getting or setting properties or invoking their methods. Besides, if the form is actually submitted to a server program, the name attributes must be assigned in order for the server to receive the element's data.

Text Object Behavior

Many scripters look to JavaScript to solve what are perceived as shortcomings or behavioral anomalies with text-related objects in forms. I want to single these out early in your scripting experience so that they do not confuse you later.

First, only the most recent browsers let scripts reliably alter the font, font size, font style, and text alignment of a text object's content. You can access changes through the element's style-related properties (see Chapter 26).

Second, most browser forms practice a behavior that was recommended long ago as an informal standard by Web pioneers. When a form contains only one text `input` object, a press of the Enter/Return key while the text object has focus automatically submits the form. For two or more fields in browsers other than IE5/Mac and Safari, you need another way to submit the form (for example, a Submit button). This one-field submission scheme works well in many cases, such as the search page of most Web search sites. But if you are experimenting with simple forms containing only one field, you can submit the form with a press of the Enter/Return key. Submitting a form that has no other action or target specified means the page performs an unconditional reload—wiping out any information entered into the form. You can, however, cancel the submission through an `onsubmit` event handler in the form, as shown later in this chapter. Also, starting with version 4 browsers, you can script the press of the Enter/Return key in any text field to submit a form (see Chapter 25).

For the visible objects in this category, event handlers are triggered from many user actions, such as giving a field focus (getting the text insertion pointer in the field) and changing text (entering new text and leaving the field). Most of your text field actions are triggered by the change of text (the `onchange` event handler). In modern browsers, event handlers fire in response to individual keystrokes as well.

Without a doubt, the single most used property of a text-related element is the `value` property. This property represents the current contents of the text element. A script can retrieve and set its content at any time. Content of the `value` property is always a string. This may require conversion to numbers (see Chapter 6) if text fields are used to enter values for some math operations.

To demonstrate how a text field's `value` property can be read and written, Listing 9-1 provides a complete HTML page with a single-entry field. Its `onchange` event handler invokes the `upperMe()` function, which converts the text to uppercase. In the `upperMe()` function, the first statement assigns the text object reference to a more convenient variable: `field`. A lot goes on in the second statement of the function. The right side of the assignment statement performs a couple of key tasks. The reference to the `value` property of the object (`field.value`) evaluates to whatever content is in the text field at that instant. That string is then handed over to one of JavaScript's string functions, `toUpperCase()`, which converts the value to uppercase. The evaluated result of the right-side statement is then assigned to the second variable: `upperCaseVersion`. Nothing has changed yet in the text box. That comes in the third statement where the `value` property of the text box is assigned whatever the `upperCaseVersion` variable holds. The need for the second statement is more for learning purposes, so you can see the process more slowly. In practice, you can combine the actions of steps two and three into one power-packed statement:

```
field.value = field.value.toUpperCase();
```

Listing 9-1: **Getting and Setting a Text Object's value Property**

```
<html>
<head>
<title>Text Object value Property</title>
<script type="text/javascript">
function upperMe() {
   var field = document.forms[0].converter;
   var upperCaseVersion = field.value.toUpperCase();
   field.value = upperCaseVersion;
}
</script>
</head>
<body>
<form onsubmit="return false">
<input type="text" name="converter" value="sample" onchange="upperMe()">
</form>
</body>
</html>
```

Later in this chapter, I show you how to reduce even further the need for explicit references in functions such as `upperMe()` in Listing 9-1. In the meantime, notice for a moment the `onsubmit` event handler in the `<form>` tag. I delve more deeply into this event handler later in this chapter, but I want to point out the construction that prevents a single-field form from being submitted when you press the Enter key. If the event handler weren't there, a press of the Enter key would reload the page, returning the field to its original text. Try it!

The Button Object

I have used the button-type `input` element in many examples up to this point in the tutorial. The button is one of the simplest objects to script. In the simplified object model of this tutorial, the button object has only a few properties that are rarely accessed or modified in day-to-day scripts. Like the text object, the visual aspects of the button are governed not by HTML or scripts, but by the operating system and browser that the page visitor uses. By far, the most useful event handler of the button object is the `onclick` event handler. It fires whenever the user clicks the button. Simple enough. No magic here.

The Checkbox Object

A checkbox is also a simple element of the `form` object, but some of the properties may not be intuitive entirely. Unlike the `value` property of a plain button object (the text of the button label), the `value` property of a checkbox is any other text you want associated with the object. This text does not appear on the page in any fashion, but the property (initially set via the `value` attribute) might be important to a script that wants to know more about the purpose of the checkbox within the form.

The key property of a checkbox object is whether or not the box is checked. The `checked` property is a Boolean value: `true` if the box is checked, `false` if not. When you see that a property is a Boolean value, it's a clue that the value might be usable in an `if` or `if...else` condition expression. In Listing 9-2, the value of the `checked` property determines which alert box the user sees.

Listing 9-2: **The Checkbox Object's checked Property**

```
<html>
<head>
<title>Checkbox Inspector</title>
<script type="text/javascript">
function inspectBox() {
   if (document.forms[0].checkThis.checked) {
      alert("The box is checked.");
   } else {
      alert("The box is not checked at the moment.");
   }
}
</script>
</head>
<body>
<form>
<input type="checkbox" name="checkThis">Check here<br>
<input type="button" value="Inspect Box" onclick="inspectBox()">
</form>
</body>
</html>
```

Checkboxes are generally used as preferences setters, rather than as action inducers. While a checkbox object has an `onclick` event handler, a click of a checkbox should never do anything drastic, such as navigate to another page.

The Radio Object

Setting up a group of radio objects for scripting requires a bit more work. To let the browser manage the highlighting and unhighlighting of a related group of buttons, you must assign the same name to each of the buttons in the group. You can have multiple groups within a form, but each member of the same group must share the same name.

Assigning the same name to a form element forces the browser to manage the elements differently than if they each had a unique name. Instead, the browser maintains an array list of objects with the same name. The name assigned to the group becomes the name of the array. Some properties apply to the group as a whole; other properties apply to individual buttons within the group and must be addressed via array index references. For example, you can find out how many buttons are in a group by reading the `length` property of the group:

```
document.forms[0].groupName.length
```

If you want to find out if a particular button is currently highlighted — via the same checked property used for the checkbox — you must access the button element individually:

```
document.forms[0].groupName[0].checked
```

Listing 9-3 demonstrates several aspects of the radio button object, including how to look through a group of buttons to find out which one is checked and how to use the `value` attribute and corresponding property for meaningful work.

The page includes three radio buttons and a plain button. Each radio button's `value` attribute contains the full name of one of the Three Stooges. When the user clicks the button,

the `onclick` event handler invokes the `fullName()` function. In that function, the first statement creates a shortcut reference to the form. Next, a `for` repeat loop looks through all of the buttons in the `stooges` radio button group. An `if` construction looks at the `checked` property of each button. When a button is highlighted, the `break` statement bails out of the `for` loop, leaving the value of the `i` loop counter at the number where the loop broke ranks. The alert dialog box then uses a reference to the `value` property of the `i`th button so that the full name can be displayed in the alert.

Listing 9-3: **Scripting a Group of Radio Objects**

```
<html>
<head>
<title>Extracting Highlighted Radio Button</title>
<script type="text/javascript">
function fullName() {
   var form = document.forms[0];
   for (var i = 0; i < form.stooges.length; i++) {
      if (form.stooges[i].checked) {
         break;
      }
   }
   alert("You chose " + form.stooges[i].value + ".");
}
</script>
</head>

<body>
<form>
<p>Select your favorite Stooge:
<input type="radio" name="stooges" value="Moe Howard" checked>Moe
<input type="radio" name="stooges" value="Larry Fine">Larry
<input type="radio" name="stooges" value="Curly Howard">Curly<br>
<input type="button" name="Viewer" value="View Full Name..."
onclick="fullName()"></p>
</form>
</body>
</html>
```

The select Object

The most complex form control to script is the `select` element object. As you can see from the DOM Level 0 form object hierarchy diagram (see Figure 9-1), the `select` object is really a compound object: an object that contains an array of `option` objects. Moreover, you can establish this object in HTML to display itself as either a pop-up list or a scrolling list — the latter configurable to accept multiple selections by users. For the sake of simplicity at this stage, this lesson focuses on deployment as a pop-up list that allows only single selections.

Some properties belong to the entire `select` object; others belong to individual options inside the `select` object. If your goal is to determine which item the user selects, and you want the code to work on the widest range of browsers, you must use properties of both the `select` and `option` objects.

The most important property of the `select` object itself is the `selectedIndex` property, accessed as follows:

```
document.form[0].selectName.selectedIndex
```

This value is the index number of the currently selected item. As with most index counting schemes in JavaScript, the first item (the one at the top of the list) has an index of zero. The `selectedIndex` value is critical for enabling you to access properties of the selected option. Two important properties of an option item are `text` and `value`, accessed as follows:

```
document.forms[0].selectName.options[n].text
document.forms[0].selectName.options[n].value
```

The `text` property is the string that appears onscreen in the `select` object. It is unusual for this information to be exposed as a `form` object property because in the HTML that generates a `select` object, the text is defined as an `<option>` tag's nested text. But inside the `<option>` tag, you can set a `value` attribute, which, like the radio buttons shown earlier, enables you to associate some hidden string information with each visible entry in the list.

To read the `value` or `text` property of a selected option most efficiently for all browsers, you can use the `select` object's `selectedIndex` property as an index value to the option. References for this kind of operation get pretty long, so take the time to understand what's happening here. In the following function, the first statement creates a shortcut reference to the `select` object. The `selectedIndex` property of the `select` object is then substituted for the `index` value of the `options` array of that same object:

```
function inspect() {
   var list = document.forms[0].choices;
   var chosenItemText = list.options[list.selectedIndex].value;
}
```

To bring a `select` object to life, use the `onchange` event handler. As soon as a user makes a new selection in the list, this event handler runs the script associated with that event handler. Listing 9-4 shows a common application for a `select` object. Its text entries describe places to go in and out of a Web site, while the `value` attributes hold the URLs for those locations. When a user makes a selection in the list, the `onchange` event handler triggers a script that extracts the `value` property of the selected option and assigns that value to the `location.href` object property to effect the navigation. Under JavaScript control, this kind of navigation doesn't need a separate Go button on the page.

Listing 9-4: **Navigating with a select Object**

```
<html>
<head>
<title>Select Navigation</title>
<script type="text/javascript">
function goThere() {
   var list = document.forms[0].urlList;
   location.href = list.options[list.selectedIndex].value;
}
</script>
</head>
```

Continued

Listing 9-4 *(continued)*

```
<body>
<form>
Choose a place to go:
<select name="urlList" onchange="goThere()">
   <option selected value="index.html">Home Page
   <option value="store.html">Shop Our Store
   <option value="policies.html">Shipping Policies
   <option value="http://www.google.com">Search the Web
</select>
</form>
</body>
</html>
```

Recent browsers also expose the `value` property of the selected option item by way of the `value` property of the `select` object. This is certainly a logical and convenient shortcut, and you can use it safely if your target browsers include IE, Mozilla-based browsers, and Safari.

There is much more to the `select` object, including the ability to change the contents of a list in newer browsers. Chapter 24 covers the `select` object in depth.

Passing Form Data and Elements to Functions

In all of the examples so far in this lesson, when an event handler invokes a function that works with form elements, the form or form control is explicitly referenced in the function. But valuable shortcuts exist for transferring information about the form or form control directly to the function without dealing with those typically long references that start with the `window` or `document` object level.

JavaScript features a keyword — `this` — that always refers to whatever object contains the script in which the keyword is used. Thus, in an `onchange` event handler for a text field, you can pass a reference to the text object to the function by inserting the `this` keyword as a parameter to the function:

```
<input type="text" name="entry" onchange="upperMe(this)">
```

At the receiving end, the function defines a parameter variable that turns that reference into a variable that the rest of the function can use:

```
function upperMe(field) {
   statement[s]
}
```

The name you assign to the function's parameter variable is purely arbitrary, but it is helpful to give it a name that expresses what the reference is. Importantly, this reference is a "live" connection back to the object. Therefore, statements in the script can get and set property values of the object at will.

For other functions, you may wish to receive a reference to the entire form, rather than just the object calling the function. This is certainly true if the function needs to access other elements of the same form. Because every form control object contains a property that points to the containing form, you can use the `this` keyword to reference the current control, plus its `form` property as `this.form`, as in

```
<input type="button" value="Click Here" onclick="inspect(this.form)">
```

The function definition should then have a parameter variable ready to be assigned to the form object reference. Again, you decide the name of the variable. I tend to use the variable name `form` as a way to remind me exactly what kind of object is referenced.

```
function inspect(form) {
   statement[s]
}
```

Listing 9-5 demonstrates passing both an individual form element and the entire form in the performance of two separate acts. This page makes believe it is connected to a database of Beatles songs. When you click the Process Data button, it passes the `form` object, which the `processData()` function uses to access the group of radio buttons inside a `for` loop. Additional references using the passed `form` object extract the `value` properties of the selected radio button and the text field.

The text field has its own event handler, which passes just the text field to the `verifySong()` function. Notice how short the reference is to reach the `value` property of the song field inside the function.

Listing 9-5: **Passing a Form Object and Form Element to Functions**

```
<html>
<head>
<title>Beatle Picker</title>
<script type="text/javascript">
function processData(form) {
   for (var i = 0; i < form.Beatles.length; i++) {
      if (form.Beatles[i].checked) {
         break
      }
   }
   // assign values to variables for convenience
   var beatle = form.Beatles[i].value
   var song = form.song.value
   alert("Checking whether " + song + " features " + beatle + "...")
}

function verifySong(entry) {
   var song = entry.value
   alert("Checking whether " + song + " is a Beatles tune...")
}
</script>
</head>
```

Continued

Listing 9-5 *(continued)*

```
<body>
<form onsubmit="return false">
<p>Choose your favorite Beatle:
<input type="radio" name="Beatles" value="John Lennon" checked>John
<input type="radio" name="Beatles" value="Paul McCartney">Paul
<input type="radio" name="Beatles" value="George Harrison">George
<input type="radio" name="Beatles" value="Ringo Starr">Ringo</p>

<p>Enter the name of your favorite Beatles song:<br>
<input type="text" name="song" value = "Eleanor Rigby"
onchange="verifySong(this)">
<input type="button" name="process" value="Process Request..."
onclick="processData(this.form)"</p>
</form>
</body>
</html>
```

Get to know the usage of the `this` keyword in passing `form` and `form element` objects to functions. The technique not only saves you typing in your code, but it also ensures accuracy in references to those objects.

Submitting and Prevalidating Forms

The scripted equivalent of submitting a form is the `form` object's `submit()` method. All you need in the statement is a reference to the form and this method:

```
document.forms[0].submit();
```

Before you get ideas about having a script silently submit a form to a URL bearing the `mailto:` protocol, forget it. Because such a scheme could expose visitors' e-mail addresses without their knowledge, `mailto:` submissions are either blocked or revealed to users as a security precaution.

Before a form is submitted, you may wish to perform some last-second validation of data in the form or in other scripting (for example, changing the form's `action` property based on user choices). You can do this in a function invoked by the form's `onsubmit` event handler. Specific validation routines are beyond the scope of this tutorial (but are explained in substantial detail in Chapter 43 on the CD-ROM), but I want to show you how the `onsubmit` event handler works.

You can let the results of a validation function cancel a submission if the validation shows some incorrect data or empty fields. To control submission, the `onsubmit` event handler must evaluate to `return true` (to allow submission to continue) or `return false` (to cancel submission). This is a bit tricky at first because it involves more than just having the function called by the event handler return `true` or `false`. The `return` keyword must be part of the final evaluation.

Listing 9-6 shows a page with a simple validation routine that ensures all fields have something in them before allowing submission to continue. (The sample form has no `action` attribute, so this sample form doesn't get sent to the server.) Notice how the `onsubmit` event handler (which passes a reference to the `form` object as a parameter — in this case the `this` keyword points to the `form` object because its tag holds the event handler) includes the `return` keyword before the function name. When the function returns its `true` or `false` value, the event handler evaluates to the requisite `return true` or `return false`.

Listing 9-6: **Last-Minute Checking Before Form Submission**

```
<html>
<head>
<title>Validator</title>
<script type="text/javascript">
function checkForm(form) {
   for (var i = 0; i < form.elements.length; i++) {
      if (form.elements[i].value == "") {
         alert("Fill out ALL fields.");
         return false;
      }
   }
   return true;
}
</script>
</head>

<body>
<form onsubmit="return checkForm(this)">
Please enter all requested information:<br>
First Name:<input type="text" name="firstName"><br>
Last Name:<input type="text" name="lastName"><br>
Rank:<input type="text" name="rank"><br>
Serial Number:<input type="text" name="serialNumber"><br>

<input type="submit">
</form>
</body>
</html>
```

One quirky bit of behavior involving the `submit()` method and `onsubmit` event handler needs explanation. While you might think (and logically so, in my opinion) that the `submit()` method would be the exact scripted equivalent of a click of a real Submit button, it's not. The `submit()` method does not cause the form's `onsubmit` event handler to fire at all. If you want to perform validation on a form submitted via the `submit()` method, invoke the validation in the script function that ultimately calls the `submit()` method.

So much for the basics of forms and form controls. In the next chapter, you step away from HTML for a moment to look at more advanced JavaScript core language items: strings, math, and dates.

Exercises

1. Rework Listings 9-1, 9-2, 9-3, and 9-4 so that the script functions all receive the most efficient form or form element references directly from the invoking event handler.

2. For the following form (assume it's the only form on the page), write at least 10 ways to reference the text input field as an object in all modern scriptable browsers.

```
<form name="subscription" action="cgi-bin/maillist.pl" method="post">
   <input type="text" id="email" name="email">
   <input type="submit">
</form>
```

3. In the following HTML tag, what kind of information do you think is being passed with the event handler? Write a function that displays in an alert dialog box the information being passed.

```
<input type="text" name="phone" onchange="format(this.value)">
```

4. A document contains two forms named `specifications` and `accessories`. In the `accessories` form is a field named `acc1`. Write at least two different statements that set the contents of that field to `Leather Carrying Case`.

5. Create a page that includes a `select` object to change the background color of the current page. The property that you need to set is `document.bgColor`, and the three values you should offer as options are `red`, `yellow`, and `green`. In the `select` object, the colors should display as `Stop`, `Caution`, and `Go`.

✦ ✦ ✦

CHAPTER 10

Strings, Math, and Dates

In This Chapter

How to modify strings with common string methods

When and how to use the `Math` object

How to use the `Date` object

For most of the lessons in the tutorial so far, the objects at the center of attention belong to the document object model. But as indicated in Chapter 2, a clear dividing line exists between the document object model and the JavaScript language. The language has some of its own objects that are independent of the document object model. These objects are defined such that if a vendor wished to implement JavaScript as the programming language for an entirely different kind of product, the language would still use these core facilities for handling text, advanced math (beyond simple arithmetic), and dates. You can find formal specifications of these objects in the ECMA-262 recommendation.

Core Language Objects

It is often difficult for newcomers to programming — or even experienced programmers who have not worked in object-oriented worlds before — to think about objects, especially when attributed to "things" that don't seem to have a physical presence. For example, it doesn't require lengthy study to grasp the notion that a button on a page is an object. It has several physical properties that make perfect sense. But what about a string of characters? As you learn in this chapter, in an object-based environment such as JavaScript, everything that moves is treated as an object — each piece of data from a Boolean value to a date. Each such object probably has one or more properties that help define the content; such an object may also have methods associated with it to define what the object can do or what you can do to the object.

I call all objects that are not part of the document object model *core language objects*. You can see the full complement of them in the Quick Reference in Appendix A. In this chapter, I focus on the `String`, `Math`, and `Date` objects.

String Objects

You have already used `String` objects many times in earlier lessons. A *string* is any text inside a quote pair. A quote pair consists of either double quotes or single quotes. This allows one string to nest inside

another, as often happens in event handlers. In the following example, the `alert()` method requires a quoted string as a parameter, but the entire method call also must be inside quotes:

```
onclick="alert('Hello, all')"
```

JavaScript imposes no practical limit on the number of characters that a string can hold. However, most older browsers have a limit of 255 characters in length for a script statement. This limit is sometimes exceeded when a script includes a lengthy string that is to become scripted content in a page. You need to divide such lines into smaller chunks using techniques described in a moment.

You have two ways to assign a string value to a variable. The simplest is a basic assignment statement:

```
var myString = "Howdy";
```

This works perfectly well except in some exceedingly rare instances. Beginning with Navigator 3 and Internet Explorer 4, you can also create a string object using the more formal syntax that involves the `new` keyword and a constructor function (that is, it "constructs" a new object):

```
var myString = new String("Howdy");
```

Whichever way you use to initialize a variable with a string, the variable receiving the assignment can respond to all `String` object methods.

Joining strings

Bringing two strings together as a single string is called *concatenating* strings, a term you learned in Chapter 6. String concatenation requires one of two JavaScript operators. Even in your first script in Chapter 3, you saw how the addition operator (+) linked multiple strings together to produce the text dynamically written to the loading Web page:

```
document.write(" of <span class='highlight'>" + navigator.appName + "</span>.");
```

As valuable as that operator is, another operator can be even more scripter friendly. This operator is helpful when you are assembling large strings in a single variable. The strings may be so long or cumbersome that you need to divide the building process into multiple statements. The pieces may be combinations of *string literals* (strings inside quotes) or variable values. The clumsy way to do it (perfectly doable in JavaScript) is to use the addition operator to append more text to the existing chunk:

```
var msg = "Four score";
msg = msg + " and seven";
msg = msg + " years ago,";
```

But another operator, called the *add-by-value operator*, offers a handy shortcut. The symbol for the operator is a plus and equal sign together (+=). This operator means "append the stuff on the right of me to the end of the stuff on the left of me." Therefore, the preceding sequence is shortened as follows:

```
var msg = "Four score";
msg += " and seven";
msg += " years ago,";
```

You can also combine the operators if the need arises:

```
var msg = "Four score";
msg += " and seven" + " years ago";
```

I use the add-by-value operator a lot when accumulating HTML text to be written to the current document or another window.

String methods

Of all the core JavaScript objects, the `String` object has the most diverse collection of methods associated with it. Many methods are designed to help scripts extract segments of a string. Another group, rarely used and now obsolete in favor of CSS, wraps a string with one of several style-oriented tags (a scripted equivalent of tags for font size, style, and the like).

To use a string method, the string being acted upon becomes part of the reference followed by the method name. All methods return a value of some kind. Most of the time, the returned value is a converted version of the string object referred to in the method call — but the original string is still intact. To capture the modified version, you need to assign the results of the method to a variable:

```
var result = string.methodName();
```

The following sections introduce you to several important string methods available to all browser brands and versions.

Changing string case

Two methods convert a string to all uppercase or lowercase letters:

```
var result = string.toUpperCase();
var result = string.toLowerCase();
```

Not surprisingly, you must observe the case of each letter of the method names if you want them to work. These methods come in handy when your scripts need to compare strings that may not have the same case (for example, a string in a lookup table compared with a string typed by a user). Because the methods don't change the original strings attached to the expressions, you can simply compare the evaluated results of the methods:

```
var foundMatch = false;
if (stringA.toUpperCase() == stringB.toUpperCase()) {
   foundMatch = true;
}
```

String searches

You can use the `string.indexOf()` method to determine if one string is contained by another. Even within JavaScript's own object data, this can be useful information. For example, the `navigator.userAgent` property reveals a lot about the browser that loads the page. A script can investigate the value of that property for the existence of, say, "Win" to determine that the user has a Windows operating system. That short string might be buried somewhere inside a long string, and all the script needs to know is whether the short string is present in the longer one — wherever it might be.

The `string.indexOf()` method returns a number indicating the index value (zero based) of the character in the larger string where the smaller string begins. The key point about this method is that if no match occurs, the returned value is `-1`. To find out whether the smaller string is inside, all you need to test is whether the returned value is something other than `-1`.

Two strings are involved with this method: the shorter one and the longer one. The longer string is the one that appears in the reference to the left of the method name; the shorter string is inserted as a parameter to the `indexOf()` method. To demonstrate the method in action, the following fragment looks to see if the user is running Windows:

```
var isWindows = false;
if (navigator.userAgent.indexOf("Win") != -1) {
   isWindows = true;
}
```

The operator in the `if` construction's condition (`!=`) is the inequality operator. You can read it as meaning "is not equal to."

Extracting copies of characters and substrings

To extract a single character at a known position within a string, use the `charAt()` method. The parameter of the method is an index number (zero based) of the character to extract. When I say *extract*, I don't mean delete, but rather grab a snapshot of the character. The original string is not modified in any way.

For example, consider a script in a main window that is capable of inspecting a variable, `stringA`, in another window that displays map images of different corporate buildings. When the window has a map of Building C in it, the `stringA` variable contains "Building C." The building letter is always at the tenth character position of the string (or number 9 in a zero-based counting world), so the script can examine that one character to identify the map currently in that other window:

```
var stringA = "Building C";
var bldgLetter = stringA.charAt(9);
   // result: bldgLetter = "C"
```

Another method — `string.substring()` — enables you to extract a contiguous sequence of characters, provided you know the starting and ending positions of the substring of which you want to grab a copy. Importantly, the character at the ending position value is not part of the extraction: All applicable characters, up to but not including that character, are part of the extraction. The string from which the extraction is made appears to the left of the method name in the reference. Two parameters specify the starting and ending index values (zero based) for the start and end positions:

```
var stringA = "banana daiquiri";
var excerpt = stringA.substring(2,6);
   // result: excerpt = "nana"
```

String manipulation in JavaScript is fairly cumbersome compared to some other scripting languages. Higher-level notions of words, sentences, or paragraphs are completely absent. Therefore, sometimes it takes a bit of scripting with string methods to accomplish what seems like a simple goal. And yet you can put your knowledge of expression evaluation to the test as you assemble expressions that utilize heavily nested constructions. For example, the following fragment needs to create a new string that consists of everything from the larger string except the first word. Assuming the first word of other strings can be of any length, the second statement utilizes the `string.indexOf()` method to look for the first space character and adds 1 to that value to serve as the starting index value for an outer

`string.substring()` method. For the second parameter, the `length` property of the string provides a basis for the ending character's index value (one more than the actual character needed).

```
var stringA = "The United States of America";
var excerpt = stringA.substring(stringA.indexOf(" ") + 1, stringA.length);
   // result: excerpt = "United States of America"
```

Creating statements like this one is not something you are likely to enjoy over and over again, so in Chapter 27 I show you how to create your own library of string functions you can reuse in all of your scripts that need their string-handling facilities. More powerful string-matching facilities are built into today's browsers by way of regular expressions (see Chapter 27 and Chapter 42 on the CD-ROM).

The Math Object

JavaScript provides ample facilities for math — far more than most scripters who don't have a background in computer science and math will use in a lifetime. But every genuine programming language needs these powers to accommodate clever programmers who can make windows fly in circles on the screen.

The `Math` object contains all of these powers. This object is unlike most of the other objects in JavaScript in that you don't generate copies of the object to use. Instead your scripts summon a single `Math` object's properties and methods. (One `Math` object actually occurs per window or frame, but this has no impact whatsoever on your scripts.) Programmers call this kind of fixed object a *static object*. That `Math` object (with an uppercase M) is part of the reference to the property or method. Properties of the `Math` object are constant values, such as pi and the square root of two:

```
var piValue = Math.PI;
var rootOfTwo = Math.SQRT2;
```

`Math` object methods cover a wide range of trigonometric functions and other math functions that work on numeric values already defined in your script. For example, you can find which of two numbers is the larger:

```
var larger = Math.max(value1, value2);
```

Or you can raise one number to a power of 10:

```
var result = Math.pow(value1, 10);
```

More common, perhaps, is the method that rounds a value to the nearest integer value:

```
var result = Math.round(value1);
```

Another common request of the `Math` object is a random number. The `Math.random()` method returns a floating-point number between 0 and 1. If you design a script to act like a card game, you need random integers between 1 and 52; for dice, the range is 1 to 6 per die. To generate a random integer between zero and any top value, use the following formula:

```
Math.floor(Math.random() * (n + 1))
```

where `n` is the top number. (`Math.floor` returns the integer part of any floating-point number.) To generate random numbers between 1 and any higher number, use this formula:

```
Math.floor(Math.random() * n) + 1
```

where n equals the top number of the range. For the dice game, the formula for each die is

```
newDieValue = Math.floor(Math.random() * 6) + 1;
```

To see this, enter the right-hand part of the preceding statement in the top text box of The Evaluator Jr. and repeatedly click the Evaluate button.

One bit of help JavaScript doesn't offer except in IE5.5+ and Mozilla-based browsers is a way to specify a number-formatting scheme. Floating-point math can display more than a dozen numbers to the right of the decimal. Moreover, results can be influenced by each operating system's platform-specific floating-point errors, especially in earlier versions of scriptable browsers. For other browsers you must perform any number formatting — for dollars and cents, for example — through your own scripts. Chapter 28 provides an example.

The Date Object

Working with dates beyond simple tasks can be difficult business in JavaScript. A lot of the difficulty comes with the fact that dates and times are calculated internally according to *Greenwich Mean Time (GMT)* — provided the visitor's own internal PC clock and control panel are set accurately. As a result of this complexity, better left for Chapter 29, this section of the tutorial touches on only the basics of the JavaScript Date object.

A scriptable browser contains one global Date object (in truth, one Date object per window) that is always present, ready to be called upon at any moment. The Date object is another one of those static objects. When you wish to work with a date, such as displaying today's date, you need to invoke the Date object constructor function to obtain an instance of a Date object tied to a specific time and date. For example, when you invoke the constructor without any parameters, as in

```
var today = new Date();
```

the Date object takes a snapshot of the PC's internal clock and returns a date object for that instant. Notice the distinction between the static Date object and a Date object instance, which contains an actual date value. The variable, today, contains not a ticking clock, but a value that you can examine, tear apart, and reassemble as needed for your script.

Internally, the value of a Date object instance is the time, in milliseconds, from zero o'clock on January 1, 1970, in the Greenwich Mean Time zone — the world standard reference point for all time conversions. That's how a Date object contains both date and time information.

You can also grab a snapshot of the Date object for a particular date and time in the past or future by specifying that information as parameters to the Date object constructor function:

```
var someDate = new Date("Month dd, yyyy hh:mm:ss");
var someDate = new Date("Month dd, yyyy");
var someDate = new Date(yy,mm,dd,hh,mm,ss);
var someDate = new Date(yy,mm,dd);
var someDate = new Date(GMT milliseconds from 1/1/1970);
```

If you attempt to view the contents of a raw Date object, JavaScript converts the value to the local time zone string as indicated by your PC's control panel setting. To see this in action, use The Evaluator Jr.'s top text box to enter the following:

```
new Date();
```

Your PC's clock supplies the current date and time as the clock calculates them (even though JavaScript still stores the date object's millisecond count in the GMT zone). You can,

however, extract components of the `Date` object via a series of methods that you apply to a `Date` object instance. Table 10-1 shows an abbreviated listing of these properties and information about their values.

Table 10-1: Some Date Object Methods

Method	*Value Range*	*Description*
dateObj.getTime()	0-...	Milliseconds since 1/1/70 00:00:00 GMT
dateObj.getYear()	70-...	Specified year minus 1900; four-digit year for 2000+
dateObj.getFullYear()	1970-...	Four-digit year (Y2K-compliant); version 4+ browsers
dateObj.getMonth()	0-11	Month within the year (January = 0)
dateObj.getDate()	1-31	Date within the month
dateObj.getDay()	0-6	Day of week (Sunday = 0)
dateObj.getHours()	0-23	Hour of the day in 24-hour time
dateObj.getMinutes()	0-59	Minute of the specified hour
dateObj.getSeconds()	0-59	Second within the specified minute
dateObj.setTime(*val*)	0-...	Milliseconds since 1/1/70 00:00:00 GMT
dateObj.setYear(*val*)	70-...	Specified year minus 1900; four-digit year for 2000+
dateObj.setMonth(*val*)	0-11	Month within the year (January = 0)
dateObj.setDate(*val*)	1-31	Date within the month
dateObj.setDay(*val*)	0-6	Day of week (Sunday = 0)
dateObj.setHours(*val*)	0-23	Hour of the day in 24-hour time
dateObj.setMinutes(*val*)	0-59	Minute of the specified hour
dateObj.setSeconds(*val*)	0-59	Second within the specified minute

Be careful about values whose ranges start with zero, especially the months. The `getMonth()` and `setMonth()` method values are zero based, so the numbers are one less than the month numbers you are accustomed to working with (for example, January is 0, December is 11).

You may notice one difference about the methods that set values of a `Date` object. Rather than returning some new value, these methods actually modify the value of the `Date` object referenced in the call to the method.

Date Calculations

Performing calculations with dates frequently requires working with the millisecond values of the `Date` objects. This is the surest way to compare date values. To demonstrate a few `Date` object machinations, Listing 10-1 displays the current date and time as the page loads. Another script shows one way to calculate the date and time seven days from the current date and time value.

Listing 10-1: **Date Object Calculations**

```
<html>
<head>
<title>Date Calculation</title>
<script type="text/javascript">
function nextWeek() {
   var todayInMS = today.getTime();
   var nextWeekInMS = todayInMS + (60 * 60 * 24 * 7 * 1000);
   return new Date(nextWeekInMS);
}
</script>
</head>

<body>
Today is:
<script type="text/javascript">
var today = new Date();
document.write(today);
</script>
<br>
Next week will be:
<script type="text/javascript">
document.write(nextWeek());
</script>
</body>
</html>
```

In the Body portion, the first script runs as the page loads, setting a global variable (`today`) to the current date and time. The string equivalent is written to the page. In the second Body script, the `document.write()` method invokes the `nextWeek()` function to get a value to display. That function utilizes the `today` global variable, copying its millisecond value to a new variable: `todayInMS`. To get a date seven days from now, the next statement adds the number of milliseconds in seven days (60 seconds times 60 minutes times 24 hours times seven days times 1000 milliseconds) to today's millisecond value. The script now needs a new `Date` object calculated from the total milliseconds. This requires invoking the `Date` object constructor with the milliseconds as a parameter. The returned value is a `Date` object, which is automatically converted to a string version for writing to the page.

To add or subtract time intervals from a `Date` object, you can use a shortcut that doesn't require the millisecond conversions. By combining the date object's set and get methods, you can let the `Date` object work out the details. For example, in Listing 10-1 you could eliminate the function entirely, and let the following two statements in the second Body script obtain the desired result:

```
today.setDate(today.getDate() + 7);
document.write(today);
```

Because JavaScript tracks the date and time internally as milliseconds, the accurate date appears in the end, even if the new date is into the next month. JavaScript automatically takes care of figuring out how many days there are in a month as well as in leap years.

Many other quirks and complicated behavior await you if you script dates in your page. As later chapters demonstrate, however, the results may be worth the effort.

Exercises

1. Create a Web page that has one form field for entry of the user's e-mail address and a Submit button. Include a pre-submission validation routine that verifies that the text field has the @ symbol found in all e-mail addresses before you allow submission of the form.

2. Given the string `"Internet Explorer"`, fill in the blanks of the `string.substring()` method parameters here that yield the results shown to the right of each method call:

```
var myString = "Internet Explorer";
myString.substring(___,___)     // result = "Int"
myString.substring(___,___)     // result = "plorer"
myString.substring(___,___)     // result = "net Exp"
```

3. Fill in the rest of the function in the listing that follows so that it looks through every character of the entry field and counts how many times the letter "e" appears in the field. (Hint: All that is missing is a `for` repeat loop.)

```
<html>
<head>
<title>Wheel o' Fortuna</title>
<script type="text/javascript">
function countE(form) {
   var count = 0;
   var inputString = form.mainstring.value.toLowerCase();
   missing code
   var msg = "The string has " + count;
   msg += " instances of the letter e.";
   alert(msg);
}
</script>
</head>

<body>
<form>
Enter any string: <input type="text" name="mainstring" size="30"><br>
<input type="button" value="Count the Es"
onclick="countE(this.form)">
</form>
</body>
</html>
```

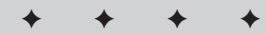
4. Create a page that has two fields and one button. The button should trigger a function that generates two random numbers between 1 and 6, placing each number in one of the fields. (Think of using this page as a substitute for rolling a pair of dice in a board game.)

5. Create a page that displays the number of days between today and next Christmas.

✦ ✦ ✦

CHAPTER 11

Scripting Frames and Multiple Windows



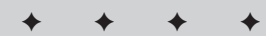
In This Chapter

Relationships among frames in the browser window

How to access objects and values in other frames

How to control navigation of multiple frames

Communication skills between separate windows



One of the attractive aspects of JavaScript for some applications on the client is that it allows user actions in one frame or window to influence what happens in other frames and windows. In this section of the tutorial, you extend your existing knowledge of object references to the realm of multiple frames and windows.

Frames: Parents and Children

You've see in earlier top-level hierarchy illustrations (such as Figure 4-1) that the `window` object is at the very top of the chart. The `window` object also has several synonyms, which stand in for the `window` object in special cases. For instance, in Chapter 8, you learned that `self` is synonymous with `window` when the reference applies to the same window that contains the script's document. In this lesson, you learn the roles of three other references that point to objects behaving as windows — `frame`, `top`, and `parent`.

Loading an ordinary HTML document into the browser creates a model in the browser that starts out with one `window` object and the document it contains. (The document likely contains other elements, but I'm not concerned with that stuff yet.) The top rungs of the hierarchy model are as simple as can be, as shown in Figure 11-1. This is where references begin with `window` or `self` (or with `document` because the current window is assumed).

The instant a framesetting document loads into a browser, the browser starts building a slightly different hierarchy model. The precise structure of that model depends entirely on the structure of the frameset defined in that framesetting document. Consider the following skeletal frameset definition:

```
<html>
<frameset cols="50%,50%">
   <frame name="leftFrame" src="somedoc1.html">
   <frame name="rightFrame" src="somedoc2.html">
</frameset>
</html>
```

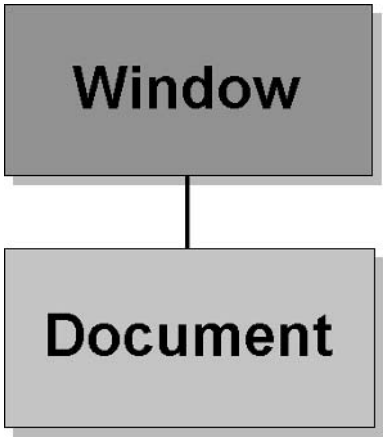


Figure 11-1: Single-frame window and document hierarchy.

This HTML splits the browser window into two frames side by side, with a different document loaded into each frame. The model is concerned only with structure — it doesn't care about the relative sizes of the frames or whether they're set up in columns or rows.

Framesets establish relationships among the frames in the collection. Borrowing terminology from the object-oriented programming world, the framesetting document loads into a *parent window*. Each of the frames defined in that parent window document is a *child frame*. Figure 11-2 shows the hierarchical model of a two-frame environment. This illustration reveals a lot of subtleties about the relationships among framesets and their frames.

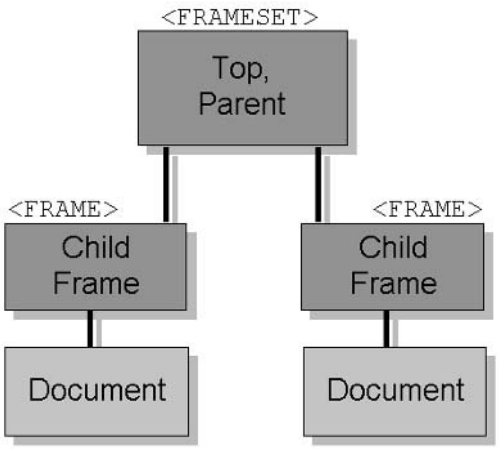


Figure 11-2: Two-frame window and document hierarchy.

It is often difficult at first to visualize the frameset as a window object in the hierarchy. After all, with the exception of the URL showing in the Location/Address field, you don't see anything about the frameset in the browser. But that window object exists in the object model. Notice, too, that in the diagram the framesetting parent window has no document object showing. This may also seem odd because the window obviously requires an HTML file containing the specifications for the frameset. In truth, the parent window has a `document` object associated with it, but it is omitted from the diagram to better portray the relationships among parent and child windows. A frameset parent's document cannot contain most of the typical HTML objects such as forms and controls, so references to the parent's document are rarely, if ever, used.

If you add a script to the framesetting document that needs to access a property or method of that window object, references are like any single-frame situation. Think about the point of view of a script located in that window. Its immediate universe is the very same window.

Things get more interesting when you start looking at the child frames. Each of these frames contains a `document` object whose content you see in the browser window. And the structure is such that each frame's document is entirely independent of the other. It is as if each document lived in its own browser window. Indeed, that's why each child frame is also a window type of object. A frame has the same kinds of properties and methods of the `window` object that occupies the entire browser.

From the point of view of either child window in Figure 11-2, its immediate container is the `parent` window. When a `parent` window is at the very top of the hierarchical model loaded in the browser, that window is also referred to as the `top` object.

References among Family Members

Given the frame structure of Figure 11-2, it's time to look at how a script in any one of those windows can access objects, functions, or variables in the others. An important point to remember about this facility is that if a script has access to an object, function, or global variable in its own window, that same item can be reached by a script from another frame in the hierarchy (provided both documents come from the same Web server).

A script reference may need to take one of three possible routes in the two-generation hierarchy described so far: parent to child; child to parent; or child to child. Each of the paths between these windows requires a different reference style.

Parent-to-child references

Probably the least common direction taken by references is when a script in the parent document needs to access some element of one of its frames. The parent contains two or more frames, which means the parent maintains an array of the child frame objects. You can address a frame by array syntax or by the name you assign to it with the `name` attribute inside the `<frame>` tag. In the following examples of reference syntax, I substitute a placeholder named *`ObjFuncVarName`* for whatever object, function, or global variable you intend to access in the distant window or frame. Remember that each visible frame contains a `document` object, which is generally the container of elements you script — be sure references to the elements include `document`. With that in mind, a reference from a parent to one of its child frames follows any of these models:

```
[window.]frames[n].ObjFuncVarName
[window.]frames["frameName"].ObjFuncVarName
[window.]frameName.ObjFuncVarName
```

Numeric index values for frames are based on the order in which their <frame> tags appear in the framesetting document. You will make your life easier, however, if you assign recognizable names to each frame and use the frame's name in the reference.

Child-to-parent references

It is not uncommon to place scripts in the parent (in the Head portion) that multiple child frames or multiple documents in a frame use as a kind of script library. By loading in the frameset, these scripts load only once while the frameset is visible. If other documents from the same server load into the frames over time, they can take advantage of the parent's scripts without having to load their own copies into the browser.

From the child's point of view, the next level up the hierarchy is called the parent. Therefore, a reference from a child frame to items at the parent level is simply

```
parent.ObjFuncVarName
```

If the item accessed in the parent is a function that returns a value, the returned value transcends the parent/child borders down to the child without hesitation.

When the parent window is also at the very top of the object hierarchy currently loaded into the browser, you can optionally refer to it as the *top window*, as in

```
top.ObjFuncVarName
```

Using the top reference can be hazardous if for some reason your Web page gets displayed in some other Web site's frameset. What is your top window is not the master frameset's top window. Therefore, I recommend using the parent reference whenever possible (unless you want to blow away an unwanted framer of your Web site).

Child-to-child references

The browser needs a bit more assistance when it comes to getting one child window to communicate with one of its siblings. One of the properties of any window or frame is its parent (whose value is null for a single window). A reference must use the parent property to work its way out of the current frame to a point that both child frames have in common — the parent in this case. Once the reference is at the parent level, the rest of the reference can carry on as if starting at the parent. Thus, from one child to one of its siblings, you can use any of the following reference formats:

```
parent.frames[n].ObjFuncVarName
parent.frames["frameName"].ObjFuncVarName
parent.frameName.ObjFuncVarName
```

A reference from the other sibling back to the first looks the same, but the frames[] array index or *frameName* part of the reference differs. Of course, much more complex frame hierarchies exist in HTML. Even so, the object model and referencing scheme provide a solution for the most deeply nested and gnarled frame arrangement you can think of — following the same precepts you just learned.

Frame Scripting Tips

One of the first mistakes that frame scripting newcomers make is writing immediate script statements that call upon other frames while the pages load. The problem here is that you cannot rely on the document loading sequence to follow the frameset source code order. All you know for sure is that the parent document *begins* loading first. Regardless of the order of `<frame>` tags, child frames can begin loading at any time. Moreover, a frame's loading time depends on other elements in the document, such as images or Java applets.

Fortunately, you can use a certain technique to initiate a script once all of the documents in the frameset are completely loaded. Just as the `onload` event handler for a document fires when that document is fully loaded, a parent's `onload` event handler fires after the `onload` event handler in its child frames is fired. Therefore, you can specify an `onload` event handler in the `<frameset>` tag. That handler might invoke a function in the framesetting document that then has the freedom to tap the objects, functions, or variables of all frames throughout the object hierarchy.

Make special note that a reference to a frame as a type of window object is quite separate from a reference to the `frame` element object. An element object is one of those DOM element nodes in the document node tree (see Chapter 4). The properties and methods of this node differ from the properties and methods that accrue to a window-type object. It may be a difficult distinction to grasp, but it's an important one. The way you reference a frame — as a window object or element node — determines which set of properties and methods are available to your scripts. See Chapter 15 for a more detailed introduction to element node scripting.

If you start with a reference to the `frame` element object, you can still reach a reference to the `document` object loaded into that frame. But the syntax is different depending on the browser. IE4+ and Safari let you use the same `document` reference as for a window; Mozilla-based browsers follow the W3C DOM standard more closely, using the `contentDocument` property of the frame element. To accommodate both syntaxes you can build a reference as follows:

```
var docObj;
var frameObj = document.getElementById("myFrame");
if (frameObj.contentDocument) {
   docObj = frameObj.contentDocument;
} else {
   docObj = frameObj.document;
}
```

About iframe Elements

The `iframe` element is supported as a scriptable object in IE4+, Mozilla-based browsers, and Safari (among other modern browsers). It is often used as a way to fetch and load HTML or XML from a server without disturbing the current HTML page. Therefore it's not uncommon for an `iframe` to be hidden from view, while scripts handle all of the processing between it and the main document.

An `iframe` element becomes another member of the current window's `frames` collection. But you may also reference the `iframe` as an element object through W3C DOM `document.getElementById()` terminology. As with the distinction between the traditional frame-as-window object and DOM element object, a script reference to the `document` object within an `iframe` element object needs special handling. See Chapter 16 for additional details.

Controlling Multiple Frames — Navigation Bars

If you are enamored of frames as a way to help organize a complex Web page, you may find yourself wanting to control the navigation of one or more frames from a static navigation panel. Here, I demonstrate scripting concepts for such control using an application called Decision Helper (which you can find in Chapter 55 on the CD-ROM). The application consists of three frames (see Figure 11-3). The top-left frame is one image that has four graphical buttons in it. The goal is to turn that image into a client-side image map and script it so the pages change in the right-hand and bottom frames. In the upper-right frame, the script loads an entirely different document along the sequence of five different documents that go in there. In the bottom frame, the script navigates to one of five anchors to display the segment of instructions that applies to the document loaded in the upper-right frame.

Listing 11-1 shows a slightly modified version of the actual file for the Decision Helper application's navigation frame. The listing contains a couple of new objects and concepts not yet covered in this tutorial. But as you will see, they are extensions to what you already know about JavaScript and objects. To help simplify the discussion here, I remove the scripting and HTML for the top and bottom button of the area map. In addition, I cover only the two navigation arrows.

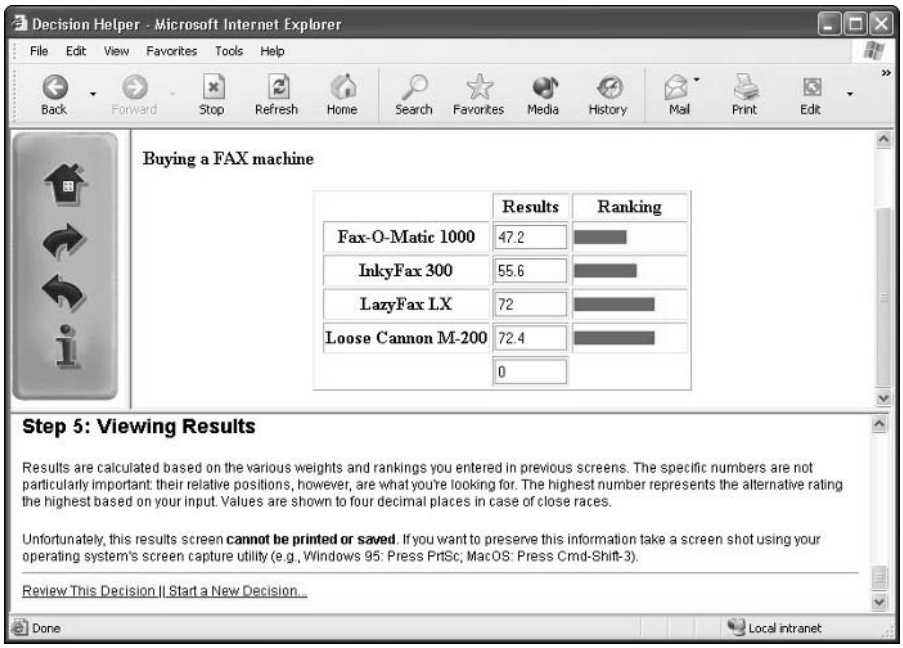


Figure 11-3: The Decision Helper screen.

Listing 11-1: **A Graphical Navigation Bar**

```
<html>
<head>
<title>Navigation Bar</title>
<script type="text/javascript">
<!-- start
function goNext() {
   var currOffset = parseInt(parent.currTitle);
   if (currOffset < 5) {
      currOffset += 1;
      parent.entryForms.location.href = "dh" + currOffset + ".htm";
      parent.instructions.location.hash = "help" + currOffset;
   } else {
      alert("This is the last form.");
   }
}
function goPrev() {
   var currOffset = parseInt(parent.currTitle);
   if (currOffset > 1) {
      currOffset -= 1;
      parent.entryForms.location.href = "dh" + currOffset + ".htm";
      parent.instructions.location.hash = "help" + currOffset;
   } else {
      alert("This is the first form.");
   }
}
// end -->
</script>
</head>
<body bgColor="white">
<map name="navigation">
<area shape="rect" coords="25,80,66,116" href="javascript:goNext()">
<area shape="rect" coords="24,125,67,161" href="javascript:goPrev()">
</map>
<img src="dhNav.gif" height="240" width="96" border="0" usemap="#navigation">
</body>
</html>
```

Look first at the HTML section for the Body portion. Almost everything there is standard stuff for defining client-side image maps. The coordinates define rectangles around each of the arrows in the larger image. The `href` attributes for the two areas point to JavaScript functions defined in the Head portion of the document (the `javascript:` pseudo-URL is covered in the next chapter).

In the frameset that defines the Decision Helper application, names are assigned to each frame. The upper-right frame is called `entryForms`; the bottom frame is called `instructions`.

Knowing that navigation from page to page in the upper-right frame requires knowledge of which page is currently loaded there, I build some other scripting into both the parent document and each of the documents that loads into that frame. A global variable called `currTitle` is defined in the parent document. Its value is an integer indicating which page of the sequence (1 through 5) is currently loaded. An `onload` event handler in each of the five documents (named `dh1.htm`, `dh2.htm`, `dh3.htm`, `dh4.htm`, and `dh5.htm`) assigns its page number to that parent global variable. This arrangement allows all frames in the frameset to share that value easily.

When a user clicks the right-facing arrow to move to the next page, the `goNext()` function is called. The first statement gets the `currTitle` value from the parent window and assigns it to a local variable: `currOffset`. An `if...else` construction tests whether the current page number is less than five. If so, the add-by-value operator adds one to the local variable so I can use that value in the next two statements.

In those next two statements, I adjust the content of the two right frames. Using the `parent` reference to gain access to both frames, I set the `location.href` property of the top-right frame to the name of the file next in line (by concatenating the number with the surrounding parts of the filename). The second statement sets the `location.hash` property (which controls the anchor being navigated to) to the corresponding anchor in the `instructions` frame (anchor names `help1`, `help2`, `help3`, `help4`, and `help5`).

A click of the left-facing arrow reverses the process, subtracting 1 from the current page number (using the subtract-by-value operator) and changing the same frames accordingly.

The example shown in Listing 11-1 is one of many ways to script a navigation frame in JavaScript. Whatever methodology you use, much interaction occurs among the frames in the frameset.

References for Multiple Windows

In Chapter 8, you saw how to create a new window and communicate with it by way of the `window` object reference returned from the `window.open()` method. In this section, I show you how one of those subwindows can communicate with objects, functions, and variables in the window or frame that creates the subwindow.

Every `window` object has a property called `opener`. This property contains a reference to the window or frame that held the script whose `window.open()` statement generated the subwindow. For the main browser window and frames therein, this value is `null`. Because the `opener` property is a valid window reference, you can use it to begin the reference to items in the original window — just like a script in a child frame uses `parent` to access items in the parent document. The parent-child terminology doesn't apply to subwindows, however.

Listings 11-2 and 11-3 contain documents that work together in separate windows. Listing 11-2 displays a button that opens a smaller window and loads Listing 11-3 into it. The main window document also contains a text field that gets filled in when you enter text into a corresponding field in the subwindow.

In the main window document, the `newWindow()` function generates the new window. Because no other statements in the document require the reference to the new window just opened, the statement does not assign its returned value to any variable. This is an acceptable practice in JavaScript if you don't need the returned value of a function or method.

Listing 11-2: **A Main Window Document**

```
<html>
<head>
<title>Main Document</title>
<script type="text/javascript">
function newWindow() {
   window.open("subwind.htm","sub","height=200,width=200");
}
</script>
</head>

<body>
<form>
<input type="button" value="New Window" onclick="newWindow()">
<br>
Text incoming from subwindow:
<input type="text" name="entry">
</form>
</body>
</html>
```

All of the action in the subwindow document comes in the `onchange` event handler of the text field. It assigns the subwindow field's own value to the value of the field in the opener window's document. Remember that the contents of each window and frame belong to a document. So even after your reference targets a specific window or frame, the reference must continue helping the browser find the ultimate destination, which is generally some element of the document.

Listing 11-3: **A Subwindow Document**

```
<html>
<head>
<title>A SubDocument</title>
</head>
<body>
<form onsubmit="return false">
Enter text to be copied to the main window:
<input type="text"
onchange="opener.document.forms[0].entry.value = this.value">
</form>
</body>
</html>
```

Just one more lesson to go before I let you explore all the details elsewhere in the book. I use the final tutorial chapter to show you some fun things you can do with your Web pages, such as changing images when the user rolls the mouse atop a picture.

Exercises

Before answering the first three questions, study the structure of the following frameset for a Web site that lists college courses:

```
<frameset rows="85%,15%">
   <frameset cols="20%,80%">
      <frame name="mechanics" src="history101M.html">
      <frame name="description" src="history101D.html">
   </frameset>
   <frameset cols="100%">
      <frame name="navigation" src="navigator.html">
   </frameset>
</frameset>
</html>
```

1. Each document that loads into the description frame has an `onload` event handler in its `<body>` tag that stores a course identifier in the framesetting document's global variable called `currCourse`. Write the `onload` event handler that sets this value to `"history101"`.

2. Draw a block diagram that describes the hierarchy of the windows and frames represented in the frameset definition.

3. Write the JavaScript statements located in the navigation frame that loads the file `"french201M.html"` into the mechanics frame and the file `"french201D.html"` into the description frame.

4. While a frameset is still loading, a JavaScript error message suddenly appears saying that "window.document.navigation.form.selector is undefined." What do you think is happening in the application's scripts, and how can you solve the problem?

5. A script in a child frame of the main window uses `window.open()` to generate a second window. How can a script in the second window access the `location` object (URL) of the top (framesetting) window in the main browser window?

✦ ✦ ✦

CHAPTER 12

Images and Dynamic HTML

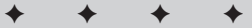
In This Chapter

How to precache images

How to swap images for mouse rollovers

Assigning scripts as tag attribute URLs

Modifying Body content dynamically



The previous eight lessons have been intensive, covering a lot of ground for both programming concepts and JavaScript. Now it's time to apply those fundamentals to the learning of more advanced techniques. I cover two areas here. First, I show you how to implement the ever-popular *mouse rollover* in which images swap when the user rolls the cursor around the screen. Then I introduce you to techniques for modifying a page's content after the page has loaded.

The Image Object

One of the objects contained by the document is the image object—supported in all scriptable browsers since the days of NN3 and IE4. Image object references for a document are stored in the object model as an array belonging to the `document` object. You can therefore reference an image by array index or image name. Moreover, the array index can be a string version of the image's name. Thus, all of the following are valid references to an image object:

```
document.images[n]
document.images["imageName"]
document.imageName
```

If your goal is to support scriptable images for browsers such as NN3 and NN4, you must be aware of image object limitations for those browsers. In particular, the range of scriptable properties is limited, although the all-important `src` property is accessible. Also, no mouse-related event handlers are affiliated with the image object (until you get to IE4+ and NN6+). If you want to make an image a clickable item in older browsers, surround it with a link (and set the image's border to zero) or attach a client-side image map to it. The combination of a link and image is how you make a backward-compatible clickable image button.

Interchangeable images

The advantage of having a scriptable image object is that a script can change the image occupying the rectangular space already occupied by an image. In IE4+ and NN6+, the images can even change size, with surrounding content automatically reflowing around the resized image.

The script behind this kind of image change is simple enough. All it entails is assigning a new URL to the image object's `src` property. The size of the image on the page is governed by the `height` and `width` attributes set in the `<img>` tag as the page loads. The most common image rollovers use the same size image for each of the rollover states.

Precaching images

Images take extra time to download from a Web server until the images are stored in the browser's cache. If you design your page so that an image changes in response to user action, you usually want the same fast response that users are accustomed to in other programs. Making the user wait seconds for an image to change can severely detract from enjoyment of the page.

JavaScript comes to the rescue by enabling scripts to load images into the browser's memory cache without displaying the image, a technique called *precaching images*. The tactic that works best is to preload the image into the browser's image cache while the page initially loads. Users are less impatient for those few extra seconds as the main page loads than waiting for an image to download in response to some mouse action.

Precaching an image requires constructing an image object in memory. An image object created in memory differs in some respects from the document image object that you create with the `<img>` tag. Memory-only objects are created by script, and you don't see them on the page at all. But their presence in the document code forces the browser to load the images as the page loads. The object model provides an `Image` object constructor function to create the memory type of image object as follows:

```
var myImage = new Image(width, height);
```

Parameters to the constructor function are the pixel width and height of the image. These dimensions should match the `<img>` tag's `width` and `height` attributes. Once the image object exists in memory, you can then assign a filename or URL to the `src` property of that image object:

```
myImage.src = "someArt.gif";
```

When the browser encounters a statement assigning a URL to an image object's `src` property, the browser fetches and loads that image into the image cache. All the user sees is some extra loading information in the status bar, as if another image were in the page. By the time the entire page loads, all images generated in this way are tucked away in the image cache. You can then assign your cached image's `src` property or the actual image URL to the `src` property of the document image created with the `<img>` tag:

```
document.images[0].src = myImage.src;
```

The change to the image in the document is instantaneous.

Listing 12-1 is a simple listing for a page that has one `<img>` tag and a select list that enables you to replace the image in the document with any of four precached images (including the original image specified for the tag). If you type this listing — as I strongly recommend — you can obtain copies of the four image files from the companion CD-ROM in the Chapter 12 directory of listings (you must still type the HTML and code, however).

Listing 12-1: **Precaching Images**

```
<html>
<head>
<title>Image Object</title>
<script type="text/javascript">
// initialize empty array
var imageLibrary = new Array();
// pre-cache four images
imageLibrary["image1"] = new Image(120,90);
imageLibrary["image1"].src = "desk1.gif";
imageLibrary["image2"] = new Image(120,90);
imageLibrary["image2"].src = "desk2.gif";
imageLibrary["image3"] = new Image(120,90);
imageLibrary["image3"].src = "desk3.gif";
imageLibrary["image4"] = new Image(120,90);
imageLibrary["image4"].src = "desk4.gif";

// load an image chosen from select list
function loadCached(list) {
   var img = list.options[list.selectedIndex].value;
   document.thumbnail.src = imageLibrary[img].src;
}
</script>
</head>

<body >
<h2>Image Object</h2>
<img src="desk1.gif" name="thumbnail" height="90" width="120">
<form>
<select name="cached" onchange="loadCached(this)">
<option value="image1">Bands
<option value="image2">Clips
<option value="image3">Lamp
<option value="image4">Erasers
</select>
</form>
</body>
</html>
```

As the page loads, it executes several statements immediately. These statements create an empty array that is populated with four new memory image objects. Each image object has a filename assigned to its `src` property. These images are loaded into the image cache as the page loads. Down in the Body portion of the document, an `<img>` tag stakes its turf on the page and loads one of the images as a starting image.

A `select` element lists user-friendly names for the pictures while housing the names of image objects already precached in memory. When the user makes a selection from the list, the `loadCached()` function extracts the selected item's value — which is a string index of the image within the `imageLibrary` array. The `src` property of the chosen image object is assigned to the `src` property of the visible image object on the page, and the precached image appears instantaneously.

Creating image rollovers

A favorite technique to add some pseudo-excitement to a page is to swap button images as the user rolls the cursor atop them. The degree of change to the image is largely a matter of taste. The effect can be subtle — a slight highlight or glow around the edge of the original image — or drastic — a radical change of color. Whatever your approach, the scripting is the same.

When several of these graphical buttons occur in a group, I tend to organize the memory image objects as arrays and create naming and numbering schemes that facilitate working with the arrays. Listing 12-2 shows such an arrangement for four buttons that control a jukebox. The code in the listing is confined to the image-swapping portion of the application. This is the most complex and lengthiest listing of the tutorial, so it requires a bit of explanation as it goes along.

Listing 12-2: **Image Rollovers**

```
<html>
<head>
<title>Jukebox/Image Rollovers</title>
<script type="text/javascript">
```

Only browsers capable of handling image objects should execute statements that precache images. Therefore, the entire sequence is nested inside an `if` construction that tests for the presence of the `document.images` array. In older browsers, the condition evaluates to "undefined," which an `if` condition treats as `false`.

```
if (document.images) {
```

Image precaching starts by building two arrays of image objects. One array stores information about the images depicting the graphical button's "off" position; the other is for images depicting their "on" position. These arrays use strings (instead of integers) as index values. The string names correspond to the names given to the visible image objects whose tags come later in the source code. The code is clearer to read (for example, you know that the `offImgArray["play"]` entry has to do with the Play button image). Also, as you see later in this listing, rollover images don't conflict with other visible images on the page (a possibility if you rely exclusively on numeric index values when referring to the visible images for the swapping).

After creating the array and assigning new blank image objects to the first four elements of the array, I go through the array again, this time assigning file pathnames to the `src` property of each object stored in the array. These lines of code execute as the page loads, forcing the images to load into the image cache along the way.

```
    // precache all 'off' button images
    var offImgArray = new Array();
    offImgArray["play"] = new Image(75,33);
    offImgArray["stop"] = new Image(75,33);
    offImgArray["pause"] = new Image(75,33);
    offImgArray["rewind"] = new Image(86,33);
```

```
   // off image array -- set 'off' image path for each button
   offImgArray["play"].src = "images/playoff.jpg";
   offImgArray["stop"].src = "images/stopoff.jpg";
   offImgArray["pause"].src = "images/pauseoff.jpg";
   offImgArray["rewind"].src = "images/rewindoff.jpg";

   // precache all 'on' button images
   var onImgArray = new Array();
   onImgArray["play"] = new Image(75,33);
   onImgArray["stop"] = new Image(75,33);
   onImgArray["pause"] = new Image(75,33);
   onImgArray["rewind"] = new Image(86,33);

   // on image array -- set 'on' image path for each button
   onImgArray["play"].src = "images/playon.jpg";
   onImgArray["stop"].src = "images/stopon.jpg";
   onImgArray["pause"].src = "images/pauseon.jpg";
   onImgArray["rewind"].src = "images/rewindon.jpg";
}
```

As you can see in the following HTML, when the user rolls the mouse atop any of the visible document image objects, the `onmouseover` event handler (from the link object surrounding the image in the document) invokes the `imageOn()` function, passing the name of the particular image. The `imageOn()` function uses that name to synchronize the `document.images` array entry (the visible image) with the entry of the in-memory array of "on" images from the `onImgArray` array. The `src` property of the array entry is assigned to the corresponding document image `src` property.

```
// functions that swap images & status bar
function imageOn(imgName) {
   if (document.images) {
      document.images[imgName].src = onImgArray[imgName].src;
   }
}
```

The same goes for the `onmouseout` event handler, which needs to turn the image off by invoking the `imageOff()` function with the same index value.

```
function imageOff(imgName) {
   if (document.images) {
      document.images[imgName].src = offImgArray[imgName].src;
   }
}
```

Both the `onmouseover` and `onmouseout` event handlers set the status bar to prevent the ugly `javascript:` URL (described later) from appearing there as the user rolls the mouse atop the image. The `onmouseout` event handler sets the status bar message to an empty string.

```
function setMsg(msg) {
   window.status = msg;
   return true;
}
```

For this demonstration, I disable the functions that control the jukebox. But I leave the empty function definitions here so they catch the calls made by the clicks of the links associated with the images.

```
// controller functions (disabled)
function playIt() {
}
function stopIt() {
}
function pauseIt(){
}
function rewindIt() {
}
</script>
</head>

<body>
<center>
<form>
Jukebox Controls<br>
```

I surround each image in the document with a link because the link object has the event handlers needed to respond to the mouse rolling over the area for compatibility back to NN3. Each link's `onmouseover` event handler calls the `imageOn()` function, passing the name of the image object to be swapped. Because both the `onmouseover` and `onmouseout` event handlers require a `return true` statement to work in older browsers, I combine the second function call (`to setMsg()`) with the `return true` requirement. The `setMsg()` function always returns `true` and is combined with the `return` keyword before the call to the `setMsg()` function. It's just a trick to reduce the amount of code in these event handlers.

If you are typing this listing to try it out, be sure to keep each entire `<a>` tag and its attributes in one unbroken line; or insert a carriage return *before* any event handler name.

```
<a href="javascript:playIt()"
   onmouseover="imageOn('play'); return setMsg('Play the selected tune')"
   onmouseout="imageOff('play'); return setMsg('')">
<img src="images/playoff.jpg" name="play" height="33" width="75" border="0">
</a>
<a href="javascript:stopIt()"
   onmouseover="imageOn('stop'); return setMsg('Stop the playing tune')"
   onmouseout="imageOff('stop'); return setMsg('')">
<img src="images/stopoff.jpg" name="stop" height="33" width="75" border="0">
</a>
<a href="javascript:pauseIt()"
   onmouseover="imageOn('pause'); return setMsg('Pause the playing tune')"
   onmouseout="imageOff('pause'); return setMsg('')">
<img src="images/pauseoff.jpg" name="pause" height="33" width="75" border="0">
</a>
<a href="javascript:rewindIt()"
   onmouseover="imageOn('rewind'); return setMsg('Rewind back to the
beginning')"
   onmouseout="imageOff('rewind'); return setMsg('')">
<img src="images/rewindoff.jpg" name="rewind" height="33" width="86" border="0">
</a>
</form>
</center>
</body>
</html>
```

You can see the results of this lengthy script in Figure 12-1. As the user rolls the mouse atop one of the images, it changes from a light to dark color by swapping the entire image. You can access the image files on the CD-ROM, and I encourage you to enter this lengthy listing and see the magic for yourself.

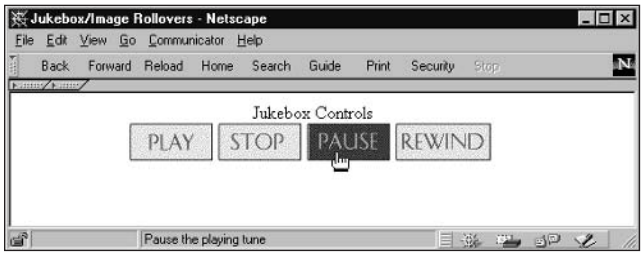


Figure 12-1: Typical mouse rollover image swapping.

The javascript: Pseudo-URL

You have seen several instances in this and previous chapters of applying what is called the `javascript:` pseudo-URL to `href` attributes of `<a>` and `<area>` tags. This technique should be used sparingly at best, especially for public Web sites that may be accessed by users with non-scriptable browsers (for whom the links will be inactive).

The technique was implemented to supplement the `onclick` event handler of objects that act as hyperlinks. Especially in the early scripting days, when elements such as images had no event handlers of their own, hyperlinked elements surrounding those inactive elements allowed users to appear to interact directly with elements such as images. When the intended action was to invoke a script function (rather than navigate to another URL, as is usually the case with a hyperlink), the language designers invented the `javascript:` protocol for use in assignments to the `href` attributes of hyperlink elements (instead of leaving the required attribute empty).

When a scriptable browser encounters an `href` attribute pointing to a `javascript:` pseudo-URL, the browser executes the script content after the colon when the user clicks on the element. For example, near the end of Listing 12-2, all four `<a>` tags point to `javascript:` pseudo-URLs that invoke script functions on the page, such as

```
<a href="javascript:playIt()" ... >
```

Be aware that unless you override the status bar text with mouse event handlers (as shown in Listing 12-2), the pseudo-URL appears in the status bar for the user to see (and perhaps be frightened). More importantly, however, remember that this URL is to be used only for assignment to `href` attributes. Do not use it with event handlers.

Popular Dynamic HTML Techniques

Because today's scriptable browsers uniformly permit scripts to access each element of the document and automatically reflow the page's content when anything changes, a much

higher degree of dynamism is possible in your applications. Dynamic HTML is a very deep subject, with lots of browser-specific peculiarities. In this final section of the tutorial, you will learn techniques that work in all W3C DOM-compatible browsers. I'll focus on two of the most common tasks for which DHTML is used: changing element styles and modifying Body content.

Changing stylesheet settings

Each element that renders on the page (and even some that don't) has a property called `style`. This property provides script access to all Cascading Style Sheet (CSS) properties supported for that element by the current browser. Property values are the same as those used for CSS specifications — frequently a different syntax than similar settings that used to be made by HTML tag attributes. For example, if you wish to set the text color of a quote element whose ID is `FranklinQuote`, the syntax is

```
document.getElementById("FranklinQuote").style.color = "rgb(255, 255, 0)";
```

Because the CSS `color` property accepts other ways of specifying colors (such as the traditional hexadecimal triplet — `#ffff00`), you may use those as well.

Some CSS property names, however, do not conform to JavaScript naming conventions. Several CSS property names contain hyphens. When that occurs, the scripted equivalent of the property compresses the words together and capitalizes the start of each word. For example, the CSS property `font-weight` would be set in script as follows:

```
document.getElementById("highlight").style.fontWeight = "bold";
```

A related technique puts more of the design burden on the CSS code. For example, if you define CSS rules for two different classes, you can simply switch the class definition being applied to the element by way of the element object's `className` property. For example, let's say you define two CSS class definitions with different background colors:

```
.normal {background-color: #ffffff}
.highlighted {background-color: #ff0000}
```

In the HTML page, the element first receives its default class assignment as follows:

```
<p id="news" class="normal">...</p>
```

A script statement can then change the class of that element object so that the highlighted style applies to it:

```
document.getElementById("news").className = "highlighted";
```

Restoring the original class name also restores its look and feel. This approach is also a quick way to change multiple style properties of an element with a single statement.

Dynamic content via W3C DOM nodes

In Chapter 8 you met the `document.createElement()` and `document.createTextNode()` methods. These methods create new DOM objects out of thin air, which you may then modify by setting properties (attributes) prior to plugging the new stuff into the document tree for all to see.

As an introduction to this technique, I'll demonstrate the steps you would go through to add an element and its text to a placeholding `span` element on the page. In this example, a

paragraph element belonging to a class called `centered` will be appended to a `span` whose ID is `placeholder`. Some of the text for the content of the paragraph comes from a text field in a form (the visitor's first name). Here is the complete sequence:

```
var newElem = document.createElement("p");
newElem.className = "centered";
var newText = document.createTextNode("Thanks for visiting, " +
   document.forms[0].firstName.value);
// insert text node into new paragraph
newElem.appendChild(newText);
// insert completed paragraph into placeholder
document.getElementById("placeholder").appendChild(newElem);
```

The W3C DOM approach takes a lot of tiny steps to create, assemble, and insert the pieces into their destinations. After creating the element and text nodes, the text node must be inserted into the element node. Because the new element node is empty when it is created, the DOM `appendChild()` method plugs the text node into the element (between its start and end tags, if you could see the tags). With the paragraph element assembled, it then gets inserted at the end of the initially empty `span` element. Additional W3C DOM methods (described in Chapters 15 and 16) provide more ways to insert, remove, and replace nodes.

Dynamic content via the innerHTML property

Prior to the W3C DOM specification, Microsoft invented a property of all element objects: `innerHTML`. This property first appeared in IE4, and became popular due to its practicality. The property's value is a string containing HTML tags and other content, just as it would appear in an HTML document inside the current element's tags. Even though the W3C DOM working group did not implement this property for the published standard, the property proved to be too practical and popular for modern browser makers to ignore. You can find it implemented as a de facto standard in Mozilla-based browsers and Safari, among others.

To show you the difference in the approach, the following code example shows the same content creation and insertion as shown in the previous W3C DOM section, but this time with the `innerHTML` property:

```
// accumulate HTML as a string
var newHTML = "<p class='centered'>Thanks for visiting, ";
newHTML += document.forms[0].firstName.value;
newHTML += "</p>";
// blast into placeholder element's content
document.getElementById("placeholder").innerHTML = newHTML;
```

While the `innerHTML` version seems more straightforward — and it makes it easier for HTML coders to visualize what's being added — the more code-intensive DOM node approach is more efficient when the Body modification task entails lots of content. Extensive string concatenation operations can slow down browser script processing. Sometimes the shortest script is not necessarily the fastest.

And so ends the final lesson of the *JavaScript Bible, Fifth Edition* tutorial. If you have gone through every lesson and tried your hand at the exercises, you are now ready to dive into the rest of the book to learn the fine details and many more features of both the document object model and the JavaScript language. You can work sequentially through the chapters of Parts III and IV, but before too long, you should also take a peek at Chapter 45 on the CD-ROM to learn some debugging techniques that help the learning process.

Exercises

1. Explain the difference between a document image object and the memory type of image object.

2. Write the JavaScript statements needed to precache an image file named `jane.jpg` that later will be used to replace the document image defined by the following HTML:

```
<img name="people" src="john.jpg" height="120" width="100"
alt="people">
```

3. With the help of the code you wrote for question 2, write the JavaScript statement that replaces the document image with the memory image.

4. Backward-compatible document image objects do not have event handlers for mouse events. How do you trigger scripts needed to swap images for mouse rollovers?

5. Assume that a `table` element contains an empty table cell (`td`) element whose ID is `forwardLink`. Using W3C DOM node creation techniques, write the sequence of script statements that create and insert the following hyperlink into the table cell:

```
<a href="page4.html">Next Page</a>
```

✦ ✦ ✦

P A R T

Document Objects Reference

In This Part

Chapter 13
JavaScript Essentials

Chapter 14
Document Object Model Essentials

Chapter 15
Generic HTML Element Objects

Chapter 16
Window and Frame Objects

Chapter 17
Location and History Objects

Chapter 18
The Document and Body Objects

Chapter 19
Link and Anchor Objects

Chapter 20
Image, Area, and Map Objects

Chapter 21
The Form and Related Objects

Chapter 22
Button Objects

Chapter 23
Text-Related Form Objects

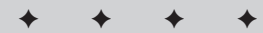
Chapter 24
Select, Option, and FileUpload Objects

Chapter 25
Event Objects

Chapter 26
Style Sheet and Style Objects

CHAPTER 13

JavaScript Essentials



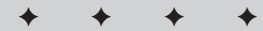
In This Chapter

How to separate the language from the document object model

Where scripts go in your documents

JavaScript language versions

Language highlights for experienced programmers



Whenever JavaScript is discussed in the context of the Web browser environment, it is sometimes difficult to distinguish between JavaScript the scripting language and the objects that you use the language to control. Even so, it's important to separate the language from the object model just enough to help you make important design decisions when considering JavaScript-enhanced pages. You may come to appreciate the separation in the future if you use JavaScript for other object models, such as server-side programming. All the basics of the language are identical. Only the objects differ.

This chapter elaborates on many of the fundamental subjects about the core JavaScript language raised throughout the tutorial (Part II), particularly as they relate to deploying scripts in a world in which visitors to your pages may use a wide variety of browsers. Along the way, you receive additional insights into the language itself. You can find details about the JavaScript core language syntax in Part IV.

JavaScript Versions

The JavaScript language has its own numbering system, which is completely independent of the version numbers assigned to browsers. The language's creator, Netscape, historically has had the most influence on the numbering system.

The first version, logically enough, was JavaScript 1.0. This was the version implemented in Navigator 2 and the first release of Internet Explorer 3. As the language evolved with succeeding browser versions, the JavaScript version number incremented in small steps. Internet Explorer 6 and Mozilla-based browsers support JavaScript 1.5.

Each successive generation of JavaScript employs additional language features. For example, in JavaScript 1.0, arrays were not developed fully, causing scripted arrays to not track the number of items in the array. JavaScript 1.1 filled that hole by providing a constructor function for generating arrays and an inherent `length` property for any generated array.

The JavaScript version implemented in a browser is not always a good predictor of core language features available for that browser. For example, while JavaScript 1.2 (as implemented by Netscape in NN4) included broad support for regular expressions, not all of those features appeared in Microsoft's corresponding JScript implementation in IE4. By the same token, Microsoft implemented `try-catch` error handling in its JScript in IE5, but Netscape didn't include that feature until the Mozilla-based NN6 implementation of JavaScript 1.5. Therefore, the language version number is far less important than the browser version in determining which language features are available for you to use.

Core Language Standard — ECMAScript

Although Netscape first developed the JavaScript language, Microsoft incorporated the language in Internet Explorer 3. Microsoft did not want to license the "Java" name from its trademark owner (Sun Microsystems), which is why the language became known in the IE environment as JScript. Except for some very esoteric exceptions and the pace of newly introduced features, the two languages are essentially identical. The levels of compatibility between browser brands for a comparable generation are remarkably high for the core language (unlike the vast disparities in object model implementations discussed in Chapter 14).

As mentioned in Chapter 2, standards efforts have been under way to create industry-wide recommendations for browser makers to follow (to make developers' lives easier). The core language was among the first components to achieve standard status. Through the European standards body called ECMA, a formal standard for the language has been agreed to and published. The first specification for the language, dubbed ECMAScript by the standards group, was roughly the same as JavaScript 1.1 in Netscape Navigator 3. The standard defines how various data types are treated, how operators work, what a particular data-specific syntax looks like, and other language characteristics. A newer version (called version 3) adds many enhancements to the core language (version 2 was version 1 with errata fixed). You can view the current version of the ECMA-262 specification at `http://www.ecma-international.org/`. If you are a student of programming languages, you will find the document fascinating; if you simply want to script your pages, you will probably find the minutia mind-boggling.

Both Netscape and Microsoft have pledged to make their browsers compliant with the ECMA standard. The vast majority of the ECMAScript standard has appeared in Navigator since version 3 and Internet Explorer since version 4. And, as new features are added to the ECMA standard, they tend to find their way into newer browsers as well. The latest version of ECMAScript is version 3, which is supported in JavaScript 1.5 (Moz1) and JScript in IE6.

Version 4 of ECMAScript is currently in the works, along with comparable implementations of JavaScript (2.0) and JScript by The Mozilla Foundation and Microsoft, respectively.

Embedding Scripts in HTML Documents

Scriptable browsers offer several ways to include scripts or scripted elements in your HTML documents. Not all approaches are available in all versions of every browser, but you have sufficient flexibility starting with Navigator 3 and some versions of Internet Explorer 3. When you consider that the vast majority of computer users are now using at least version 4 browsers, it's safe to assume a core level of script support among Web users. Exceptions to this rule include users who have specifically turned off scripting in their browsers, some organizations that install browsers with scripting turned off, and users with physical disabilities who require specialized browsers.

<script> tags

The simplest and most compatible way to include script statements in an HTML document is inside a <script>. . .</script> tag set that specifies the scripting language via the type attribute. You can have any number of such tag sets in your document. For example, you can define some functions in the Head section to be called by event handlers in HTML tags within the Body section. Another tag set can reside within the Body to write part of the content of the page as the page loads. Place only script statements and comments between the start and end tags of the tag set. Do not place any HTML tags inside unless they are part of a string parameter to a document.write() statement that creates content for the page.

Every opening <script> tag should specify the type attribute. Because the <script> tag is a generic tag indicating that the contained statements are to be interpreted as executable script and not renderable HTML, the tag is designed to accommodate any scripting language the browser knows.

Specifying the language version

Browsers starting with IE5 and Moz1 support the type attribute of the <script> tag. This attribute accepts the type of a script as a MIME type. For example, the MIME type of JavaScript is specified as type="text/javascript". So, a <script> block for JavaScript is coded as follows:

```
<script type="text/javascript">...</script>
```

The type attribute is required for the <script> tag as of HTML 4. Earlier versions of HTML, and therefore earlier browsers, recognize the language="JavaScript" attribute setting as opposed to type. The language attribute allows the scripter to write for a specific minimum version of JavaScript or, in the case of Internet Explorer, other languages such as VBScript. For example, the JavaScript interpreter built into Navigator 3 knows the JavaScript 1.1 version of the language; Navigator 4 and Internet Explorer 4 include the JavaScript 1.2 version. For versions beyond the original JavaScript, you may specify the language version by appending the version number after the language name without any spaces, as in

```
<script language="JavaScript1.1">...</script>

<script language="JavaScript1.2">...</script>
```

It's important to note that the language attribute was deprecated in HTML 4, with the type attribute being the recommended way of specifying the scripting language for <script> tags. However, only IE5+ and W3C DOM-compatible browsers recognize this attribute, and JavaScript versions are not taken into account with this methodology. To be both backward compatible and forward looking, you can specify both the language and type attributes in your <script> tags because older browsers ignore the type attribute. Following is an example of how you might do this:

```
<script type="text/javascript" language="JavaScript 1.2">...</script>
```

<script for> tags

IE4+ browsers offer a variation on the <script> tag that binds a <script> tag's statements to a specific object and event generated by that object. In addition to the language specification, the tag's attributes must include for and event attributes (not part of the HTML 4.0 specification). The value assigned to the for attribute is a reference to the desired object. Most often, this is simply the identifier assigned to the object's id attribute (IE4+ enables you to reference an object by either *document.all.objectID* or just *objectID*). The event attribute is the event handler name that you wish the script to respond to. For example, if

you design a script to perform some action upon a `mousedown` event in a paragraph whose ID is `myParagraph`, the script statements are enclosed in the following tag set:

```
<script for="myParagraph" event="onmousedown" type="text/javascript">
...
</script>
```

Statements inside the tag set execute only upon the firing of the event. No function definitions are required.

This way of binding an object's event to a script means that there is no event handler defined in the element's tag. Therefore, it guarantees that only IE4 or later can carry out the script when the event occurs. But the tag and attributes contain a lot of source code overhead for each object's script, so this is not a technique that you should use for script statements that need to be called by multiple objects.

Also be aware that you cannot use this tag variation if non-IE or pre-IE4 browsers load the page. In such browsers, script statements execute as the page loads, which certainly causes script errors.

Hiding script statements from older browsers

The number of people using old Web browsers that don't support scripting languages has diminished considerably in the past few years. However, new devices, such as mobile phones and pocket-sized computers, often employ compact browsers that don't have built-in JavaScript interpreters.

Nonscriptable browsers do not know about the `<script>` tag. Normally, browsers ignore tags they don't understand. That's fine when a tag is just one line of HTML, but a `<script>` tag sets off any number of script statement lines in a document. Old and compact browsers don't know to expect a closing `</script>` tag. Therefore, their natural inclination is to render any lines they encounter after the opening `<script>` tag. Unfortunately, this places script code squarely in the document — surely to confuse anyone who sees such gibberish on the page.

You can, however, exercise a technique that tricks most nonscriptable browsers into ignoring the script statements: surround the script statements — inside the `<script>` tag set — with HTML comment markers. An HTML comment begins with the sequence `<!--` and ends with `-->`. Therefore, you should embed these comment sequences in your scripts according to the following format:

```
<script type="text/javascript">
<!--
script statements here
//-->
</script>
```

JavaScript interpreters know to ignore a line that begins with the HTML beginning comment sequence, but they need a little help with the ending sequence. The close of the HTML comment starts with a JavaScript comment sequence (`//`). This tells JavaScript to ignore the line; but a nonscriptable browser sees the ending HTML symbols and begins rendering the page with the next HTML tag or other text in the document. An older browser doesn't know what the `</script>` tag is, so the tag is ignored and rendering begins after that.

If you design your pages for public access, it's still a good idea to include these HTML comment lines in all your `<script>` tag sets. Make sure they go inside the tags, not outside. Also note that most of the script examples in this book do not include these comments for the sake of saving space in the listings.

Hiding scripts entirely?

It may be misleading to say that this HTML comment technique "hides" scripts from older browsers. In truth, the comments hide the scripts from being rendered by the browsers. The tags and script statements, however, are still downloaded to the browser and appear in the source code when viewed by the user.

A common wish among authors is to truly hide scripts from visitors to a page. Client-side JavaScript must be downloaded with the page and is, therefore, visible in the source view of pages. There are, of course, some tricks you can implement that may disguise client-side scripts from prying eyes. The most easily implemented technique is to let the downloaded page contain no visible elements, only scripts that assemble the page that the visitor sees. Source code for such a page is simply the HTML for the page. But that page is not interactive because no scripting is attached unless it is written as part of the page — defeating the goal of hiding scripts. Any scripted solution for disguising scripts is immediately defeatable by the user turning off scripting temporarily before downloading the page. All of your code is ready for source view.

If you are worried about other scripters "stealing" your scripts, your best protection is to include a copyright notification in your page's source code. Not only are your scripts visible to the world, but so, too, are a thief's scripts. This way you can easily see when someone lifts your scripts verbatim.

One other option for minimizing other people "borrowing" your JavaScript code is to use a JavaScript obfuscator, which is a special application that scrambles your code and makes it much harder to read and understand. The code still works fine but it is very hard to modify in any way. You would use an obfuscator just before placing your code online, making sure to keep the original version for making changes. One JavaScript obfuscator that has been available for several years is a shareware program called *JavaScript Scrambler* (`http://www.quadhead.de/`).

Script libraries (.js files)

If you do a lot of scripting or script a lot of pages for a complex Web application, you will certainly develop some functions and techniques that you can use for several pages. Rather than duplicate the code in all of those pages (and go through the nightmare of making changes to all copies for new features or bug fixes), you can create reusable script library files and link them to your pages.

Such an external script file contains nothing but JavaScript code — no `<script>` tags, no HTML. The script file you create must be a text-only file, but its filename must end with the two-character extension `.js`. To instruct the browser to load the external file at a particular point in your regular HTML file, you add an `src` attribute to the `<script>` tag as follows:

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

This kind of tag should go at the top of the document so it loads before any other in-document `<script>` tags load. If you load more than one external library, include a series of these tag sets at the top of the document.

Take notice of two features about this external script tag construction. First, the `<script>` `</script>` tag pair is required, even though nothing appears between them. You can mix `<script>` tag sets that specify external libraries with in-document scripts in the same document. Second, avoid putting other script statements between the start and end tags when the start tag contains an `src` attribute.

How you reference the source file in the `src` attribute depends on its physical location and your HTML coding style. In the preceding example, the `.js` file is assumed to reside in the same directory as the HTML file containing the tag. But if you want to refer to an absolute URL, the protocol for the file is `http://` (just like with an HTML file):

```
<script type="text/javascript" src="http://www.cool.com/hotscript.js"></script>
```

A very important prerequisite for using script libraries with your documents is that your Web server software must know how to map files with the `.js` extension to a MIME type of `application/x-javascript`. If you plan to deploy JavaScript in this manner, be sure to test a sample on your Web server beforehand and arrange for any necessary server configuration adjustments.

When a user views the source of a page that links in an external script library, code from the `.js` file does not appear in the window even though the browser treats the loaded script as part of the current document. However, the name or URL of the `.js` file is plainly visible (displayed exactly as it appears in your source code). Anyone can then turn off JavaScript in the browser and open that file (using the `http://` protocol) to view the `.js` file's source code. In other words, an external JavaScript source file is no more hidden from view than JavaScript embedded directly in an HTML file.

Browser Version Detection

Without question, the biggest challenge facing many client-side scripters is how to program an application that accommodates a wide variety of browser versions and brands, each one of which can bring its own quirks and bugs. Happy is the intranet developer who knows for a fact that the company has standardized its computers with a particular brand and version of browser. But that is a rarity, especially in light of the concept of the *extranet*—private corporate networks and applications that open up for access to the company's suppliers and customers.

Having dealt with this problem since the original scripted browser (NN2) had to work alongside a hoard of nonscriptable browsers, I have identified several paths that an application developer can follow. Unless you decide to be autocratic about browser requirements for using your site, you must make compromises in desired functionality or provide multiple paths in your Web site for two or more classes of browsers. In this section, I give you several ideas about how to approach development in a fragmented browser world.

While JavaScript support has stabilized to some degree when it comes to desktop Web browsers, the popularity of mobile phone and handheld Web browsers has complicated the matter; few compact Web browsers support JavaScript as of yet.

Is JavaScript on?

Very often, the first decision an application must make is whether the client accessing the site is JavaScript-enabled. Non-JavaScript-enabled browsers fall into two categories: a) JavaScript-capable browsers that have JavaScript turned off in the preferences; and b) browsers that have no built-in JavaScript interpreter.

Using the <noscript> tag

Except for some of the earliest releases of NN2, all JavaScript-capable browsers have a preferences setting to turn off JavaScript (and a separate one for Java). You should know that even

though JavaScript is turned on by default in most browsers, many institutional deployments turn it off when the browser is installed on client machines. The reasons behind this MIS deployment decision vary from scares about Java security violations incorrectly associated with JavaScript, valid JavaScript security concerns on some browser versions, and the fact that some firewalls try to filter JavaScript lines from incoming HTML streams.

All JavaScript-capable browsers include a set of `<noscript>. . .</noscript>` tags to balance the `<script>. . .</script>` tag set. If one of these browsers has JavaScript turned off, the `<script>` tag is ignored but the `<noscript>` tag is observed. As with the `<noframes>` tag, you can use the body of a `<noscript>` tag set to display HTML that lets users know JavaScript is turned off, and therefore the full benefit of the page isn't available unless they turn on JavaScript. Listing 13-1 shows a skeletal HTML page that uses these tags.

Listing 13-1: **Employing the <noscript> Tag**

```
<html>
<head>
   <title>Some Document</title>
   <script type="text/javascript">
      // script statements
   </script>
</head>

<body>
   <noscript><b>Your browser has JavaScript turned off.</b><br />
   You will experience a more enjoyable time at this Web site if you
   turn JavaScript on.
   <hr /></noscript>

   <h2>The body of your document.</h2>
</body>
</html>
```

You can display any standard HTML within the `<noscript>` tag set. An icon image is a colorful way to draw the user's attention to the special advice at the top of the page. If your document is designed to create content dynamically in one or more places in the document, you may have to include a `<noscript>` tag set after more than one `<script>` tag set to let users know what they're missing. Do not include the HTML comment tags that you use in hiding JavaScript statements from older browsers. Their presence inside the `<noscript>` tags prevents the HTML from rendering.

Other nonscriptable browsers

At this juncture, I must point out that newcomers to scripting frequently want to know what script to write to detect whether JavaScript is turned on. Because scripters are so ready to write a script to work around all situations, it takes some thought to realize that a non-JavaScript browser cannot execute such a script: If no JavaScript interpreter exists in the browser (or it is turned off), the script is ignored. I suppose that the existence of a JavaScript-accessible method for Java detection — the `navigator.javaEnabled()` method — promises a parallel method for JavaScript. But logic fails to deliver on that unspoken promise.

Another desire is to have JavaScript substitute document content when the browser is JavaScript-enabled. Only in IE4+ and W3C DOM-compatible browsers can a script replace regular HTML with scripted content. If you develop content that must be backward compatible with older browsers, remember that all HTML in a document appears in the browser window, while scripted content can be additive only.

You can use this additive scripting to create unusual effects when displaying different links and (with a caveat) body text for scriptable and nonscriptable browsers. Listing 13-2 shows a short document that uses HTML comment symbols to trick nonscriptable browsers into displaying a link to Netscape's Web site and two lines of text. A scriptable browser takes advantage of a behavior that allows only the nearest `<a>` tag to be associated with a closing `</a>` tag. Therefore, the Microsoft link isn't rendered at all, but the link to my Web site is. For the body text, the script assigns the same text color to a segment of HTML body text as the document's background. While the colored text is camouflaged in a scriptable browser (and some other text written to the document), the "hidden" text remains invisible in the document. HTML fans frown upon this kind of element spoofing, which will likely run afoul of HTML validators. However, it can be fun to play with.

Listing 13-2: **Rendering Different Content for Scriptable and Nonscriptable Browsers**

```
<html>
<head>
   <title></title>
</head>

<body bgcolor="#FFFFFF">
   <a href="http://www.microsoft.com">
   <script type="text/javascript">
   <!--
   document.writeln("<a href='http://www.dannyg.com'>")
   //-->
   </script> Where?</a>
   <hr />
   <script type="text/javascript">
   <!--
   document.write("Howdy from the script!<font color='#FFFFFF'>")
   //-->
   </script>If you can read this, JavaScript is not available.
   <script type="text/javascript">
   <!--
   document.write("<\/font>")
   //-->
   </script><br />
   Here's some stuff afterward.
</body>
</html>
```

Scripting for different browsers

A number of solutions exist for accommodating different client browsers because the specific compatibility need might be as simple as letting a link navigate to a scripted page for

script-enabled browsers, as involved as setting up distinct areas of your application for different browser classes, or any degree in between. The first step in planning for compatibility is determining what your goals are for various visitor classes.

Establishing goals

Once you map out your application, you must then look at the implementation details to see which browser is required for the most advanced aspect of the application. For example, if the design calls for image swapping on mouse rollovers, that feature requires NN3+ and IE4+, which is a relatively safe assumption these days. In implementing Dynamic HTML features, you have potentially three different ways to implement tricks (such as movable elements or changeable content) because the document object models require different scripting (and sometimes HTML) for NN4, IE4+, and the W3C DOM implemented in Moz1+, IE5+, Safari, and other recent browsers.

In an ideal scenario, you have an appreciation for the kinds of browsers that your visitors use. For example, if you want to implement some DHTML features, but NN4 usage is only a small and decreasing percentage of hits, you can probably get by with designing for the IE4+ and W3C DOM. Or you may wish to forget the past and design your DHTML exclusively for W3C DOM-compatible browsers. If your Web hosting service maintains a log of visitor activity to your site, you can study the browsers listed among the hits to see which browsers your visitors use.

After you determine the lowest common denominator for the optimum experience, you then must decide how gracefully you want to degrade the application for visitors whose browsers do not meet the common denominator. For example, if you plan a page or site that requires a W3C DOM-compatible browser for all the bells and whistles, you can provide an escape path with content in a simple format that every browser from Lynx to IE4 and NN4 can view. You might even provide for users of handheld devices a third offering with limited or no scriptability that is designed specifically for a constrained user interface.

Creating an application or site that has multiple paths for viewing the same content may sound good at the outset, but don't forget that maintenance chores lie ahead as the site evolves. Will you have the time, budget, and inclination to keep all paths up to date? Despite whatever good intentions a designer of a new Web site may have, in my experience the likelihood that a site will be maintained properly diminishes rapidly with the complexity of the maintenance task.

Implementing a branching index page

If you decide to offer two or more paths into your application or content, one place you can start visitors down their individual paths is at the default page for your site. Numerous techniques are available that can redirect visitors to the appropriate perceived starting point of the site.

One design to avoid is placing the decision about the navigation path in the hands of the visitor. Offering buttons or links that describe the browser requirements may work for users who are HTML and browser geeks, but average consumers surfing the Web these days likely don't have a clue about what level of HTML their browsers support or whether they are JavaScript-enabled. It is incumbent upon the index page designer to automate the navigation task as much as possible.

A *branching index page* has almost no content. It is not the "home page" per se of the site, but rather a gateway to the entire Web site. Its job is to redirect users to what appears to be the home page for the site. Listing 13-3 shows what such a branching index page looks like.

Listing 13-3: **A Branching Index Page**

```
<html>
<head>
   <title>GiantCo On The Web</title>
   <script type="text/javascript">
   <!--
   window.location = "home1.html"
   //-->
   </script>
   <meta http-equiv="REFRESH" content=
   "0; URL=http://www.giantco.com/home2.html">
</head>

<body>
   <center>
      <a href="home2.html"><img src="images/giantcoLogo.gif" height=
      "60" width="120" border="0" alt="Go To GiantCo Home Page" /></a>
   </center>
</body>
</html>
```

Notice that the only visible content is an image surrounded by a standard link. The `<body>` tag contains no background color or art. A single script statement is located in the Head. A `<meta>` tag is also in the Head to automate navigation for some users. To see how a variety of browsers respond to this page, here are what three different classes of browser do with Listing 13-3:

- **A JavaScript-enabled browser.** Although the entire page may load momentarily (at most, flashing the company logo for a brief moment), the browser executes the script statement that loads `home1.html` into the window. In the meantime, the image is preloaded into the browser's memory cache. This image should be reused in `home1.html` so the download time isn't wasted on a one-time image. If your pages require a specific browser brand or minimum version number, this is the place to filter out browsers that don't meet the criteria (which may include the installation of a particular plug-in). Use the properties of the `navigator` object (Chapter 38 on the CD-ROM) to write a *browser sniffer script* that allows only those browsers meeting your design minimum to navigate to the scripted home page. All other browsers fall through to the next execution possibility.
- **A modern browser with JavaScript turned off or missing.** Several modern browsers recognize the special format of the `<meta>` tag as one that loads a URL into the current window after a stated number of seconds. In Listing 13-3, that interval is zero seconds. The `<meta>` tag is executed only if the browser ignores the `<script>` tag. Therefore, any scriptable browser that has JavaScript turned off or any browser that knows `<meta>` tags but no scripting follows the refresh command for the `<meta>` tag. If you utilize this tag, be very careful to observe the tricky formatting of the `content` attribute value. A semicolon and the subattribute `url` follow the number of seconds. A complete URL for your non-scriptable home page version is required for this subattribute. Importantly, the entire `content` attribute value is inside one set of quotes.
- **Older graphical browsers, compact PDA browsers, and Lynx.** The last category includes graphical browsers with limited capabilities, as well as intentionally stripped-down browsers. Lynx is designed to work in a text-only VT-100 terminal screen; mobile phones, personal digital assistants (PDAs), and handheld computers have browsers

optimized for usage through relatively slow network connections and viewing on small screens. Numerous other browsers are designed to provide Web accessibility for users with disabilities through technologies such as speech synthesis and touch screens (see `http://www.w3.org/WAI`). If such browsers do not understand the `<meta>` tag for refreshing content, they land at this page with no further automatic processing. But by creating an image that acts as a link, the user will likely click (or tap) on it to continue. The link then leads to the nonscriptable home page. Also note that the `alt` attribute for the image is supplied. This takes care of Lynx and compact browsers (with image loading off) because these browsers show the `alt` attribute text in lieu of the image. Users click or tap on the text to navigate to the URL referenced in the link tag.

I have a good reason to keep the background of the branching index page plain. For those whose browsers automatically lead them to a content-filled home page, the browser window flashes from a set background color to the browser's default background color before the new home page and its background color appear. By keeping the initial content to only the company logo, less screen flashing and obvious navigation are visible to the user.

One link—alternate destinations

Another filtering technique is available directly from links. With the exceptions of NN2 and IE3, a link can navigate to one destination via a link's `onclick` event handler and to another via the `href` attribute if the browser is not scriptable.

The trick is to include an extra `return false` statement in the `onclick` event handler. This statement cancels the link action of the `href` attribute. For example, if a nonscriptable browser should go to one version of a page at the click of a link and the scriptable browser should go to another, the link tag is as follows:

```
<a href="nonJSCatalog.html" onclick="location.href='JSCatalog.html';return
false">Product Catalog</a>
```

Only nonscriptable browsers, NN2, and IE3 go to the `nonJSCatalog.html` page; all others go to the `JSCatalog.html` page.

Object detection

The final methodology for implementing browser version branching is known as *object detection*. The principle is simple: If an object type exists in the browser's object model, it is safe to execute script statements that work with that object.

Perhaps the best example of object detection is the way scripts can swap images on a page in newer browsers without tripping up on older browsers that don't implement images as objects. In a typical image swap, `onmouseover` and `onmouseout` event handlers (assigned to a link surrounding an image, to be backward compatible) invoke functions that change the `src` property of the desired image. Each of those functions is invoked for all scriptable browsers, but you want them to run their statements only when images can be treated as objects.

Object models that implement images always include an array of image objects belonging to the `document` object. The `document.images` array always exists, even with a length of zero when no images are on the page. Therefore, if you wrap the image swapping statements inside an `if` construction that lets browsers pass only if the `document.images` array exists, older browsers simply skip over the statements:

```
function imageSwap(imgName, url) {
   if (document.images) {
      document.images[imgName].src = url;
   }
}
```

Object detection works best when you know for sure how all browsers implement the object. In the case of `document.images`, the implementation across browsers is identical, so it is a very safe branching condition. That's not always the case, and you should use this feature with careful thought. For example, IE4 introduced a `document` object array called `document.all`, which is used very frequently in building references to HTML element objects. NN4, however, did not implement that array, but instead had a document-level array object called `layers`, which was not implemented in IE4. Unfortunately, many scripters used the existence of these array objects as determinants for browser version. They set global variables signifying a minimum version of IE4 if `document.all` existed and NN4 if `document.layers` existed. This is most dangerous because there is no way of knowing if a future version of a browser may adopt the object of the other browser brand or eliminate a language feature. For example, Opera in its native setting supports the `document.all` array. But if you expect that browser to support every detail of the IE4 browser, scripts will break left and right.

This is why I recommend object detection not for browser version sniffing but for object availability branching, as shown previously for images. Moreover, it is safest to implement object detection only when all major browser brands (and the W3C DOM recommendation) have adopted the object so that behavior is predictable wherever your page loads in the future.

Techniques for object detection include testing for the availability of an object's method. A reference to an object's method returns a value, so such a reference can be used in a conditional statement. For example, the following code fragment demonstrates how a function can receive an argument containing the string ID of an element and convert the string to a valid object reference for three different document object models:

```
function myFunc(elemID) {
   var obj;
   if (document.getElementById) {
      obj = document.getElementById(elemID);
   } else if (document.all) {
      obj = document.all(elemID);
   } else if (document.layers) {
      obj = document.layers[elemID];
   }
   if (obj) {
      // statements that work on the object
   }
}
```

With this object detection scheme, it no longer matters which browser brand, operating system, and version supports a particular way of changing an element ID to an object reference. Whichever of the three `document` object properties or method is supported by the browser (or the first one, if the browser supports more than one), that is the property or method used to accomplish the conversion. If the browser supports none of them, no further statements execute.

If your script wants to check for the existence of an object's property or method, you may also have to check for the existence of the object beforehand if that object is not part of all browers' object models. An attempt to reference a property of a non-existent object in a conditional expression generates a script error. To avoid the error, you can cascade the conditional tests with the help of the `&&` operator. The following fragment tests for the existence of both the `document.body` object and the `document.body.style` property:

```
if (document.body && document.body.style) {
   // statements that work on the body's style property
}
```

If the test for `document.body` fails, JavaScript bypasses the second test.

One potential "gotcha" to using conditional expressions to test for the existence of an object's property is that even if the property exists but its value is zero or an empty string, the conditional test reports that the property does not exist. To work around this potential problem, the conditional expression can examine the data type of the value to ensure that the property genuinely exists. A nonexistent property for an object reports a data type of `undefined`. Use the `typeof` operator (see Chapter 32) to test for a valid property:

```
if (document.body && typeof document.body.scroll != "undefined") {
   // statements that work on the body's scroll property
}
```

Object detection is the wave of the future, and I wholeheartedly recommend designing your scripts to take advantage of it in lieu of branching on particular browser name strings and version numbers. Scriptable features are gradually finding their way into browsers embedded in a wide range of non-traditional computing devices. These browsers may not go by the same names and numbering systems that we know today, yet such browsers may be able to interpret your scripts. By testing for browser functionality, your scripts will likely require less maintenance in the future. You can see more object detection at work in Chapters 47 and 56 on the CD-ROM.

Designing for Compatibility

Each new major release of a browser brings compatibility problems for page authors. It's not so much that old scripts break in the new versions (well-written scripts rarely break in new versions with the rare exception of the jump from NN4 to the new browser engine in Mozilla). No, the problems center on the new features that attract designers when the designers forget to accommodate visitors who have not yet advanced to the latest and greatest browser version or who don't share your browser brand preference.

Catering only to the lowest common denominator can more than double your development time due to the expanded testing matrix necessary to ensure a good working page in all operating systems and on all versions. Decide how important the scripted functionality you employ in a page is for every user. If you want some functionality that works only in a later browser, you may have to be a bit autocratic in defining the minimum browser for scripted access to your page — any lesser browser gets shunted to a simpler presentation of your site's data.

Another possibility is to make a portion of the site accessible to most, if not all, browsers, and restrict the scripting to only the occasional enhancement that nonscriptable browser users won't miss. Once the application reaches a certain point in the navigation flow, the user needs a more capable browser to get to the really good stuff. This kind of design is a carefully planned strategy that lets the site welcome all users up to a point, but then enables the application to shine for users of, say, W3C DOM-compatible browsers.

The ideal page is one that displays useful content on any browser, but whose scripting enhances the experience of the page visitor — perhaps by offering more efficient site navigation or interactivity with the page's content. That is certainly a worthy goal to aspire to. But even if you can achieve this ideal on only some pages, you will reduce the need for defining entirely separate, difficult-to-maintain paths for browsers of varying capabilities.

Regardless of your specific browser compatibility strategy, the good news is that time tends to minimize the problem. Web standards have solidified greatly in the past few years, and browser vendors are finally making a more serious effort to support those standards. Furthermore, as more of the Web community upgrades to modern browsers, the issue of supporting old browsers becomes less and less significant. Bottom line — there is light at the end of the tunnel.

Dealing with beta browsers

If you have crafted a skillfully scripted Web page or site, you may be concerned when a prerelease (or *beta*) version of a browser available to the public causes script errors or other compatibility problems to appear on your page. Do yourself a favor — don't overreact to bugs and errors that occur in prerelease browser versions. If your code is well written, it should work with any new generation of browser. If the code doesn't work correctly, consider the browser to be buggy. Report the bug (preferably with a simplified test case script sample) to the browser maker.

The exception to the "it's a beta bug" rule arose in the transition from NN4 to the new Mozilla engine (first released as NN6). As you learn in Chapter 14, a conscious effort to eliminate a proprietary NN4 feature (the `<layer>` tag and corresponding scriptable object) caused many NN4 scripts to break on Moz1 betas (and final release). Had scripters gone to report the problem to the new browsers' developer (Mozilla), they would have learned of the policy change, and planned for the new implementation. It is extremely rare for a browser to eliminate a popular feature so quickly, but it can happen.

It is often difficult to prevent yourself from getting caught up in browser makers' enthusiasm for a new release. But remember that a prerelease version is not a shipping version. Users who visit your page with prerelease browsers should know that there may be bugs in the browser. That your code does not work with a prerelease version is not a sin, nor is it worth losing sleep over. Just be sure to connect with the browser's maker either to find out if the problem will continue in the final release or to report the bug so the problem doesn't make it into the release version.

The Evaluator Sr.

In Chapter 6, you were introduced to a slimmed-down version of The Evaluator Jr., which provides an interactive workbench to experiment with expression evaluation and object inspection. At this point, you should meet The Evaluator Sr., a tool you will use in many succeeding chapters to help you learn both core JavaScript and DOM terminology.

Figure 13-1 shows the top part of the page. Two important features differentiate this full version from the Jr. version in Chapter 6.

IE Browser Version Headaches

As described more fully in the discussion of the `navigator` object in Chapter 38 on the CD-ROM, your scripts can easily determine which browser is the one running the script. However, the properties that reveal the version don't always tell the whole story about Internet Explorer.

As you can see in detail in Chapter 38 on the CD-ROM, the `navigator.appVersion` property for IE5, 5.5, and 6 for Windows reports version 4 (the same as IE4). You can still "sniff" for versions 5 and 6 (you can find the designation `MSIE 5` or `MSIE6` in the `navigator.userAgent` property), but the process is not as straightforward as it could be. The best advice is to be vigilant when new browsers come on the scene or adopt object detection techniques in your scripts.

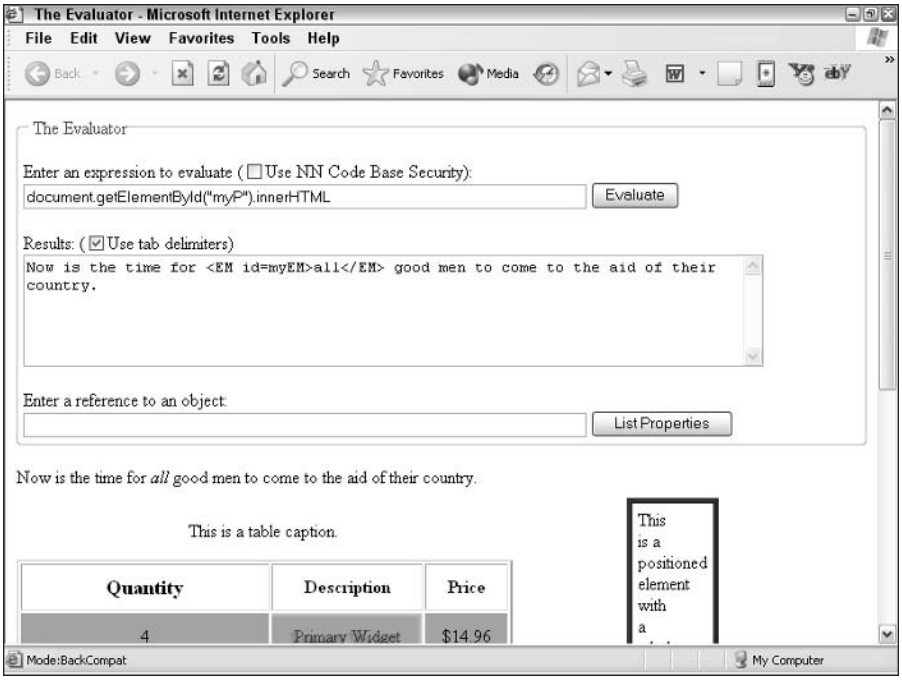


Figure 13-1: The Evaluator Sr.

First, you can try some Netscape/Mozilla secure features if you have Code Base Principles turned on for your browser (Chapter 46 on the CD-ROM) and you check the Use Code Base Security checkbox (NN4+ only). Second, the page has several HTML elements preinstalled, which you can use to explore DOM properties and methods. As with the smaller version, a set of 26 one-letter global variables (`a` through `z`) are initialized and ready for you to assign values for extended evaluation sequences.

You should copy the file `evaluator.html` from the companion CD-ROM to a local hard disk and set a bookmark for it in all of your test browsers. Feel free to add your own elements to the bottom of the page to explore other objects. I describe a version of The Evaluator for embedding in your projects as a debugging tool in Chapter 45 on the CD-ROM, where you can learn more built-in functionality of The Evaluator.

Compatibility ratings in reference chapters

With the proliferation of scriptable browser versions since Navigator 2, it is important to know up front whether a particular language or object model object, property, method, or event handler is supported in the lowest common denominator for which you are designing. Therefore, beginning with Chapter 15 of this reference part of the book, I include frequent compatibility ratings, such as the following example:

Compatibility: WinIE5+, MacIE5+, NN4+, Moz1.01+, Safari1+

A plus sign after a browser version number means that the language feature was first implemented in the numbered version and continues to be supported in succeeding versions. A minus sign means that the feature is not supported in that browser. The browsers tested for compatibility include Internet Explorer for Windows and Macintosh, Netscape Navigator, Mozilla (including all browsers based on the Mozilla engine), and Apple's Safari. I also recommend that you print the JavaScript and Browser Objects Quick Reference file shown in Appendix A. The file is on the companion CD-ROM in Adobe Acrobat format. This quick reference clearly shows each object's properties, methods, and event handlers, along with keys to the browser version in which each language item is supported. You should find the printout to be valuable as a day-to-day resource.

Language Essentials for Experienced Programmers

In this section, experienced programmers can read the highlights about the core JavaScript language in terms that may not make complete sense to those with limited or no scripting experience. This section is especially for you if you found the tutorial of Part II rudimentary. Here, then, is the quick tour of the essential issues surrounding the core JavaScript language:

- **JavaScript is a scripting language.** The language is intended for use in an existing *host environment* (for example, a Web browser) that exposes objects whose properties and behaviors are controllable via statements written in the language. Scripts execute within the context of the host environment. The host environment controls what, if any, external environmental objects may be addressed by language statements running in the host environment. For security and privacy reasons, Web browsers generally afford little or no direct access via JavaScript to browser preferences, the operating system, or other programs beyond the scope of the browser. The exception to this rule is that modern browsers allow deeper client access (with the user's permission) through trust mechanisms such as signed scripts (Netscape) or trusted ActiveX controls (Microsoft).
- **JavaScript is object-based.** Although JavaScript exhibits many syntactic parallels with the Java language, JavaScript is not as pervasively object-oriented as Java. The core language includes several built-in static objects from which working objects are generated. Objects are created via a call to a constructor function for any of the built-in objects plus the `new` operator. For example, the following expression generates a `String` object and returns a reference to that object:

  ```
  new String("Hello");
  ```

Table 13-1 lists the built-in objects with which scripters come in contact.

Table 13-1: JavaScript Built-In Objects

Array[1]	Boolean	Date	Error[2]
EvalError[2]	Function[1]	Math	Number[1]
Object[1]	RangeError[2]	ReferenceError[2]	RegExp[3]
String[1]	SyntaxError[2]	TypeError[2]	URIError[2]

[1]Although defined in ECMA Level 1, was first available in NN3 and IE3/J2.

[2]Defined in ECMA Level 3; implemented in Moz1.

[3]Defined in ECMA Level 3; implemented fully in NN4 and IE6.

- **JavaScript is loosely typed.** Variables, arrays, and function return values are not defined to be of any particular data type. In fact, an initialized variable can hold different data type values in subsequent script statements (obviously not good practice, but possible nonetheless). Similarly, an array may contain values of multiple types. The range of built-in data types is intentionally limited:
  - Boolean (`true` or `false`)
  - Null
  - Number (double-precision 64-bit format IEEE 734 value)
  - Object (encompassing the `Array` object)
  - String
  - Undefined
- **The host environment defines global scope.** Web browsers traditionally define a browser window or frame to be the global context for script statements. When a document unloads, all global variables defined by that document are destroyed.
- **JavaScript variables have either global or local scope.** A global variable in a Web browser is typically initialized in `var` statements that execute as the document loads. All statements in that document can read or write that global variable. A local variable is initialized inside a function (also with the `var` operator). Only statements inside that function may access that local variable.
- **Scripts sometimes access JavaScript static object properties and methods.** Some static objects encourage direct access to their properties or methods. For example, all properties of the `Math` object act as constant values (for example, `Math.PI`).
- **You can add properties or methods to working objects at will.** To add a property to an object, simply assign a value of any type to it. For example, to add an `author` property to a string object named `myText`, use:

  ```
  myText.author = "Jane";
  ```

  Assign a function reference to an object property to give that object a new method:

  ```
  // function definition
  function doSpecial(arg1) {
     // statements
  }
  // assign function reference to method name
  myObj.handleSpecial = doSpecial;
  ...
  // invoke method
  myObj.handleSpecial(argValue);
  ```

  Inside the function definition, the `this` keyword refers to the object that owns the method.
- **JavaScript objects employ prototype-based inheritance.** All object constructors create working objects whose properties and methods inherit the properties and methods defined for the *prototype* of that object. Starting with NN3 and late versions of IE3, scripts can add and delete custom properties and/or methods associated with the static object's prototype so that new working objects inherit the current state of the

prototype. Scripts can freely override prototype property values or assign different functions to prototype methods in a working object if desired without affecting the static object prototype. But if inherited properties or methods are not modified in the current working object, any changes to the static object's prototype are reflected in the working object. (The mechanism is that a reference to an object's property works its way up the prototype inheritance chain to find a match to the property name.)

- **JavaScript includes a large set of operators.** You can find most operators that you are accustomed to working with in other languages.
- **JavaScript provides typical control structures.** All versions of JavaScript offer `if`, `if-else`, `for`, and `while` constructions. JavaScript 1.2 (NN4+ and IE4+) added `do-while` and `switch` constructions. Iteration constructions provide `break` and `continue` statements to modify control structure execution.
- **JavaScript functions may or may not return a value.** There is only one kind of JavaScript function. A value is returned only if the function includes a `return` keyword followed by the value to be returned. Return values can be of any data type.
- **JavaScript functions cannot be overloaded.** A JavaScript function accepts zero or more arguments, regardless of the number of parameter variables defined for the function. All arguments are automatically assigned to the `arguments` array, which is a property of a function object. Parameter variable data types are not predefined.
- **Values are passed "by reference" and "by value."** An object passed to a function is actually a reference to that object, offering full read/write access to properties and methods of that object. But other types of values (including object properties) are passed by value, with no reference chain to the original object. Thus, the following nonsense fragment empties the text box when the `onchange` event fires:

```
function emptyMe(arg1) {
   arg1.value = "";
}
...
<input type="text" value="Howdy" onchange="emptyMe(this)">
```

  But in the following version, nothing happens to the text box:

```
function emptyMe(arg1) {
   arg1 = "";
}
...
<input type="text" value="Howdy" onchange="emptyMe(this.value)">
```

  The local variable (`arg1`) simply changes from `"Howdy"` to an empty string.

- **Error trapping techniques depend on JavaScript version.** There is no error trapping in NN2 or IE3. Error trapping in NN3, NN4, and IE4 is event-driven in the Web browser object model. JavaScript, as implemented in IE5+ and Moz1+, Safari, and other recent browsers, supports `try-catch` and `throw` statements, as well as built-in error objects that are not dependent on the host environment.
- **Memory management is not under script control.** The host environment manages memory allocation, including garbage collection. Different browsers may handle memory in different ways.
- **White space (other than a line terminator) is insignificant.** Space and tab characters may separate lexical units (for example, keywords, identifiers, and so on).

- **A line terminator is usually treated as a statement delimiter.** Except in very rare constructions, JavaScript parsers automatically insert the semicolon statement delimiter whenever they encounter one or more line terminators (for example, carriage returns or line feeds). A semicolon delimiter is required between two statements on the same physical line of source code. Moreover, string literals may not have carriage returns in their source code (but an escaped newline character (`\n`) may be a part of the string).

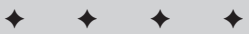
Onward to Object Models

The core language is only a small part of what you work with while scripting Web pages. The bulk of your job entails understanding the ins and outs of document object models as implemented in several generations of browsers. That's where the next chapter picks up the "essentials" story.

✦ ✦ ✦

CHAPTER 14

Document Object Model Essentials



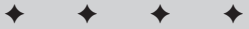
In This Chapter

Object models versus browser versions

Proprietary model extensions

Structure of the W3C DOM

Mixing object models in a single document



Without question, the biggest challenge facing client-side Web scripters is the sometimes-baffling array of document object models that have competed for our attention throughout the short history of scriptable browsers. Netscape got the ball rolling in Navigator 2 with the first object model. By the time the version 4 browsers came around, the original object model had gained not only some useful cross-browser features, but also a host of features that were unique to only Navigator or Internet Explorer. The object models were diverging, causing no end of headaches for page authors whose scripts had to run on as many browsers as possible. A ray of hope emerged from the standards process of the World Wide Web Consortium (W3C) in the form of a document object model (DOM) recommendation. The DOM brought forward much of the original object model, plus new ways of addressing every object in a document. The goal of this chapter is to put each of the object models into perspective and help you select the model(s) you intend to support in your Web applications. But before we get to those specifics, let's examine the role of the object model in designing scripted applications.

The Object Model Hierarchy

In the tutorial chapters of Part II, you were introduced to the fundamental ideas behind a document object hierarchy in scriptable browsers. In other object-oriented environments, object hierarchy plays a much greater role than it does in JavaScript-able browsers. (In JavaScript, you don't have to worry about related terms, such as classes, inheritance, and instances.) Even so, you cannot ignore the hierarchy concept because much of your code relies on your ability to write references to objects that depend on their positions within the hierarchy.

Calling these objects "JavaScript objects" is not entirely correct. These are really browser document objects: you just happen to use the JavaScript language to bring them to life. Some scripters of Microsoft Internet Explorer use the VBScript language to script the very same document objects. Technically speaking, JavaScript objects apply to data types and other core language objects separate from the document. The more you can keep document and core language objects separate in your head, the more quickly you can deal with browser brand compatibility issues.

Hierarchy as road map

For the programmer, the primary role of the document object hierarchy is to provide scripts with a way to reference a particular object among all the objects that a browser window can contain. The hierarchy acts as a road map the script can use to know precisely which object to address.

Consider, for a moment, a scene in which you and your friend Tony are in a high school classroom. It's getting hot and stuffy as the afternoon sun pours in through the wall of windows on the west side of the room. You say to Tony, "Would you please open a window?" and motion your head toward a particular window in the room. In programming terms, you've issued a command to an object (whether or not Tony appreciates the comparison). This human interaction has many advantages over anything you can do in programming. First, by making eye contact with Tony before you speak, he knows that he is the intended recipient of the command. Second, your body language passes along some parameters with that command, pointing ever so subtly to a particular window on a particular wall.

If, instead, you are in the principal's office using the public address system, and you broadcast the same command, "Would you please open a window?," no one knows what you mean. Issuing a command without directing it to an object is a waste of time because every object thinks, "That can't be meant for me." To accomplish the same goal as your one-on-one command, the broadcast command has to be something like, "Would Tony Jeffries in Room 312 please open the middle window on the west wall?"

Let's convert this last command to JavaScript *dot syntax* form (see Chapter 4). Recall from the tutorial that a reference to an object starts with the most global point of view and narrows to the most specific point of view. From the point of view of the principal's office, the location hierarchy of the target object is

```
room312.Jeffries.Tony
```

You can also say that Tony's knowledge about how to open a window is one of Tony's methods. The complete reference to Tony and his method then becomes

```
room312.Jeffries.Tony.openWindow()
```

Your job isn't complete yet. The method requires a parameter detailing which window to open. In this case, the window you want is the middle window of the west wall of Room 312. Or, from the hierarchical point of view of the principal's office, it becomes

```
room312.westWall.middleWindow
```

This object road map is the parameter for Tony's `openWindow()` method. Therefore, the entire command that comes over the PA system is

```
room312.Jeffries.Tony.openWindow(room312.westWall.middleWindow)
```

If, instead of barking out orders while sitting in the principal's office, you attempt the same task via radio from an orbiting space shuttle to all the inhabitants on Earth, imagine how laborious your object hierarchy is. The complete reference to Tony's `openWindow()` method and the window that you want opened has to be mighty long to distinguish the desired objects from the billions of objects within the space shuttle's view.

The point is that the smaller the scope of the object-oriented world you're programming, the more you can assume about the location of objects. For client-side JavaScript, the scope is no wider than the browser itself. In other words, every object that a JavaScript script can work

with resides within the browser application. With few exceptions, a script does not access anything about your computer hardware, operating system, other applications, desktop, or any other stuff beyond the browser program.

The browser document object road map

Figure 14-1 shows the lowest common denominator document object hierarchy that is implemented in all scriptable browsers. Notice that the `window` object is the topmost object in the entire scheme. Everything you script in JavaScript is in the browser's window.

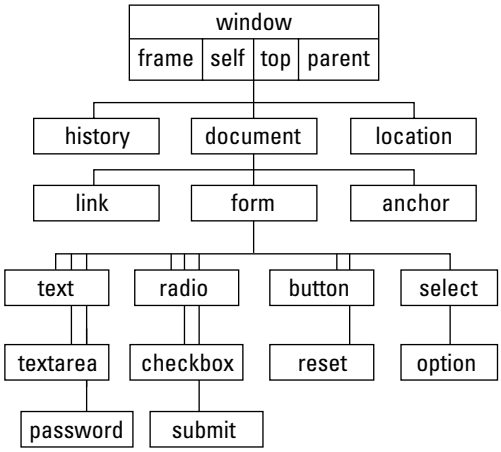


Figure 14-1: The lowest common denominator browser document object hierarchy.

Pay attention to the shading of the concentric rectangles. Every object in the same shaded area is at the same level relative to the `window` object. When a line from an object extends to the next darker shaded rectangle, that object contains all the objects in darker areas. There exists, at most, one of these lines between levels. The `window` object contains the `document` object; the `document` object contains a form object; a form object contains many different kinds of form control elements.

Study Figure 14-1 to establish a mental model for the basic scriptable elements of a Web page. Models of more recent browsers have more objects in their hierarchies, but the fundamental organization remains. After you script these objects several times, the object hierarchy will become second nature to you — even if you don't necessarily remember every detail (property, method, and event handler) of every object. At least you know where to look for information.

How Document Objects Are Born

Most of the objects that a browser creates for you are established when an HTML document loads into the browser. The same kind of HTML code you use to create links, anchors, and input elements tells a JavaScript-enhanced browser to create those objects in memory. The objects are there whether or not your scripts call them into action.

The only visible differences to the HTML code for defining those objects are the one or more optional attributes specifically dedicated to JavaScript. By and large, these attributes specify the event you want the user interface element to react to and what JavaScript should do when the user takes that action. By relying on the document's HTML code to perform the object generation, you can spend more time figuring out how to do things with those objects or have them do things for you.

Bear in mind that objects are created in their load order. And if you create a multiframe environment, a script in one frame cannot communicate with another frame's objects until both frames load. This trips up a lot of scripters who create multiframe and multiwindow sites (more in Chapter 16).

Object Properties

A property generally defines a particular current setting of an object. The setting may reflect a visible attribute of an object, such as the state of a checkbox (checked or not); it may also contain information that is not so obvious, such as the action and method of a submitted form.

Document objects have most of their initial properties assigned by the attribute settings of the HTML tags that generate the objects. Thus, a property may be a word (for example, a name) or a number (for example, a size). A property can also be an array, such as an array of images contained by a document. If the HTML does not include all attributes, the browser usually fills in a default value for both the attribute and the corresponding JavaScript property.

When used in script statements, property names are case-sensitive. Therefore, if you see a property name listed as `bgColor`, you must use it in a script statement with that exact combination of lowercase and uppercase letters. But when you set an initial value of a property by way of an HTML attribute, the attribute name (like all of HTML) is not case-sensitive. Thus, `<BODY BGCOLOR="white">` and `<body bgcolor="white">` both set the same `bgColor` property value. Although XHTML won't validate correctly if you use anything but lowercase letters for tag and attribute names, most browsers continue to be case-insensitive for markup, regardless of the HTML or XHTML version you specify for the page's `DOCTYPE`. The case for property names is not influenced by the case of the markup attribute name.

Each property determines its own read/write status. Some properties are read-only, whereas you can change others on the fly by assigning a new value to them. For example, to put some new text into a text box object, you assign a string to the object's `value` property:

```
document.forms[0].phone.value = "555-1212";
```

Once an object contained by the document exists (that is, its HTML is loaded into the document), you can also add one or more custom properties to that object. This can be helpful if you wish to associate some additional data with an object for later retrieval. To add such a property, simply specify it in the same statement that assigns a value to it:

```
document.forms[0].phone.delimiter = "-";
```

Any property you set survives as long as the document remains loaded in the window and scripts do not overwrite the object. Be aware, however, that reloading the page usually destroys custom properties.

A Note to Experienced Object-Oriented Programmers

Although the basic object model hierarchy appears to have a class/subclass relationship, many of the traditional aspects of a true, object-oriented environment don't apply to the model. The original JavaScript document object hierarchy is a *containment hierarchy*, not an *inheritance hierarchy*. No object inherits properties or methods of an object higher up the chain. Nor is there any automatic message passing from object to object in any direction. Therefore, you cannot invoke a window's method by sending a message to it via the `document` or a form object. All object references must be explicit.

Predefined document objects are generated only when the HTML code containing their definitions loads into the browser. You cannot modify many properties, methods, and event handlers in early object models once you load the document into the browser. In Chapter 33, you learn how to create your own objects, but those objects do not present new visual elements on the page that go beyond what HTML, Java applets, and plug-ins can portray.

Inheritance *does* play a role, as you will see later in this chapter, in the object model defined by the W3C. The new hierarchy is of a more general nature to accommodate requirements of XML as well as HTML. But the containment hierarchy for HTML objects, as described in this section, is still valid in W3C DOM-compatible browsers.

Object Methods

An object's method is a command that a script can give to that object. Some methods return values, but that is not a prerequisite for a method. Also, not every object has methods defined for it. In a majority of cases, invoking a method from a script causes some action to take place. The resulting action may be obvious (such as resizing a window) or something more subtle (such as sorting an array in memory).

All methods have parentheses after them, and they always appear at the end of an object's reference. When a method accepts or requires parameters, the parameter values go inside the parentheses (with multiple parameters separated by commas).

While an object has its methods predefined by the object model, you can also assign one or more additional methods to an object that already exists (that is, after its HTML is loaded into the document). To do this, a script in the document (or in another window or frame accessible by the document) must define a JavaScript function and then assign that function to a new property name of the object. In the following example written to take advantage of version 4 or later browser features, the `fullScreen()` function invokes one `window` object method and adjusts two `window` object properties. By assigning the function reference to the new `window.maximize` property, I define a `maximize()` method for the `window` object. Thus, a button's event handler can call that method directly.

```
// define the function
function fullScreen() {
   this.moveTo(0,0);
   this.outerWidth = screen.availWidth;
   this.outerHeight = screen.availHeight;
}
// assign the function to a custom property
window.maximize = fullScreen;
...
<!-- invoke the custom method -->
<input type="button" value="Maximize Window" onclick="window.maximize()" />
```

Object Event Handlers

An *event handler* specifies how an object reacts to an event that is triggered by a user action (for example, a button click) or a browser action (for example, the completion of a document load). Going back to the earliest JavaScript-enabled browser, event handlers were defined inside HTML tags as extra attributes. They included the name of the attribute, followed by an equal sign (working as an assignment operator) and a string containing the script statement(s) or function(s) to execute when the event occurs (see Chapter 5). Event handlers also have other forms. In NN3+ and IE4+, event handlers have corresponding methods for their objects and every event handler is a property of its object.

Event handlers as methods

Consider a button object whose sole event handler is `onclick`. This means whenever the button receives a click event, the button triggers the JavaScript expression or function call assigned to that event handler in the button's HTML definition:

```
<input type="button" name="clicker" value="Click Me" onclick="doIt()" />
```

Normally, that click event is the result of a user physically clicking the button in the page. In NN3+ and IE4+, you can also trigger the event handler with a script by calling the event handler as if it were a method of the object:

```
document.formName.clicker.onclick();
```

Invoking an event handler this way is different from using a method to simulate the physical action denoted by the event. For example, imagine a page containing three simple text fields. One of those fields has an `onfocus` event handler defined for it. Physically tabbing to or clicking in that field brings focus to the field and thereby triggers its `onfocus` event handler. If the field does not have focus, a button can invoke that field's `onfocus` event handler by referencing it as a method:

```
document.formName.fieldName.onfocus();
```

This scripted action does not bring physical focus to the field. The field's own `focus()` method, however, does that under script control.

A byproduct of an event handler's capability to act like a method is that you can define the action of an event handler by defining a function with the event handler's name. For example, instead of specifying an `onload` event handler in a document's `<body>` tag, you can define a function like this:

```
function onload() {
   statements
}
```

This capability is particularly helpful if you want event handler actions confined to a script running in NN3, IE4, or later. Your scripts don't require special traps for Navigator 2 or Internet Explorer 3.

Event handlers as properties

Although event handlers are commonly defined in an object's HTML tag, you also have the power in NN3+ and IE4+ to assign or change an event handler just like you assign or change the property of an object. The value of an event handler property looks like a function definition. For example, given this HTML definition:

```
<input type="text" name="entry" onfocus="doIt()" />
```

the value of the object's `onfocus` (all lowercase) property is

```
function onfocus() {
   doIt();
}
```

You can, however, assign an entirely different function to an event handler by assigning a function reference to the property. Such references don't include the parentheses that are part of the function's definition. (You see this again much later in Chapter 33 when you assign functions to object properties.)

Using the same text field definition you just looked at, you can assign a different function to the event handler because based on user input elsewhere in the document you want the field to behave differently when it receives the focus. If you define a function like this:

```
function doSomethingElse() {
   statements
}
```

you can then assign the function to the field with this assignment statement:

```
document.formName.entry.onfocus = doSomethingElse;
```

Because the new function reference is written in JavaScript, you must observe case for the function name. Although NN4 accepts interCap versions of the event handler names, you are best served across all browsers by sticking with all lowercase event handler names as properties.

Be aware, however, that as with several settable object properties that don't manifest themselves visually, any change you make to an event handler property disappears with a document reload. Therefore, I advise you not to make such changes except as part of a script that also invokes the event handler like a method: Any gap in time leaves room for users to reload the page accidentally or intentionally.

If your scripts create new element objects dynamically, you can assign event handlers to these objects by way of event handler properties. For example, the following code uses W3C DOM syntax to create a new button input element and assign an `onclick` event handler that invokes a function defined elsewhere in the script:

```
var newElem = document.createElement("input");
newElem.type = "button";
newElem.value = "Click Here";
newElem.onclick = doIt;
document.forms[0].appendChild(newElem);
```

Because every event handler operates as both property and method, I don't list these properties and methods as part of each object's definition in the next chapters. You can be assured this feature works for every JavaScript object that has an event handler starting with Navigator 3 and Internet Explorer 4.

Object Model Smorgasbord

A survey of the entire evolution of scriptable browsers from NN2 and IE3 through IE6 and Moz1 reveals six (yes, six!) distinct document object model families. Even if your job entails developing content for just one current browser version, you may be surprised that family members from more than one document object model inhabit your authoring space.

Studying the evolution of the object model is extremely valuable for newcomers to scripting. It is too easy to learn the latest object model gadgets in your current browser, only to discover that your heroic scripting efforts are lost on earlier browsers accessing your pages. Therefore, take a look at the six major object model types and how they came into being. Table 14-1 lists the object model families (in chronological order of their release) and the browser versions that support them. Later in this chapter are some guidelines you can follow to help you choose the object model(s) that best suit your users' "appetites."

Table 14-1: Object Model Families

Model	*Browser Support*
Basic Object Model	NN2, NN3, IE3/J1, IE3/J2, NN4, IE4, IE5, IE5.5, IE6, Moz1, Safari1
Basic Plus Images	NN3, IE3.01 (Mac only), NN4, IE4, IE5, IE5.5, IE6, Moz1, Safari1
NN4 Extensions	NN4
IE4 Extensions	IE4, IE5, IE5.5, IE6 (some features in all versions require Win32 OS)
IE5 Extensions	IE5, IE5.5, IE6 (some features in all versions require Win32 OS)
W3C DOM (I and II)	IE5 (partial), IE5.5 (partial), IE6 (partial), Moz1 (most), Safari 1 (partial)

Basic Object Model

The first scriptable browser, Netscape Navigator 2, implemented a very basic document object model. Figure 14-1 provides a visual guide to the objects that were exposed to scripting. The hierarchical structure starts with the window and drills inward toward the document, forms, and form control elements. A document is a largely immutable page on the screen. Only elements that are by nature interactive — links and form elements such as text fields, buttons, and so on — are treated as objects with properties, methods, and event handlers.

The heavy emphasis on form controls opened up numerous possibilities that were radical ideas at the time. Because a script could inspect the values of form controls, forms could be pre-validated on the client. If the page included a script that performed some calculations, data entry and calculated results were displayed via editable text fields.

Additional objects that exist outside of the document — `window`, `history`, and `location` objects — provide scriptable access to simple yet practical properties of the browser that loads the page. The most global view of the environment is the `navigator` object, which includes properties about the browser brand and version.

When Internet Explorer 3 arrived on the scene, the short life of Navigator 2 was nearing its end. Even though NN3 was already widely available in prerelease form, IE3 implemented the basic object model from NN2 (plus one `window` object property from NN3). Therefore, despite the browser version number discrepancy, NN2 and IE3 are essentially the same with respect to their document object models. For a brief moment in Internet Time, there was nearly complete harmony between Microsoft and Netscape document object models — albeit at a very simple level.

Basic Object Model Plus Images

A very short time after IE3 was released, Netscape released Navigator 3 with an object model that built upon the original version. A handful of existing objects — especially the `window` object — gained new properties, methods, and/or event handlers. Scripts could also communicate with Java applets as objects. But the biggest new object on the scene was the `Image` object and the array of image objects exposed to the `document` object.

Most of the properties for an NN3 image object gave read-only access to values typically assigned to attributes in the `<img>` tag. But you could modify one property — the `src` property — after the page loaded. Scripts could swap out images within the fixed image rectangle. Although these new image objects didn't have mouse-related event handlers, nesting an image inside a link (which had `onmouseover` and new `onmouseout` event handlers) let scripts implement "image rollovers" to liven up a page.

As more new scripters investigated the possibilities of adding JavaScript to their pages, frustration ensued when the image swapping they implemented for NN3 failed to work in IE3. Although you could easily script around the lack of an image object to prevent script errors in IE3 (see Chapter 12), the lack of this "cool" page feature disappointed many. Had they also taken into account the installed base of NN2 in the world, they would have been disappointed there, too. To confuse matters even more, the Macintosh version of IE 3.01 (the second release of the IE3/Mac browser) implemented scriptable image objects.

Despite these rumblings of compatibility problems to come, the object model implemented in Navigator 3 eventually became the baseline reference for future document object models. With few exceptions, code written for this object model runs on all browsers from NN3 and IE4 through the latest versions of both brands. Exceptions primarily involve Java applet object support in non-Windows versions of IE4+.

Navigator 4–Only Extensions

The next browser released to the world was Netscape Navigator 4. Numerous additions to the existing objects put more power into the hands of scripters. You could move and resize browser windows within the context of script-detectable `screen` object properties (for example, how big the user's monitor screen was). Two concepts that represented new thinking about the object model were an enhanced event model and the layer object.

Event capture model

Navigator 4 added many new events to the repertoire. Keyboard events and more mouse events (`onmousedown` and `onmouseup`) allowed scripts to react to more user actions on form control elements and links. All of these events worked as they did in previous object models in which event handlers were typically assigned as attributes to an element's tag (although you could also assign event handlers as properties in script statements). To facilitate some of the Dynamic HTML potential in the rest of the Navigator 4 object model, the event model was substantially enhanced.

At the root of the system is the idea that when a user performs some physical action on an event-aware object (for example, clicking a form button), the event reaches that button from top down through the document object hierarchy. If you have multiple objects that share the

same event handler, it may be more convenient to capture that event in just one place — the `window` or `document` object level — rather than assigning the same event handler to all the elements. The default behavior of Navigator 4 allowed the event to reach the target object, just as it had in earlier browsers. But you could also turn on *event capture* in the `window`, `document`, or layer object. Once captured, the event could be handled at the upper level, preprocessed before being passed onto its original target, or redirected to another object altogether.

Whether or not you capture events, the Navigator 4 event model produces an `event` object (lowercase "e" to distinguish from the static `Event` object) for each event. That object contains properties that reveal more information about the specific event, such as the keyboard character pressed for a keyboard event or the position of a click event on the page. Any event handler can inspect `event` object properties to learn more about the event and process the event accordingly.

Layers

Perhaps the most radical addition to the NN4 object model was a new object that reflected an entirely new HTML element, the `layer` element. A *layer* is a container that is capable of holding its own HTML document, yet it exists in a plane in front of the main document. You can move, size, and hide a layer under script control. This new element allowed, for the first time, overlapping elements in an HTML page.

To accommodate the layer object in the document object hierarchy, Netscape defined a nesting hierarchy such that a layer was contained by a document. As the result, the `document` object acquired a property (`document.layers`) that was an array of layer objects in the document. This array exposed only the first level of layer(s) in the current `document` object. References to a layer in the main document started with any one of the following:

```
document.layerName
document.layers[n]
document.layers[layerName]
```

Each layer had its own `document` object because each layer could load an external HTML document if desired. Thus, if a script needed access to, say, a form control element inside a layer, the reference would begin:

```
document.layerName.document.forms[0]....
```

If a layer contained yet another layer, the reference grew even longer:

```
document.outerLayerName.document.innerLayerName.document.forms[0]...
```

As a positionable element, a layer object had numerous properties and methods that allowed scripts to move, hide, show, and change its stacking order.

Unfortunately for Netscape, the W3C did not agree to make the `<layer>` tag a part of the HTML 4.0 specification. As such, it is an orphan element that exists only in Navigator 4 (not implemented in Moz1 or later). The same goes for the scripting of the layer object and its nested references. Navigator 4 does, however, implement a little bit of the HTML 4.0 and CSS specifications for positionable elements because you can assign Cascading Style Sheets (with the position and related attributes) to `div` and `span` elements in NN4. Navigator treats positioned `div` or `span` elements as near equivalents of layer objects for scripting purposes. This means, however, that even if you can get the HTML to work the same across browsers (not always guaranteed due to occasionally different rendering characteristics of positioned `div`

elements in NN4 and IE4), the scripting for NN4 must adhere to the layer syntax, which differs from the IE4 CSS syntax.

Internet Explorer 4+ Extensions

Microsoft broke important new ground with the release of IE4, which came several months after the release of NN4. The main improvements were in the exposure of all HTML elements, scripted support of CSS, and a new event model. Some other additions were available only on Windows 32-bit operating system platforms.

HTML element objects

The biggest change to the object model world was that every HTML element became a scriptable object, while still supporting the original object model. Microsoft invented the `document.all` array (also called a *collection*). This array contains references to every element in the document, regardless of element nesting. If you assign an identifier (name) to the `id` attribute of an element, you can reference the element by the following syntax:

```
document.all.elementID
```

In most cases, you can also drop the `document.all.` part of the reference and begin with only the element ID.

Every element object has an entirely new set of properties and methods that give scripters a level of control over document content unlike anything seen before. Table 14-2 shows the properties and methods that all HTML element objects have in common in IE4 (properties followed by brackets are arrays).

Table 14-2: IE4 HTML Element Features in Common

Properties	*Methods*
all[]	click()
children[]	contains()
className	getAttribute()
document	insertAdjacentHTML()
filters[]	insertAdjacentText()
id	removeAttribute()
innerHTML	scrollIntoView()
innerText	setAttribute()
isTextEdit	
lang	
language	
offsetHeight	
offsetLeft	
offsetParent	

Continued

Table 14-2 *(continued)*

Properties	*Methods*
offsetTop	
offsetWidth	
outerHTML	
outerText	
parentElement	
parentTextEdit	
sourceIndex	
style	
tagName	
title	

You can find details for all of the items listed in Table 14-2 in Chapter 15. But several groups of properties deserve special mention here.

Four properties (`innerHTML`, `innerText`, `outerHTML`, and `outerText`) provide read/write access to the actual content within the body of a document. This means that you no longer must use text boxes to display calculated output from scripts. You can modify content inside paragraphs, table cells, or anywhere on the fly. The browser's rendering engine immediately reflows a document when the dimensions of an element's content change. That feature puts the "Dynamic" in "Dynamic HTML." To those of us who scripted the static pages of earlier browsers, this feature — now taken for granted — was nothing short of a revelation.

The series of "offset" properties are related to the position of an element on the page. These properties are distinct from the kind of positioning performed by CSS. Therefore, you can get the dimensions and location of any element on the page, making it easier to move positionable content atop elements that are part of the document and may appear in various locations due to the browser window's current size.

Finally, the `style` property is the gateway to CSS specifications defined for the element. Importantly, the script can modify the numerous properties of the `style` object. Therefore, you can modify font specifications, colors, borders, and the positioning properties after the page loads. The dynamic reflow of the page takes care of any layout changes that the alteration requires (for example, adjusting to a bigger font size).

Element containment hierarchy

While IE4 still recognizes the element hierarchy of the original object model (see Figure 14-1), the document object model for IE4 does not extend this kind of hierarchy fully into other elements. If it did, it would mean that `td` elements inside a table might have to be addressed via its next outer `tr` or `table` element (just as a form control element must be addressed via its containing `form` element). Look at Figure 14-2 to see how all HTML elements are grouped together under the `document` object. The `document.all` array flattens the containment hierarchy as far as referencing object goes. A reference to the most deeply nested TD element is still `document.all.`*`cellID`*. The highlighted pathway from the `window` object is the predominant reference path used when working with the IE4 document object hierarchy.

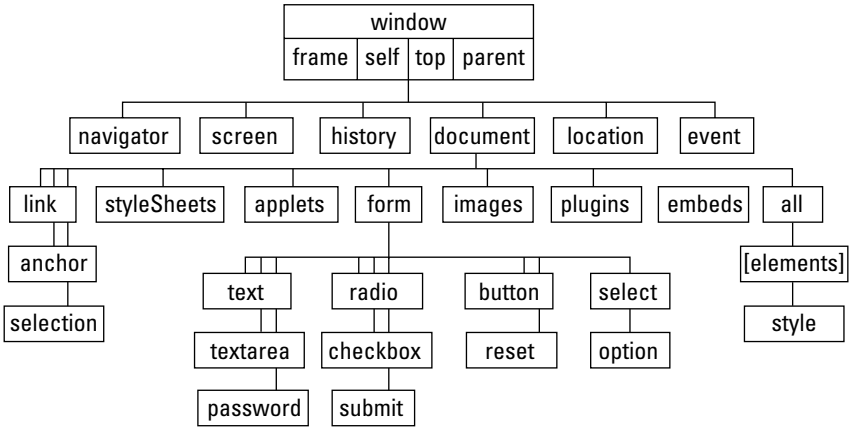


Figure 14-2: The IE4 document object hierarchy.

Element containment in IE4, however, is important for other reasons. Because an element can inherit some stylesheet attributes from an element that contains it, you should devise a document's HTML by embedding every piece of content inside a container. Paragraph elements are text containers (with start and end tags), not tall line breaks between text chunks. IE4 introduced the notion of a parent-child relationship between a container and elements nested within it. Also, the position of an element may be calculated relative to the position of its next outermost positioning context.

The bottom line here is that element containment doesn't have anything to do with object references (like the original object model). It has everything to do with the *context* of an element relative to the rest of the page's content.

Cascading Style Sheets

By arriving a bit later to market with its version 4 browser than Netscape, Microsoft benefited from having the CSS Level 1 specification more fully developed before the browser's release. Therefore, the implementation is far more complete than that of NN4 (but it is not 100 percent compatible with the standard).

I should point out that the scriptability of stylesheet properties is a bit at odds with the first-generation CSS specification, which seemed to ignore the potential of scripting styles with JavaScript. Many CSS attribute names are hyphenated words (for example, `text-align`, `z-index`). But hyphens are not allowed in identifier names in JavaScript. This necessitated conversion of the multiword CSS attribute names to interCap JavaScript property names. Therefore, `text-align` becomes `textAlign` and `z-index` becomes `zIndex`. You can access all of these properties through an element's `style` property:

```
document.all.elementID.style.stylePropertyName
```

One byproduct of the scriptability of stylesheets in IE4 and later is what some might call the *phantom page syndrome*. This occurs when the layout of a page is handled after the primary HTML for the page has downloaded to the browser. As the page loads, not all content may be visible, or it may be in a visual jumble. An `onload` event handler in the page then triggers scripts to set styles and/or content for the page. Elements jump around to get to their final resting places. This may be disconcerting to some users who at first see a link to click; but by

the time the cursor reaches the click location, the page has reflowed, thereby moving the link to somewhere else on the page.

For Internet Explorer users with 32-bit Windows operating systems, IE4 includes some extra features in the object model that can enhance presentations. *Filters* are stylesheet additives that offer a variety of visual effects on body text. For example, you can add a drop shadow or a glowing effect to text by simply applying filter styles to the text, or you can create the equivalent of a slide presentation by placing the content of each slide in a positioned `div` element. Although filters follow the CSS syntax, they are not a part of the W3C specification.

Event bubbling

Just as Netscape invented an event model for NN4, so, too, did Microsoft invent one for IE4. Unfortunately for cross-browser scripters, the two event models are quite different. Instead of events trickling down the hierarchy to the target element, an IE event starts at the target element and, unless instructed otherwise, "bubbles up" through the element containment hierarchy to eventually reach the `window` object. At any object along the way, an event handler can perform additional processing on that event if desired. Therefore, if you want a single event handler to process all click events for the page, assign the event handler to the `body` or `window` object so the events reach those objects (provided the event bubbling isn't cancelled by some other object along the containment hierarchy).

IE also has an `event` object (a property of the `window` object) that contains details about the event, such as the keyboard key pressed for a keyboard event and the location of a mouse event. Names for these properties are entirely different from the event object properties of NN4.

Despite what seems like incompatible, if not completely opposite, event models in NN4 and IE4, you can make a single set of scripts handle events in both browsers (see Chapter 25 and Chapter 56 on the CD-ROM for examples). In fact, the two event models are made to work together in the W3C DOM Level 2 specification, described later in this chapter.

Event binding of scripts

IE4 introduced an additional way of binding events to objects via a `<script>` tag that has two additional, non-W3C attributes: `for` and `event` (see a syntax example in Chapter 13 in the section titled "<script for> tags"). The value assigned to the `for` attribute is the ID of an element object for which the script is intended; the value of the `event` attribute is the name of the event handler (for example, `onclick`) by which the script statements within the tag are to be triggered.

Inside the tags are straight script statements, but when the browser sees the special attributes, execution is deferred until the event fires for the designated object. The instant the event fires for the object, the script statements inside the tag execute. This special form of script tag takes the place of a function definition assigned to the event handler by other means. This technique appears to have been a "dry run" for what eventually became DHTML behaviors in IE5/Windows (see the following section).

You can use this binding method only if you run the page inside IE4+. All other browsers, including IE3, ignore the special attributes and treat the statements inside the tags as statements to execute as the page loads.

Internet Explorer 5+ Extensions

With the release of IE5, Microsoft built more onto the proprietary object model it launched in IE4. Although the range of objects remained pretty much the same, the number of properties, methods, and event handlers for the objects increased dramatically. Some of those additions were added to meet some of the specifications of the W3C DOM (discussed in the next section), occasionally causing a bit of incompatibility with IE4. But Microsoft also pushed ahead with efforts for Windows users only that may not necessarily become industry standards: DHTML behaviors and HTML applications.

A *DHTML behavior* is a chunk of script — saved as an external file — that defines some action (usually a change of one or more style properties) that you can apply to any kind of element. The goal is to create a reusable component that you can load into any document whose elements require that behavior. The behavior file is known as an *HTML component*, and the file has an `.htc` extension. Components are XML documents whose XML tags specify events and event-handling routines for whatever element is assigned that behavior. Script statements in `.htc` documents are written inside `<script>` tag sets just as in regular, scriptable HTML documents. As an example of a DHTML behavior, you can define a behavior that turns an element's text to red whenever the cursor rolls atop it and reverts to black when the cursor rolls out. When you assign the behavior to an element in the document (via CSS-like rule syntax), the element picks up that behavior and responds to the user accordingly. You can apply that same behavior to any element(s) you like in the document. (Microsoft has submitted behaviors to the W3C for possible inclusion into CSS Level 3.) You can see an example of a DHTML behavior in Chapter 15's description of the `addBehavior()` method and read an extended discussion in Chapter 47 on the CD-ROM.

HTML applications (*HTAs* in Microsoft parlance) are HTML files that include an XML element known as the `hta:application` element. You can download an HTA to IE5+ from the server as if it were a Web page (although its file extension is `.hta` rather than `.htm` or `.html`). A user can also install an HTA on a client machine so it behaves very much like an application with a Desktop icon and significant control over the look of the window. HTAs are granted greater security privileges on the client so that this "application" can behave more like a regular program. In fact, you can elect to turn off the system menu bar and use DHTML techniques to build your own menu bar for the application. Implementation details of HTAs are beyond the scope of this book, but you should be aware of their existence. More information is available at `http://msdn.microsoft.com`.

The W3C DOM

Conflicting browser object models from Netscape and Microsoft made life difficult for developers. Scripters craved a standard that would serve as a common denominator much like HTML and CSS standards did for content and styles. The W3C took up the challenge of creating a document object model standard, the W3C DOM.

The charter of the W3C DOM working group was to create a document object model that could be applied to both HTML and XML documents. Because an XML document can have tags of virtually any name (as defined by the Document Type Definition), it has no intrinsic structure or fixed vocabulary of elements like an HTML document does. As a result, the DOM specification had to accommodate the known structure of HTML (as defined in the HTML 4.0 specification) as well as the unknown structure of an XML document.

To make this work effectively, the working group divided the DOM specification into two sections. The first, called the *Core DOM*, defines specifications for the basic document structure that both HTML and XML documents share. This includes notions of a document containing elements that have tag names and attributes; an element is capable of containing zero or more other elements. The second part of the DOM specification addresses the elements and other characteristics that apply only to HTML. The HTML portion "inherits" all the features of the Core DOM, while providing a measure of backward compatibility to object models already implemented in legacy browsers and providing a framework for new features.

It is important for veteran scripters to recognize that the W3C DOM does not specify all features from existing browser object models. Many features of the IE4 (and later) object model are not part of the W3C DOM specification. This means that if you are comfortable in the IE environment and wish to shift your focus to writing for the W3C DOM spec, you have to change some practices as highlighted in this chapter. Navigator 4 page authors lose the `<layer>` tag (which is not part of HTML 4.0 and likely will never see the light of day in a standard) as well as the layer object. In many respects, especially with regard to Dynamic HTML applications, the W3C DOM is an entirely new DOM with new concepts that you must grasp before you can successfully script in the environment.

By the same token, you should be aware that whereas Mozilla-based browsers go to great lengths to implement all of DOM Level 1 and most of Level 2, Microsoft (for whatever reason) features only a partial implementation of the W3C DOM through IE5.5. Although IE6 implements more W3C DOM features, some important parts, notably W3C DOM events, are missing. Other modern browsers, such as Safari, provide basic W3C DOM support, but have not yet caught up with Mozilla levels of DOM support.

DOM levels

Like most W3C specifications, one version is rarely enough. The job of the DOM working group was too large to be swallowed whole in one sitting. Therefore, the DOM is a continually evolving specification. The timeline of specification releases rarely coincides with browser releases. Therefore, it is very common for any given browser release to include only some of the most recent W3C version.

The first formal specification, DOM Level 1, was released well after NN4 and IE4 shipped. The HTML portion of Level 1 includes DOM Level 0. This is essentially the object model as implemented in Navigator 3 (and for the most part in Internet Explorer 3 plus image objects). Perhaps the most significant omission from Level 1 is an event model (it ignores even the simple event model implemented in NN2 and IE3).

DOM Level 2 builds on the work of Level 1. In addition to several enhancements of both the Core and HTML portions of Level 1, Level 2 adds significant new sections (published as separate modules) on the event model, ways of inspecting a document's hierarchy, XML namespaces, text ranges, stylesheets, and style properties. Work on Level 3 is under way, but very little of it has yet reached browsers.

What stays the same

By adopting DOM Level 0 as the starting point of the HTML portion of the DOM, the W3C provided a way for a lot of existing script code to work even in a W3C DOM-compatible browser. Every object you see in the original object model starting with the `document` object (see Figure 14-1) plus the image object are in DOM Level 0. Almost all of the same object properties and methods are also available.

More importantly, when you consider the changes to referencing other elements in the W3C DOM (discussed in the next section), we're lucky that the old ways of referencing objects such as forms, form control elements, and images still work. Had the working group been planning from a clean slate, it is unlikely that the `document` object would have been given properties consisting of arrays of forms, links, and images.

The only potential problems you could encounter with your existing code have to do with a handful of properties that used to belong to the `document` object. In the new DOM, four style-related properties of the `document` object (`alinkColor`, `bgColor`, `linkColor`, and `vlinkColor`) become properties of the `body` object (referenced as `document.body`). In addition, the three link color properties pick up new names in the process (`aLink`, `link`, `vLink`). It appears, however, that for now, IE6 and Moz1 maintain backward compatibility with the older document object color properties.

Also, note that the DOM specification concerns itself only with the document and its content. Objects such as `window`, `navigator`, and `screen` are not part of the DOM specification through Level 2. Scripters are still at the mercy of browser makers for compatibility in these areas, but the `window` object (or its equivalent) likely will be added to the W3C DOM Level 3.

What isn't available

As mentioned earlier, the W3C DOM is not simply a restatement of existing browser specifications. Many convenience features of the IE and NN object models do not appear in the W3C DOM. If you develop Dynamic HTML content in IE4+ or NN4, you have to learn how to get along without some of these conveniences.

Navigator 4's experiment with the `<layer>` tag was not successful in the W3C process. As a result, both the tag and the scripting conventions surrounding it do not exist in the W3C DOM. To some scripters' relief, the `document.layerName` referencing scenario (even more complex with nested layers) disappears from the object model entirely. A positioned element is treated as just another element that has some special stylesheet attributes that enable you to move it anywhere on the page, stack it amid other positioned elements, and hide it from view.

Among popular IE4+ features missing from the W3C DOM are the `document.all` collection of HTML elements and four element properties that facilitate dynamic content: `innerHTML`, `innerText`, `outerHTML`, and `outerText`. A new W3C way provides for acquiring an array of all elements in a document, but generating HTML content to replace existing content or be inserted in a document requires a tedious sequence of statements (see the section "New DOM concepts" later in this chapter). Mozilla, however, has implemented the `innerHTML` property for HTML element objects in Mozilla-based browsers (Moz1+).

"New" HTML practices

Exploitation of Dynamic HTML possibilities in both IE4+ and the W3C DOM relies on some HTML practices that may be new to long-time HTML authors. At the core of these practices (espoused by the HTML 4.0 specification) is making sure that all content is within an HTML container of some kind. Therefore, instead of using the `<p>` tag as a separator between blocks of running text, surround each paragraph of the running text with a `<p>...</p>` tag set. If you don't do it, the browser treats each `<p>` tag as the beginning of a paragraph and ends the paragraph element just before the next `<p>` tag or other block-level element.

While recent browsers continue to accept the omission of certain end tags (for `td`, `tr`, and `li` elements, for instance), it is best to get in the habit of supplying these end tags. If for no other reason, they help you visualize where an element's sphere of influence truly begins and ends.

Any element that you intend to script — whether to change its content or its style — should have an identifier assigned to the element's `id` attribute. Form control elements still require `name` attributes if you submit the form content to a server. But you can freely assign a different identifier to a control's `id` attribute. Scripts can use either the `id` or the `document.formReference.elementName` reference to reach a control object. Identifiers are essentially the same as the values you assign to the `name` attributes of form and form input elements. Following the same rules for the `name` attribute value, an `id` identifier must be a single word (no white space), it cannot begin with a numeral (to avoid conflicts in JavaScript), and it should avoid punctuation symbols except for the underscore. While an element can be accessed by numeric index within the context of some surrounding element (such as the `body`), this is a risky practice when content is under construction. Unique identifiers make it much easier for scripts to reference objects and are not affected by changes in content order.

New DOM concepts

With the W3C DOM come several concepts that may be entirely new to you unless you have worked extensively with the terminology of tree hierarchies. Concepts that have the most impact on your scripting are new ways of referencing elements and nodes.

Element referencing

Script references to objects in the DOM Level 0 are observed in the W3C DOM for backward compatibility. Therefore, a form input element whose `name` attribute is assigned the value `userName` is addressed just like it always is:

```
document.forms[0].userName
```

or

```
document.formName.userName
```

But because all elements of a document are exposed to the `document` object, you can use the new `document` object method to access any element whose ID is assigned. The method is `document.getElementById()`, and the sole parameter is a string version of the identifier of the object whose reference you wish to get. To help put this in context with what you may have used with the IE4 object model, consider the following HTML paragraph tag:

```
<p id="myParagraph">...</p>
```

In IE4+, you can reference this element with

```
var elem = document.all.myParagraph;
```

Although the `document.all` collection is not implemented in the W3C DOM, use the new `document` object method (available in IE5+, Moz1+, Safari, and others) that enables you to access any element by its ID:

```
var elem = document.getElementById("myParagraph");
```

Unfortunately for scripters, this method is difficult to type since it is case-sensitive, so watch out for that ending lowercase "d".

A hierarchy of nodes

The issue surrounding containers (described earlier) comes into play for the underlying architecture of the W3C DOM. Every element or freestanding chunk of text in an HTML (or XML) document is an object that is contained by its next outermost container. Let's look at a

simple HTML document to see how this system works. Listing 14-1 is formatted to show the containment hierarchy of elements and string chunks.

Listing 14-1: **A Simple HTML Document**

```
<html>
   <head>
      <title>
         A Simple Page
      </title>
   </head>

   <body>
      <p id="paragraph1">
         This is the
         <em id="emphasis1">
            one and only
         </em>
         paragraph on the page.
      </p>
   </body>
</html>
```

What you don't see in the listing is a representation of the `document` object. The `document` object exists automatically when this page loads into a browser. Importantly, the `document` object encompasses everything you see in Listing 14-1. Therefore, the `document` object has a single nested element: the `html` element. The `html` element, in turn, has two nested elements: `head` and `body`. The `head` element contains the `title` element, while the `title` element contains a chunk of text. Down in the `body` element, the `p` element contains three pieces: a string chunk, the `em` element, and another string chunk.

According to W3C DOM terminology, each container, standalone element (such as a `br` element), or text chunk is known as a *node* — a fundamental building block of the W3C DOM. Nodes have parent-child relationships when one container holds another. As in real life, parent-child relationships extend only between adjacent generations, so a node can have zero or more children. However, the number of third-generation nodes further nested within the family tree does not influence the number of children associated with a parent. Therefore, in Listing 14-1, the `html` node has two child nodes, `head` and `body`, which are *siblings* that share the same parent. The `body` element has one child (`p`) even though that child contains three children (two text nodes and an `em` element node).

If you draw a hierarchical tree diagram of the document in Listing 14-1, it should look like the illustration in Figure 14-3.

If the document's source code contains a Document Type Definition (in a `DOCTYPE` element) above the `<html>` tag, the browser treats that `DOCTYPE` node as a sibling of the HTML element node. In that case, the root `document` node contains two child nodes.

The W3C DOM (through Level 2) defines 12 different types of nodes, 7 of which have direct application in HTML documents. These seven types of nodes appear in Table 14-3 (the rest apply to XML). Of the 12 types, the three most common are the document, element, and text types. All W3C DOM browsers (including IE5+, Moz1, Safari, and others) implement the three common node types, while Moz1 implements all of them.

```
document
+--<html>
    +--<head>
    |   +--<title>
    |       +--"A Simple Page"
    +--<body>
        +--<p ID="paragraph1">
            +--"This is the   "
            +--<em ID="emphasis1">
            |    +--"one and only"
            +--" paragraph on the page."
```

Figure 14-3: Tree diagram of nodes for the document in Listing 14-1.

Table 14-3: W3C DOM HTML-Related Node Types

Type	*Number*	*nodeName*	*nodeValue*	*Description*	*IE5+*	*Moz1*	*Safari1*
Element	1	*tag name*	`null`	Any HTML or XML tagged element	Yes	Yes	Yes
Attribute	2	*attribute name*	*attribute value*	A name-value attribute pair in an element	No[1]	Yes	Yes
Text	3	`#text`	*text content*	A text fragment contained by an element	Yes	Yes	Yes
Comment	8	`#comment`	*comment text*	HTML comment	No[1]	Yes	No
Document	9	`#document`	`null`	Root `document` object	No[1]	Yes	Yes
DocumentType	10	`DOCTYPE`	`null`	DTD specification	No	Yes	No
Fragment	11	`#document-fragment`	`null`	Series of one or more nodes outside of the document	No[1]	Yes	Yes

[1]Implemented in IE6.

Applying the node types of Table 14-3 to the node diagram in Figure 14-3, you can see that the simple page consists of one document node, six element nodes, and four text nodes.

Node properties

A node has many properties, most of which are references to other nodes related to the current node. Table 14-4 lists all properties shared by all node types in DOM Level 2.

Table 14-4: Node Object Properties (W3C DOM Level 2)

Property	*Value*	*Description*	*IE5Win+*	*IE5Mac+*	*Moz1*	*Safari1*
`nodeName`	String	Varies with node type (see Table 14-3)	Yes	Yes	Yes	Yes
`nodeValue`	String	Varies with node type (see Table 14-3)	Yes	Yes	Yes	Yes
`nodeType`	Integer	Constant representing each type	Some	Yes	Yes	Yes
`parentNode`	Object	Reference to next outermost container	Yes	Yes	Yes	Yes
`childNodes`	Array	All child nodes in source order	Yes	Yes	Yes	Yes
`firstChild`	Object	Reference to first child node	Yes	Yes	Yes	Yes
`lastChild`	Object	Reference to last child node	Yes	Yes	Yes	Yes
`previousSibling`	Object	Reference to sibling node up in source order	Yes	Yes	Yes	Yes
`nextSibling`	Object	Reference to sibling node next in source order	Yes	Yes	Yes	Yes
`attributes`	NodeMap	Array of attribute nodes	No	Yes	Yes	Yes
`ownerDocument`	Object	Containing document object	No	Yes	Yes	Yes
`namespaceURI`	String	URI to namespace definition (element and attribute nodes only)	No	No	Yes	Yes
`prefix`	String	Namespace prefix (element and attribute nodes only)	No	No	Yes	Yes
`localName`	String	Applicable to namespace-affected nodes	No	No	Yes	Yes

You can find all of the properties shown in Table 14-4 that also show themselves to be implemented in IE5+ or Moz1 in Chapter 15's listing of properties that all HTML element objects have in common. That's because an HTML element, as a type of node, inherits all of the properties of the prototypical node.

To help you see the meanings of the key node properties, Table 14-5 shows the property values of several nodes in the simple page shown in Listing 14-1. For each node column, find the node in Figure 14-3 and then follow the list of property values for that node, comparing the values against the actual node structure in Figure 14-3.

Table 14-5: Properties of Selected Nodes for a Simple HTML Document

Properties	*Nodes*			
	document	html	p	"one and only"
nodeType	9	1	1	3
nodeName	#document	html	p	#text
nodeValue	null	null	null	"one and only"
parentNode	null	document	body	em
previousSibling	null	null	null	null
nextSibling	null	null	null	null
childNodes	html	head body	"This is the " em " paragraph on the page."	(none)
firstChild	html	head	"This is the "	null
lastChild	html	body	" paragraph on the page."	null

The `nodeType` property is an integer that is helpful in scripts that iterate through an unknown collection of nodes. Most content in an HTML document is of type 1 (an HTML element) or 3 (a text node), with the outermost container, the document, of type 9. A node's `nodeName` property is either the name of the node's tag (for an `HTML` element) or a constant value (preceded by a # [hash mark] as shown in Table 14-3). And, what may surprise some, the `nodeValue` property is `null` except for the text node type, in which case the value is the actual string of text of the node. In other words, for `HTML` elements, the W3C DOM does not expose a container's HTML as a string.

It is doubtful that you will use all of the relationship-oriented properties of a node, primarily because there is some overlap in how you can reach a particular node from any other. The `parentNode` property is important because it is a reference to the current node's immediate container. While the `firstChild` and `lastChild` properties point directly to the first and last children inside a container, most scripts generally use the `childNodes` property with array notation inside a `for` loop to iterate through child nodes. If there are no child nodes, the `childNodes` array has a length of zero.

Node methods

Actions that modify the HTML content of a node in the W3C DOM world primarily involve the methods defined for the prototype `Node`. Table 14-6 shows the methods and their support in the W3C DOM–capable browsers.

The Object-Oriented W3C DOM

If you are familiar with concepts of object-oriented (OO) programming, you will appreciate the OO tendencies in the way the W3C defines the DOM. The `Node` object includes sets of properties (see Table 14-4) and methods (see Table 14-6) that are inherited by every object based on the `Node`. Most of the objects that inherit the `Node`'s behavior have their own properties and/or methods that define their specific behaviors. The following figure shows (in W3C DOM terminology) the inheritance tree from the `Node` root object. Most items are defined in the Core DOM, while items shown in boldface are from the HTML DOM portion.

Node
+--Document
| +--**HTMLDocument**
+--CharacterData
| +--Text
| | +--CDATASection
| +--Comment
+--Attr
+--Element
| +--**HTMLElement**
| +-- *(Each specific HTML element)*
+--DocumentType
+--DocumentFragment
+--Notation
+--Entity
+--Entity Reference
+--ProcessingInstruction

W3C DOM Node object inheritance tree.

You can see from the preceding figure that individual HTML elements inherit properties and methods from the generic HTML element, which inherits from the Core `Element` object, which, in turn, inherits from the basic `Node`.

It isn't important to know the `Node` object inheritance to script the DOM. But it does help explain the ECMA Script Language Binding appendix of the W3C DOM recommendation, as well as explain how a simple element object winds up with so many properties and methods associated with it.

Table 14-6: Node Object Methods (W3C DOM Level 2)

Method	*Description*	*IE5+*	*Moz1*	*Safari1*
`appendChild(`*`newChild`*`)`	Adds child node to end of current node	Yes	Yes	Yes
`cloneNode(`*`deep`*`)`	Grabs a copy of the current node (optionally with children)	Yes	Yes	Yes

Continued

Table 14-6 *(continued)*

Method	Description	IE5+	Moz1	Safari1
`hasChildNodes()`	Determines whether current node has children (Boolean)	Yes	Yes	Yes
`insertBefore(new, ref)`	Inserts new child in front of another child	Yes	Yes	Yes
`removeChild(old)`	Deletes one child	Yes	Yes	Yes
`replaceChild(new, old)`	Replaces an old child with a new one	Yes	Yes	Yes
`isSupported(feature, version)`	Determines whether the node supports a particular feature	No	Yes	Yes

The important methods for modifying content are `appendChild()`, `insertBefore()`, `removeChild()`, and `replaceChild()`. Notice, however, that all of these methods assume that the point of view for the action is from the parent of the nodes being affected by the methods. For example, to delete an element (using `removeChild()`), you don't invoke that method on the element being removed, but rather on its parent element. This leaves open the possibility for creating a library of utility functions that obviate having to know too much about the precise containment hierarchy of an element. A simple function that lets a script appear to delete an element actually does so from its parent:

```
function removeElement(elemID) {
   var elem = document.getElementById(elemID);
   elem.parentNode.removeChild(elem);
}
```

If this seems like a long way to go to accomplish the same result as setting the `outerHTML` property of an IE4+ object to empty, you are right. While some of this convolution makes sense for XML, unfortunately the W3C working group doesn't seem to have HTML scripters' best interests in mind. All is not lost, however, as you see later in this chapter.

Generating new node content

The final point about the node structure of the W3C DOM focuses on the similarly gnarled way scripters must go about generating content they want to add or replace on a page. For text-only changes (for example, the text inside a table cell), there is both an easy and a hard way to perform the task. For HTML changes, there is only the hard way (plus a handy workaround discussed later). Let's look at the hard way first and then pick up the easy way for text changes.

To generate a new node in the DOM, you look to the variety of methods that are defined for the Core DOM's `document` object (and are therefore inherited by the HTML `document` object). A node creation method is defined for nearly every node type in the DOM. The two important ones for HTML documents are `createElement()` and `createTextNode()`. The first generates an element with whatever tag name (string) you pass as a parameter; the second generates a text node with whatever text you pass.

When you first create a new element, it exists only in the browser's memory and not as part of the document containment hierarchy. Moreover, the result of the `createElement()` method is a reference to an empty element except for the name of the tag. For example, to create a new `p` element, use

```
var newElem = document.createElement("p");
```

The new element has no ID, attributes, or any content. To assign some attributes to that element, you can use the `setAttribute()` method (a method of every element object) or assign a value to the object's corresponding property. For example, to assign an identifier to the new element, use either

```
newElem.setAttribute("id", "newP");
```

or

```
newElem.id = "newP";
```

Both ways are perfectly legal. Even though the element has an ID at this point, it is not yet part of the document so you cannot retrieve it via the `document.getElementById()` method.

To add some content to the paragraph, you next generate a text node as a separate object:

```
var newText = document.createTextNode("This is the second paragraph.");
```

Again, this node is just sitting around in memory waiting for you to apply it as a child of some other node. To make this text the content of the new paragraph, you can append the node as a child of the paragraph element that is still in memory:

```
newElem.appendChild(newText);
```

If you were able to inspect the HTML that represents the new paragraph element, it would look like the following:

```
<p id="newP">This is the second paragraph.</p>
```

The new paragraph element is ready for insertion into a document. Using the document shown in Listing 14-1, you can append it as a child of the `body` element:

```
document.body.appendChild(newElem);
```

At last, the new element is part of the document containment hierarchy. You can now reference it just like any other element in the document.

Replacing node content

The addition of the paragraph shown in the last section requires a change to a portion of the text in the original paragraph (the first paragraph is no longer the "one and only" paragraph on the page). As mentioned earlier, you can perform text changes either via the `replaceChild()` method or by assigning new text to a text node's `nodeValue` property. Let's see how each approach works to change the text of the first paragraph's `em` element from "one and only" to "first."

To use `replaceChild()`, a script must first generate a valid text node with the new text:

```
var newText = document.createTextNode("first");
```

The next step is to use the `replaceChild()` method. But recall that the point of view for this method is the parent of the child being replaced. The child here is the text node inside the `em` element, so you must invoke the `replaceChild()` method on the `em` element. Also, the `replaceChild()` method requires two parameters: the first is the new node; the second is a reference to the node to be replaced. Because the script statements get pretty long with the `getElementById()` method, an intermediate step grabs a reference to the text node inside the `em` element:

```
var oldChild = document.getElementById("emphasis1").childNodes[0];
```

Now the script is ready to invoke the `replaceChild()` method on the `em` element, swapping the old text node with the new:

```
document.getElementById("emphasis1").replaceChild(newText, oldChild);
```

If you want to capture the old node before it disappears entirely, be aware that the `replaceChild()` method returns a reference to the replaced node (which is only in memory at this point, and not part of the document node hierarchy). You can assign the method statement to a variable and use that old node somewhere else, if needed.

This may seem like a long way to go; it is, especially if the HTML you are generating is complex. Fortunately, you can take a simpler approach for replacing text nodes. All it requires is a reference to the text node being replaced. You can assign that node's `nodeValue` property its new string value:

```
document.getElementById("emphasis1").childNodes[0].nodeValue = "first";
```

When an element's content is entirely text (for example, a table cell that already has a text node in it), this is the most streamlined way to swap text on the fly using W3C DOM syntax. This doesn't work for the creation of the second paragraph text earlier in this chapter because the text node did not exist yet. The `createTextNode()` method had to explicitly create it.

Also remember that a text node does not have any inherent style associated with it. The style of the containing HTML element governs the style of the text. If you want to change not only the text node's text but also how it looks, you have to modify the `style` property of the text node's parent element. Browsers that perform these kinds of content swaps and style changes automatically reflow the page to accommodate changes in the size of the content.

To summarize, Listing 14-2 is a live version of the modifications made to the original document shown in Listing 14-1. The new version includes a button and script that makes the changes described throughout this discussion of nodes. Reload the page to start over.

Listing 14-2: **Adding/Replacing DOM Content**

```
<html>
<head>
   <title>A Simple Page</title>
   <script type="text/javascript">
   function modify() {
      var newElem = document.createElement("p");
      newElem.id = "newP";
      var newText = document.createTextNode("This is the second paragraph.");
      newElem.appendChild(newText);
      document.body.appendChild(newElem);
      document.getElementById("emphasis1").childNodes[0].nodeValue = "first";
   }
   </script>
</head>

<body>
   <button onclick="modify()">Add/Replace Text</button>

   <p id="paragraph1">This is the <em id="emphasis1">one and
   only</em> paragraph on the page.</p>
</body>
</html>
```

Chapter 15 details node properties and methods that are inherited by all HTML elements. Most are implemented in both IE5+ and W3C DOM browsers. Also look to the reference material for the `document` object in Chapter 18 for other valuable W3C DOM methods.

A de facto standard: innerHTML

Microsoft was the first to implement the `innerHTML` property of all element objects starting with IE4. While the W3C DOM has not supported this property, scripters frequently find it more convenient to modify content dynamically by way of a string containing HTML markup, rather than creating and assembling element and text nodes. As a result, most modern W3C DOM browsers, including Moz1 and Safari1, support the read/write `innerHTML` property of all element objects as a de facto standard.

When you assign a string containing HTML markup to the `innerHTML` of an existing element, the browser automatically inserts the newly rendered elements into the document node tree. You may also use `innerHTML` with unmarked text to perform the equivalent of the IE-only `innerText` property.

Despite the apparent convenience of the `innerHTML` property compared to the step-by-step process of manipulating element and text node objects, browsers operate on nodes much more efficiently than on assembly of long strings. This is one case where less JavaScript code does not necessarily translate to greater efficiency.

Static W3C DOM HTML objects

The Moz1 DOM (but unfortunately not IE5+) adheres to the core JavaScript notion of prototype inheritance with respect to the object model. When a page loads into Moz1, the browser creates HTML objects based on the prototypes of each object defined by the W3C DOM. For example, if you use The Evaluator Sr. (Chapter 13) to see what kind of object the `myP` paragraph object is (enter `document.getElementById("myP")` into the top text box and click the Evaluate button), it reports that the object is based on the `HTMLParagraphElement` object of the DOM. Every "instance" of a `p` element object in the page inherits its default properties and methods from `HTMLParagraphElement` (which, in turn, inherits from `HTMLElement`, `Element`, and `Node` objects — all detailed in the JavaScript binding appendix of the W3C DOM specification).

You can use scripting to add properties to the prototypes of some of these static objects. To do so, you must use new features added to Moz1. Two new methods — `__defineGetter__()` and `__defineSetter__()` — enable you to assign functions to a custom property of an object.

These methods are Mozilla-specific. To prevent their possible collision with standardized implementations of these features in future implementations of ECMAScript, the underscore characters on either side of the method name are pairs of underscore characters.

The functions execute whenever the property is read (the function assigned via the `__defineGetter__()` method) or modified (the function assigned via the `__defineSetter__()` method). The common way to define these functions is in the form of an anonymous function (see Chapter 33). The formats for the two statements that assign these behaviors to an object prototype are as follows:

```
object.prototype.__defineGetter__("propName", function([param1[,...[,paramN]]])
{
  // statements
  return returnValue;
})
```

```
object.prototype.__defineSetter__("propName", function([param1[,...[,paramN]]])
{
  // statements
  return returnValue;
})
```

The example in Listing 14-3 demonstrates how to add a read-only property to every HTML element object in the current document. The property, called `childNodeDetail`, returns an object; the object has two properties, one for the number of element child nodes and one for the number of text child nodes. Note that the `this` keyword in the function definition is a reference to the object for which the property is calculated. And because the function runs each time a script statement reads the property, any scripted changes to the content after the page loads are reflected in the returned property value.

Listing 14-3: **Adding a Read-Only Prototype Property to All HTML Element Objects**

```
<script type="text/javascript">
if (HTMLElement) {
   HTMLElement.prototype.__defineGetter__("childNodeDetail", function() {
      var result = {elementNodes:0, textNodes:0 }
      for (var i = 0; i < this.childNodes.length; i++) {
         switch (this.childNodes[i].nodeType) {
            case 1:
               result.elementNodes++;
               break;
            case 3:
               result.textNodes++;
               break;
         }
      }
      return result;
   })
}
</script>
```

To access the property, use it like any other property of the object. For example:

```
var BodyNodeDetail = document.body.childNodeDetail;
```

The returned value in this example is an object, so you use regular JavaScript syntax to access one of the property values:

```
var BodyElemNodesCount = document.body.childNodeDetail.elementNodes;
```

Bidirectional event model

Despite the seemingly conflicting event models of NN4 (trickle down) and IE4 (bubble up), the W3C DOM event model (defined in Level 2) manages to employ both event propagation models. This gives the scripter the choice of where along an event's propagation path the event gets processed. To prevent conflicts with existing event model terminology, the W3C model invents many new terms for properties and methods for events. Some coding probably requires W3C DOM–specific handling in a page aimed at multiple object models.

The W3C event model also introduces a new concept called the *event listener*. An event listener is essentially a mechanism that instructs an object to respond to a particular kind of event — very much like the way the event handler attributes of HTML tags respond to events. But the DOM recommendation points out that it prefers use of a more script-oriented way of assigning event listeners: the `addEventListener()` method available for every node in the document hierarchy. Through this method, you advise the browser whether to force an event to bubble up the hierarchy (the default behavior that is also in effect if you use the HTML attribute type of event handler) or to be captured at a higher level.

Functions invoked by the event listener receive a single parameter consisting of the event object whose properties contain contextual details about the event (details such as the position of a mouse click, character code of a keyboard key, or a reference to the target object). For example, if a form includes a button whose job is to invoke a calculation function, the W3C DOM prefers the following way of assigning the event handler:

```
document.getElementById("calcButton").addEventListener("click", doCalc, false);
```

The `addEventListener()` method takes three parameters. The first parameter is a string of the event to listen for; the second is a reference to the function to be invoked when that event fires; and the third parameter is a Boolean value. When you set this Boolean value to `true`, it turns on event capture whenever this event is directed to this target. The function then takes its cue from the event object passed as the parameter:

```
function doCalc(evt) {
   // get shortcut reference to input button's form
   var form = evt.target.form;
   var results = 0;
   // other statements to do the calculation //
   form.result.value = results;
}
```

To modify an event listener, you use the `removeEventListener()` method to get rid of the old listener and then employ `addEventListener()` with different parameters to assign the new one.

Preventing an event from performing its default action is also a different procedure in the W3C event model than in IE. In IE4 (as well as NN3 and NN4), you can cancel the default action by allowing the event handler to evaluate to `return false`. While this still works in IE5+, Microsoft includes another property of the `window.event` object, called `returnValue`. Setting that property to `false` anywhere in the function invoked by the event handler also kills the event before it does its normal job. But the W3C event model uses a method of the event object, `preventDefault()`, to keep the event from its normal task. You can invoke this method anywhere in the function that executes when the event fires.

Detailed information about an event is contained in an event object that must be passed to an event handler function where details may be read. If you assign event handlers via the W3C DOM `addEventListener()` method or an event handler property, the event object is passed automatically as the sole parameter to the event handler function. Include a parameter variable to "catch" the incoming parameter:

```
function swap(evt) {
   // statements here to work with W3C DOM event object
}
```

But if you assign events via a tag attribute, then you must explicitly pass the event object in the call to the function:

```
<a href="http://www.example.com" onmouseover="swap(event)">
```

Unfortunately, neither IE5 through IE6 on Windows nor IE5 for Macintosh implements the W3C DOM event model. You can, however, make the two event models work together if you assign event handlers by way of object properties or tag attributes, and throw in a little object detection described later in this chapter and in more detail in Chapter 25.

Mixing Object Models

The more browsers that your audience uses, the more likely you will want to make your pages work on as many browsers as possible. You've seen in this chapter that scripts written for older browsers, such as Navigator 2 and Internet Explorer 3, tend to work in even the latest browsers without modification. But aiming at that compatibility target doesn't let you take advantage of more advanced features, in particular Dynamic HTML. You must balance the effort required to support as many as four classifications of browsers (non-DHTML, NN4, IE4/5, and W3C DOM common denominator in IE6 and Moz1) against the requirements of your audience. Moreover, those requirements can easily change over time. For example, the share of the audience using non-DHTML and NN4 browsers will diminish over time, while the installed base of browsers capable of using the Microsoft IE DOM (for IE4+) and the W3C DOM (IE6+ and Moz1+) will increase. If the percentage of visitors using NN4 is not significant at this point, you may well decide to not worry about implementing DHTML features for that browser and lump NN4 together with the rest of the non-DHTML browsers.

For any given application or Web site, it is important to develop a strategy to apply to the deployment of scripted features. But be aware that one strategy simply cannot fit all situations. The primary considerations are the breadth of browser versions reaching your site (many for public sites; perhaps only one for a tightly controlled intranet) and the amount of DHTML you intend to implement.

In the rest of this section, you see three scenarios and strategies employed to meet the developer's requirements. Although they are labeled as three different levels of aggressiveness, it is likely that you can apply individual techniques from each of the levels in establishing a strategy of your own.

The conservative approach

In the first scenario, the content requires a modest level of data entry interaction with a user via a form as well as image rollovers. Supported browsers encompass the entire range of non-scriptable and scriptable browsers, with one version of each page to serve all visitors.

If the form gathers information from the user for submission to a server CGI that stores the data in a database or performs a search based on user-supplied criteria, the obvious mode of entry is through traditional form control elements. Scriptable browsers can perform pre-submission validations to hasten the correction of any improperly formatted fields. Event handlers attached to the text fields (`onchange` event handlers) and an `onsubmit` event handler for the form itself can do the validation on the client. Nonscriptable browsers ignore the event handlers, and the form is submitted as usual, relying on server-side validation of input data (and the slow back-and-forth processing that this entails when there is an error or missing field data).

For image rollovers, links surround the image elements. The `onmouseover` and `onmouseout` event handlers for the links trigger functions that swap images. By wrapping the statements in the event handler functions in `if` constructions that test for the presence of the `document.images` array, first-generation scriptable browsers that don't implement images as objects perform no action:

```
function imageOn(imgName) {
   if (document.images) {
      document.images[imgName].src = onImages[imgName].src;
   }
}
```

The same goes for script statements in the Head that precache the swappable images as the page loads:

```
if (document.images) {
   var onImages = new Array();
   onImages["home"] = new Image(50,30);
   onImages["home"].src = "images/homeOn.gif";
   ...
}
```

This scenario can also provide added content on the page for scriptable browser users by embedding scripts within the body that use `document.write()` to generate content as the page loads. For example, the page can begin with a time-sensitive greeting ("Good Morning," "Good Afternoon," and so on), while nonscriptable browser users see a standard greeting inside the `<noscript>` tag pair.

Middle ground

The second scenario includes pages that employ stylesheets. The goal again is to support all browser users with the same HTML pages, but also provide users of modern browsers with an enhanced experience. Where supported by the browser, styles of objects change in response to user action (for example, links highlight with a special font color and background during rollover). One of the design elements on the page is a form within a table. As users enter values into some text boxes, calculated results appear at the bottom of the table, preferably as regular content within a table cell (otherwise in another text box).

This scenario requires browser version branching in several places to allow for variations in browser treatment of the features and to avoid problems with older scriptable browsers and nonscriptable browsers alike. You can (and should) perform some (if not all) of the branching via object detection, as you will see in a moment. Table 14-7 highlights the major feature requirements for this scenario and describes the browser support for each.

Table 14-7: Features and Support for a Typical "Middle Ground" Scenario

Feature	*Support and Approach*
Dynamic Styles	IE4+ and W3C DOM browsers through the `style` property of any HTML element object.
Form Calculations	Unless requiring regular expression parsing of input, should work with all scriptable browsers without any branching required.
Dynamic Content	IE4+ and W3C DOM browsers support Dynamic HTML content within a cell, but MS and W3C object models require different ways of changing a table cell's content. (Or you can use the nonstandard, but convenient, `innerHTML` property of the cell.) For older scriptable browsers, the cell should contain a text box to display the results; for nonscriptable browsers, the cell should contain a button that submits the form to a server CGI to process the calculation and return a new page with the results.

Dynamic styles

For dynamic styles, both the IE4+ and W3C object models provide access to stylesheet settings via the `style` property of any HTML element. This simplifies matters because you can wrap modifications to `style` properties inside `if` clauses that check for the existence of the `style` property for the specified object:

```
function hilite(elem) {
   if (elem.style) {
      elem.style.fontWeight = "bold";
   }
}
```

If the event handler that triggers the change can be localized to the affected element (for example, an `onmouseover` event handler for a `span` element surrounding some text), the event doesn't fire in browsers that don't also support the `style` property. (By good fortune, browsers that implement the `style` property also expose all elements to the object model.) To compensate for the differences in object references between the IE4+ and W3C models, you can pass the object as a parameter to event handler functions:

```
<span onmouseover="hilite(this)" onmouseout="revert(this)"
onclick="go('...')>...</span>
```

This technique obviates the need to use browser version detection because the functions invoked by the event handlers do not have to build DOM-specific references to the objects to adjust the style.

Branching variables

If, for now, you continue to be more comfortable with browser version detection than object detection, you can apply version detection for this "middle ground" scenario by establishing branches for the IE4+ and W3C object models. Global variables that act as flags elsewhere in your page's scripts are still the primary mechanism. For this scenario, you can initialize two global variables as follows:

```
function getIEVersion() {
   var ua = navigator.userAgent;
   var IEoffset = ua.indexOf("MSIE ");
   return parseFloat(ua.substring(IEoffset+5, ua.indexOf(";", Ieoffset)));
}

var isIE4 = ((navigator.appName.indexOf("Microsoft") == 0 &&
   parseInt(getIEVersion()) >= 4));
var isW3C = (document.documentElement) ? true : false;
```

Notice how the `getIEVersion()` function digs out the precise IE version from deep within the `navigator.userAgent` property. Both global variables are Boolean values. While each variable conveys valuable information on its own, the combination of the two reveals even more about the browser environment if necessary. Figure 14-4 shows the truth table for using the AND (`&&`) operator in a conditional clause with both values. For example, if you need a branch that works only in IE4, the `if` clause is

```
if (isIE4 && !isW3C) {...}
```

The overlap between MS and the W3C object models in IE5+ means that you need to determine for each branch which model to use when the script is running. This governs the order of nested `if` conditions when they arise. If you trap for the W3C version first, IE5+ runs the branch containing the W3C DOM syntax.

isIE4	isW3C	isIE4 && isW3C
true	true	IE5+
true	false	IE4 Only
false	true	Non-IE W3C Only
false	false	Older browser

Figure 14-4: Truth table for two browser version ariables with the AND operator.

Dynamic content

Once you have the branching variables in place, your scripts can use them for executing functions invoked by event handlers as well as for scripts that run while the page loads. The importance of the second type comes when you want a page to display one kind of HTML for one class of browsers and other HTML for other classes (or all of the rest). The design for the current scenario calls for a table cell to display the results of a form's calculation in HTML where capable. In lesser scriptable browsers, the results should appear in a text box in the table. Nonscriptable browsers should display a button to submit the form.

In the Body of the page, a script should take over and use `document.write()` for the `td` element that is to show the results. Buggy behavior in early versions of Navigator require that at least the entire `td` element be written dynamically, instead of just the cell's content. The structure of such a form and table is as follows:

```
...
<form name="calculator" action="http://xxx/cgi-bin/calculate.pl"
onsubmit="return false">
<table>
...
<tr>
   <td>...</td>
   <script type="text/javascript">
   if (isIE4 || isW3C) {
      document.write("<td id='result'>0</td>");
   } else {
      document.write("<td>");
      document.write("<input type='text' name='result' size='10' value='0' />");
      document.write("</td>");
   }
   </script>
   <noscript>
      <td>Click 'Submit' for Results</td>
   </noscript>
</tr>
</table>
<noscript>
   <input type="submit" />
</noscript>
</form>
...
```

The preceding code assumes that other table cells contain text boxes whose `onchange` event handlers trigger a calculation script. That calculation script must also branch for the two classes of scriptable browser so that results are displayed to fit the browser's object model:

```
function calculate(form) {
   var results;
   ...
   // statements here that perform math and stuff answer into 'results' variable
   ...
   if (isIE4) {
      document.all.result.innerText = results;
   } else if (isW3C) {
      document.getElementById("result").childNodes[0].nodeValue = results;
   } else {
      document.calculator.result.value = results;
   }
}
```

Adding dynamic content for NN4 requires a little more planning. The technique usually involves nesting an absolute-positioned `div` inside a relative-positioned `span`. Scripts can then use `document.write()` to create new content for the deeply nested `div` element. Pulling this off successfully entails pretty complex references through multiple layers and their documents. But no matter what lengths you go to in an effort to employ dynamic content in NN4, the new content does not automatically resize the table or cell to accommodate larger or smaller chunks of text. Without automatic reflow of the page, as is found in IE4+ and most W3C DOM browsers, writing to an NN4 positioned layer does not force other page content to move.

A radical approach

By "radical," I mean that the page content is designed to employ extensive DHTML features, including positioned (if not flying) elements on the page. Perhaps some clicking and dragging of elements can add some fun to the page while you're at it.

Employing these kinds of features requires some extensive forethought about your audience and the browsers they use. While some aspects of DHTML, such as CSS, degrade gracefully in older browsers (the content is still presented, although not in optimum font display perhaps), positioned elements do not degrade well at all. The problem is that older browsers ignore the CSS attributes that control positioning, stacking order, and visibility. Therefore, when the page loads in a pre-version 4 browser, all content is rendered in source code order. Elements that are supposed to be positioned, hidden, or overlapped are drawn on the page in "old-fashioned" rendering.

To use element positioning for the greatest effect, your Web site should preexamine the browser at some earlier page in the navigation sequence to reach the DHTML-equipped page. Only browsers capable of your fancy features should be allowed to pass onto the "cool" pages. All other browsers get diverted to another page or pathway through your application so they can at least get the information they came for, if not in the most lavish presentation. Techniques detailed in Chapter 13 demonstrate how to make a branching index page.

By filtering out non-DHTML-capable browsers, some of your job is easier — but not all. On the plus side, you can ignore a lot of weirdness that accrues to scripting bugs in earlier browsers. But you must still decide which of the three element positioning models to follow: IE4+, NN4, or W3C. Chances are that you will want to support at least two of the three unless you are in the luxurious position of designing for a single browser platform (or have taken a stand that you will support only one DOM, probably the newer W3C).

Of the three models, NN4's DOM is the trickiest one to deal with at the HTML level. While it may be possible that your content design will look the same using positioned `div` and `span` elements in all DHTML-capable browsers, often the appearance in NN4 is unacceptable. At that point, you will probably have to use scripts in your Body to dynamically generate HTML, specifying the `<layer>` tag for NN4 and positioned `<div>` elements for the rest.

Although IE4+ can use the same basic Microsoft object model, not all DHTML code renders the same on both generations of browsers. Microsoft made some changes here and there to the way some style attributes are rendered so that IE5.*x* comes into better compliance with the CSS recommendation, while IE6 is in full compliance with CSS Level 1 when you specify a complete DOCTYPE element pointing to URLs of the DTDs for HTML 4.0 or later and XHTML.

Using script libraries

As long as you plan to use scripts to dynamically generate HTML for the page, you might consider creating separate, external `.js` libraries for each of the object models you want to support for the page. Scripts in each library contain code for both the HTML accumulation (for use with `document.write()` in the main page) and for processing user interaction. Assuming that only DHTML-capable browsers reach the page, branching is required only at the beginning of the document where an object model–specific library is loaded:

```
var isIE4 = ((navigator.appName.indexOf("Microsoft") == 0 &&
   parseInt(navigator.appVersion) == 4));
var isW3C = (document.documentElement) ? true : false;
if (isW3C) {
   // give priority to W3C model for IE5+
   document.write("<script type='text/javascript' src='page3_W3C.js'><" +
      "\/script>");
} else if (isIE4) {
   document.write("<script type='text/javascript' src='page3_IE4.js'><" +
      "\/script>");
} else {
   document.write("<script type='text/javascript' src='page3_generic.js'><" +
      "\/script>");
}
```

Each of the statements that writes the `<script>` tag includes a workaround that is required on some browsers (NN4 especially) to facilitate using `document.write()` to write script tags to the page.

Once these libraries are specified for the page, script statements anywhere later in the page can invoke functions defined in each library to generate a particular element or set of elements in the object model HTML optimized for the current browser. Of course, it's not necessary to have one library devoted to each object model. You might find it more convenient for authoring and maintenance to keep all the code in one library that has numerous internal branchings for browser versions. Branches in a library can use the version-sniffing global variables defined in the main HTML page's scripts. Better still, a library can be entirely self-contained by using object detection.

Handling events

Thanks to the W3C DOM's event model implementing a similar event bubbling scheme as IE4+, you can apply that event propagation model to IE4+ and W3C DOM browsers. There are differences in the details, however. IE's approach does not pass the `event` object as a parameter to a function invoked by an event handler. Instead, the IE `event` object is a property of the

window object. Therefore, your functions have to look for the passed parameter and substitute the window.event object in its place for IE:

```
function calculate(evt) {
   evt = (evt) ? evt : window.event;
   // more statements to handle the event //
}
```

Additional branching is necessary to inspect many details of the event. For example, IE calls the object receiving the event the srcElement, while the W3C DOM calls it the target. Canceling the default behavior of the event (for example, preventing a form's submission if it fails client-side validation) is also different for the models (although the "old-fashioned" way of letting HTML-type event handlers evaluate to return false still works). You can find more event object details in Chapter 25.

Standards Compatibility Modes (DOCTYPE Switching)

Both Microsoft and Netscape/Mozilla discovered that they had, over time, implemented CSS features in ways that ultimately differed from the published standards that came later (usually after much wrangling among working group members). To compensate for these differences and make a clean break to be compatible with the standards, the major browser makers decided to let the page author's choice of <!DOCTYPE> header element details determine whether the document was designed to follow the old way (sometimes called "quirks mode") or the standards-compatible way. The tactic, known informally as DOCTYPE switching, is implemented in WinIE6, MacIE5, and all Mozilla-based browsers.

While most of the differences between the two modes are small, there are some significant differences between the two modes in WinIE6, particularly when styles or Dynamic HTML scripts rely on elements designed with borders, margins, and padding. Microsoft's original "box model" measured the dimensions of elements in a way that differed from the eventual CSS standard.

To place the affected browsers into CSS standards-compatible mode, you should include a <!DOCTYPE> element at the top of every document that specifies any of the following details:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Frameset//EN"
        "http://www.w3.org/TR/REC-html40/frameset.dtd">

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
        "http://www.w3.org/TR/REC-html40/strict.dtd">

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
        "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

Be aware, however, that older versions of WinIE, such as WinIE5 or WinIE5.5, are ignorant of the standards-compatible mode, and will use the old Microsoft quirks mode, regardless of `<!DOCTYPE>` setting. But using the standards-compatible mode DOCTYPE is more likely to force your content and stylesheets to render more similarly across the latest browsers.

Where to Go from Here

These past two chapters provided an overview of the core language and object model issues that anyone designing pages that use JavaScript must confront. The goal here is to stimulate your own thinking about how to embrace or discard levels of compatibility with your pages as you balance your desire to generate "cool" pages and serve your audience. From here on, the difficult choices are up to you.

To help you choose the objects, properties, methods, and event handlers that best suit your requirements, the rest of the chapters in Part III and all of Part IV provide in-depth references to the document object model and core JavaScript language features. Observe the compatibility ratings for each language term very carefully to help you determine which features best suit your audience's browsers. Most example listings are complete HTML pages that you can load in various browsers to see how they work. Many others invite you to explore how things work via The Evaluator Sr. (see Chapter 13). Play around with the files, making modifications to build your own applications or expanding your working knowledge of JavaScript in the browser environment.

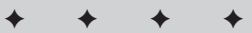
The language and object models have grown in the handful of years they have been in existence. The amount of language vocabulary has increased astronomically. It takes time to drink it all in and feel comfortable that you are aware of the powers available to you. Don't worry about memorizing the vocabulary. It's more important to acquaint yourself with the features, and then come back later when you need the implementation details.

Be patient. Be persistent. The reward will come.

✦ ✦ ✦

CHAPTER 15

Generic HTML Element Objects

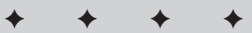


In This Chapter

Working with HTML element objects

Common properties and methods

Event handlers of all element objects



The object model specifications implemented in Internet Explorer 4+ and Mozilla-based browsers (including Netscape 6 and 7) feature a large set of scriptable objects that represent what we often call "generic" HTML elements. Generic elements can be divided into two groups. One group, such as the `b` and `strike` elements, define font styles to be applied to enclosed sequences of text. The need for these elements (and the objects that represent them) is receding as more page designers use stylesheets. The second group of elements assigns context to content within their start and end tags. Examples of contextual elements include `h1`, `blockquote`, and the ubiquitous `p` element. While browsers sometimes have consistent visual ways of rendering contextual elements by default (for example, the large, bold font of an `<h1>` tag), the specific rendering is not the intended purpose of the tags. No formal standard dictates that text within an `em` element must be italicized: the style simply has become the custom since the very early days of browsers.

All of these generic elements share a large number of scriptable properties, methods, and event handlers. The sharing extends not only among generic elements, but also among virtually every renderable element — even if it has additional, element-specific properties, methods, and/or event handlers that I cover in depth in other chapters of this reference. Rather than repeat the details of these shared properties, methods, and event handlers for each object throughout this reference, I describe them in detail only in this chapter (unless there is a special behavior, bug, or trick associated with the item in some object described elsewhere). In succeeding reference chapters, each object description includes a list of the object's properties, methods, and event handlers, but I do not list shared items over and over (making it hard to find items that are unique to a particular element). Instead, you see a pointer back to this chapter for the items in common with generic HTML element objects.

Generic Objects

Table 15-1 lists all of the objects that I treat in this reference as "generic" objects. All of these objects share the properties, methods, and event handlers described in succeeding sections and have no special items that require additional coverage elsewhere in this book.

Table 15-1: Generic HTML Element Objects

Formatting Objects	*Contextual Objects*
b	acronym
big	address
center	cite
i	code
nobr	dfn
rt	del
ruby	div
s	em
small	ins
strike	kbd
sub	listing
sup	p
tt	plaintext
u	pre
wbr	samp
	span
	strong
	var
	xmp

Properties	*Methods*	*Event Handlers*
accessKey	addBehavior()	onactivate
all[]	addEventListener()	onbeforecopy
attributes[]	appendChild()	onbeforecut
behaviorUrns[]	applyElement()	onbeforedeactivate
canHaveChildren	attachEvent()	onbeforeeditfocus
canHaveHTML	blur()	onbeforepaste
childNodes[]	clearAttributes()	onblur
children	click()	onclick
className	cloneNode()	oncontextmenu
clientHeight	componentFromPoint()	oncontrolselect
clientLeft	contains()	oncopy
clientTop	detachEvent()	oncut
clientWidth	dispatchEvent()	ondblclick
contentEditable	fireEvent()	ondeactivate
currentStyle	focus()	ondrag
dataFld	getAdjacentText()	ondragend
dataFormatAs	getAttribute()	ondragenter
dataSrc	getAttributeNode()	ondragleave

elementObject

Properties	*Methods*	*Event Handlers*
dir	getAttributeNodeNS()	ondragover
disabled	getAttributeNS()	ondragstart
document	getBoundingClientRect()	ondrop
filters[]	getClientRects()	onfilterchange
firstChild	getElementsByTagName()	onfocus
height	getElementsByTagNameNS()	onhelp
hideFocus	getExpression()	onkeydown
id	hasAttribute()	onkeypress
innerHTML	hasAttributeNS()	onkeyup
innerText	hasAttributes()	onlosecapture
isContentEditable	hasChildNodes()	onmousedown
isDisabled	insertAdjacentElement()	onmouseenter
isMultiLine	insertAdjacentHTML()	onmouseleave
isTextEdit	insertAdjacentText()	onmousemove
lang	insertBefore()	onmouseout
language	item()	onmouseover
lastChild	isSupported()	onmouseup
length	mergeAttributes()	onpaste
localName	normalize()	onpropertychange
namespaceURI	releaseCapture()	onreadystatechange
nextSibling	removeAttribute()	onresize
nodeName	removeAttributeNode()	onresizeend
nodeType	removeAttributeNS()	onresizestart
nodeValue	removeBehavior()	onselectstart
offsetHeight	removeChild()	
offsetLeft	removeEventListener()	
offsetParent	removeExpression()	
offsetTop	removeNode()	
offsetWidth	replaceAdjacentText()	
outerHTML	replaceChild()	
outerText	replaceNode()	
ownerDocument	scrollIntoView()	
parentElement	setActive()	
parentNode	setAttribute()	
parentTextEdit	setAttributeNode()	
prefix	setAttributeNodeNS()	
previousSibling	setAttributeNS()	
readyState	setCapture()	
recordNumber	setExpression()	
runtimeStyle	swapNode()	
scopeName	tags()	
scrollHeight	urns()	

Continued

elementObject

Properties	*Methods*	*Event Handlers*
scrollLeft		
scrollTop		
scrollWidth		
sourceIndex		
style		
tabIndex		
tagName		
tagUrn		
title		
uniqueID		

Syntax

To access element properties or methods, use this:

```
(IE4+)      [document.all.]objectID.property | method([parameters])
(IE5+/W3C)  document.getElementById(objectID).property | method([parameters])
```

About these objects

All objects listed in Table 15-1 are DOM representations of HTML elements that influence either the font style or the context of some HTML content. The large set of properties, methods, and event handlers associated with these objects also applies to virtually every other DOM object that represents an HTML element. Discussions about object details in this chapter apply to dozens of other objects described in succeeding chapters of this reference section.

Properties

accessKey

Value: One-character string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

For many elements, you can specify a keyboard character (letter, numeral, or punctuation symbol) that, when typed as an Alt+key combination (on the Win32 OS platform) or Ctrl+key combination (on the MacOS), brings focus to that element. An element that has focus is the one that is set to respond to keyboard activity. If the newly focused element is out of view in the document's current scroll position, the document is scrolled to bring that focused element into view (also see the `scrollIntoView()` method). The character you specify can be an uppercase or lowercase value, but these values are not case-sensitive. If you assign the same letter to more than one element, the user can cycle through all elements associated with that `accessKey` value.

Internet Explorer gives some added powers to the `accessKey` property in some cases. For example, if you assign an `accessKey` value to a `label` element object, the focus is handed to the form element associated with that label. Also, when elements such as buttons have focus, pressing the spacebar acts the same as clicking the element with a mouse.

Although W3C DOM browsers, such as Mozilla and Safari, expose this property for some element types (notably `a`, `area`, `button`, `input`, `label`, `legend`, and `textarea`), these browsers do not respond to scripted changes of the property.

Exercise some judgment in selecting characters for `accessKey` values. If you assign a letter that is normally used to access one of the Windows version browser's built-in menus (for example, Alt+F for the File menu), that `accessKey` setting overrides the browser's normal behavior. To users who rely on keyboard access to menus, your control over that key combination can be disconcerting.

Example

Listing 15-1 shows an example of how to use the `accessKey` property to manipulate the keyboard interface for navigating a Web page. When you load the script in Listing 15-1, adjust the height of the browser window so that you can see nothing below the second dividing rule. Enter any character into the Settings portion of the page and press Enter. (The Enter key may cause your computer to beep.) Then hold down the Alt (Windows) or Ctrl (Mac) key while pressing the same keyboard key. The element from below the second divider should come into view.

Listing 15-1: **Controlling the accessKey Property**

```
<html>
   <head>
      <title>accessKey Property</title>
      <script type="text/javascript">
      function assignKey(type, elem) {
         if (window.event.keyCode == 13) {
            switch (type) {
               case "button":
                  document.forms["output"].access1.accessKey = elem.value;
                  break;
               case "text":
                  document.forms["output"].access2.accessKey = elem.value;
                  break;
               case "table":
                  document.getElementById("myTable").accessKey = elem.value;
            }
            return false;
         }
      }
      </script>
   </head>
   <body>
      <h1>accessKey Property Lab</h1>
      <hr />
      Settings:<br />
      <form name="input">
         Assign an accessKey value to the Button below and press Return: <input
         type="text" size="2" maxlength="1"
         onkeypress="return assignKey('button', this)" /><br />
          Assign an accessKey value to the Text Box below and press Return:
         <input type="text" size="2" maxlength="1"
         onkeypress="return assignKey('text', this)" /><br />
          Assign an accessKey value to the Table below (IE5.5+ only) and press
```

Continued

Listing 15-1 *(continued)*

```
         Return: <input type="text" size="2" maxlength="1"
         onkeypress="return assignKey('table', this)" />
      </form>
      <br />
       Then press Alt (Windows) or Control (Mac) + the key.<br />
       <em>Size the browser window to view nothing lower than this line.</em>
      <hr />
      <form name="output" onsubmit="return false">
         <input type="button" name="access1" value="Standard Button" /> <input
         type="text" name="access2" />
      </form>
      <table id="myTable" cellpadding="10" border="2">
         <tr>
            <th>Quantity</th>
            <th>Description</th>
            <th>Price</th>
         </tr>
         <tbody bgcolor="red">
            <tr>
               <td width="100">4</td>
               <td>Primary Widget</td>
               <td>$14.96</td>
            </tr>
            <tr>
               <td>10</td>
               <td>Secondary Widget</td>
               <td>$114.96</td>
            </tr>
         </tbody>
      </table>
   </body>
</html>
```

Related Item: `scrollIntoView()` method.

`all[]`

Value: Array of nested element objects. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `all` property is a collection (array) of every HTML element and (in IE5+) XML tag within the scope of the current object. Items in this array appear in source-code order, and the array is oblivious to element containment among the items. For HTML element containers, the source-code order is dependent on the position of the start tag for the element — end tags are not counted. But for XML tags, end tags appear as separate entries in the array.

Every `document.all` collection contains objects for the `html`, `head`, `title`, and `body` element objects even if the actual HTML source code omits the tags. The object model creates these objects for every document that is loaded into a window or frame. While the `document.all` reference may be the most common usage, the `all` property is available for any container element. For example, `document.forms[0].all` exposes all elements defined within the first form of a page.

You can access any element that has an identifier assigned to its `id` attribute by that identifier in string form (as well as by index integer). Rather than use the performance-costly `eval()` function to convert a string to an object reference, use the string value of the name as an array index value:

```
var paragraph = document.all["myP"];
```

Internet Explorer enables you to use either square brackets or parentheses for single collection index values. Thus, the following two examples evaluate identically:

```
var paragraph = document.all["myP"];
var paragraph = document.all("myP");
```

In the rare case that more than one element within the `all` collection has the same ID, the syntax for the string index value returns a collection of just those identically named elements. But you can use a second argument (in parentheses) to signify the integer of the initial collection and thus single out a specific instance of that named element:

```
var secondRadio = document.all("group0",1);
```

As a more readable alternative, you can use the `item()` method (described later in this chapter) to access the same kinds of items within a collection:

```
var secondRadio = document.all.item("group0",1);
```

Also see the `tags()` method (later in this chapter) as a way to extract a set of elements from an `all` collection that matches a specific tag name.

Although a few non-IE browsers support the `all` collection, a better choice for future compatibility is the `document.getElementById()` method described in Chapter 18.

Example

Use The Evaluator (Chapter 13) to experiment with the `all` collection. Enter the following statements one at a time into the lower text box, and review the results in the text area for each.

```
document.all
myTable.all
myP.all
```

If you encounter a numbered element within a collection, you can explore that element to see which tag is associated with it. For example, if one of the results for the `document.all` collection says `document.all.8=[object]`, enter the following statement into the topmost text box:

```
document.all[8].tagName
```

Related Items: `item()`, `tags()`, `document.getElementById()` methods.

attributes[]

Value: Array of attribute object references. Read-Only
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The `attributes` property consists of an array of attributes specified for an element. In IE5+, the `attributes` array contains an entry for every possible property that the browser has defined for its elements — even if the attribute is not set explicitly in the HTML tag. Also, any attributes that you add later via script facilities such as the `setAttribute()` method are not reflected in the `attributes` array. In other words, the IE5+ `attributes` array is fixed, using default values for all properties except those that you explicitly set as attributes in the HTML tag.

***elementObject*.attributes**

Mozilla browsers' `attributes` property returns an array that is a named node map (in W3C DOM terminology) — an object that has its own properties and methods to read and write attribute values. For example, you can use the `getNamedItem(attrName)` and `item(index)` methods on the array returned from the `attributes` property to access individual attribute objects via W3C DOM syntax.

IE5+ and Moz1+ have different ideas about what an attribute object should be. Table 15-2 shows the variety of properties of an attribute object as defined by the two object models. The larger set of properties in Moz1+ reveals its dependence on the W3C DOM node inheritance model discussed in Chapter 14.

Table 15-2: Attribute Object Properties

Property	*IE5+*	*Moz1+*	*Description*
`attributes`	No	Yes	Array of nested attribute objects (null)
`childNodes`	No	Yes	Child node array
`firstChild`	No	Yes	First child node
`lastChild`	No	Yes	Last child node
`localName`	No	Yes	Name within current namespace
`name`	No	Yes	Attribute name
`nameSpaceURI`	No	Yes	XML namespace URI
`nextSibling`	No	Yes	Next sibling node
`nodeName`	Yes	Yes	Attribute name
`nodeType`	No	Yes	Node type (2)
`nodeValue`	Yes	Yes	Value assigned to attribute
`ownerDocument`	No	Yes	`document` object reference
`ownerElement`	No	Yes	Element node reference
`parentNode`	No	Yes	Parent node reference
`prefix`	No	Yes	XML namespace prefix
`previousSibling`	No	Yes	Previous sibling node
`specified`	Yes	Yes	Whether attribute is explicitly specified (Boolean)
`value`	No	Yes	Value assigned to attribute

The most helpful property of an attribute object is the Boolean `specified` property. In IE, this lets you know whether the attribute is explicitly specified in the element's tag. Because Moz1 returns only explicitly specified attributes in the `attributes` array, the value in Moz1 is always `true`. Most of the time, however, you'll probably use an element object's `getAttribute()` and `setAttribute()` methods to read and write attribute values.

Example

Use The Evaluator (Chapter 13) to examine the values of the `attributes` array for some of the elements in that document. Enter each of the following expressions into the lower text field, and see the array contents in the Results text area for each:

```
document.body.attributes
document.getElementById("myP").attributes
document.getElementById("myTable").attributes
```

If you have both IE5+ and a W3C DOM-compatible browser, compare the results you get for each of these expressions. To view the properties of a single attribute in WinIE5+, enter the following statement into the bottom text field:

```
document.getElementById("myP").attributes["class"]
```

For W3C browsers and MacIE5, use the W3C DOM syntax:

```
document.getElementById("myP").attributes.getNamedItem("class")
```

Related Items: `getAttribute()`, `mergeAttributes()`, `removeAttribute()`, `setAttribute()` methods.

behaviorUrns[]

Value: Array of behavior URN strings. Read-Only
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `behaviorUrns` property is designed to provide a list of addresses, in the form of *URNs (Uniform Resource Names),* of all behaviors assigned to the current object. If there are no behaviors, the array has a length of zero. In practice, however, IE5+ always returns an array of empty strings. Perhaps the potential exposure of URNs by script was deemed to be a privacy risk.

Example

The following function is embedded within a more complete example of WinIE HTML behaviors (Listing 15-19 in this chapter). It reports the length of the `behaviorUrns` array and shows — if the values are returned — the URL of the attached behavior.

```
function showBehaviors() {
   var num = document.getElementById("myP").behaviorUrns.length;
   var msg = "The myP element has " + num + " behavior(s). ";
   if (num > 0) {
      msg += "Name(s): \r\n";
      for (var i = 0; i < num; i++) {
         msg += document.getElementById("myP").behaviorUrns[i] + "\r\n";
      }
   }
   alert(msg);
}
```

Related Item: `urns()` method.

canHaveChildren

Value: Boolean. Read-Only
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

Useful in some dynamic content situations, the `canHaveChildren` property reveals whether a particular element is capable of containing a child (nested) element. Most elements that have start and end tags (particularly the generic elements covered in this chapter) can contain nested elements. A nested element is referred to as a *child* of its parent container.

Example

Listing 15-2 shows an example of how to use the `canHaveChildren` property to visually identify elements on a page that can have nested elements. This example uses color to demonstrate the difference between an element that can have children and one that cannot. The first button sets the `color` style property of every visible element on the page to red.

Thus, elements (including the normally non-childbearing ones such as `hr` and `input`) are affected by the color change. But if you reset the page and click the largest button, only those elements that can contain nested elements receive the color change.

Listing 15-2: **Reading the canHaveChildren Property**

```
<html>
   <head>
      <title>canHaveChildren Property</title>
      <script type="text/javascript">
      function colorAll() {
         var elems = document.getElementsByTagName("*");
         for (var i = 0; i < elems.length; i++) {
            elems[i].style.color = "red";
         }
      }

      function colorChildBearing() {
         var elems = document.getElementsByTagName("*");
         for (var i = 0; i < elems.length; i++) {
            if (elems[i].canHaveChildren) {
               elems[i].style.color = "red";
            }
         }
      }
      </script>
   </head>
   <body>
      <h1>canHaveChildren Property Lab</h1>
      <hr />
      <form name="input">
         <input type="button" value="Color All Elements"
         onclick="colorAll()" /><br />
          <input type="button" value="Reset" onclick="history.go(0)" /><br />
          <input type="button"
         value="Color Only Elements That Can Have Children"
         onclick="colorChildBearing()" />
      </form>
      <br />
      <hr />
      <form name="output">
         <input type="checkbox" checked="checked" />Your basic checkbox <input
         type="text" name="access2" value="Some textbox text." />
      </form>
      <table id="myTable" cellpadding="10" border="2">
         <tr>
            <th>Quantity</th>
            <th>Description</th>
            <th>Price</th>
         </tr>
         <tbody>
            <tr>
               <td width="100">4</td>
               <td>Primary Widget</td>
               <td>$14.96</td>
```

```
            </tr>
            <tr>
               <td>10</td>
               <td>Secondary Widget</td>
               <td>$114.96</td>
            </tr>
         </tbody>
      </table>
   </body>
</html>
```

Related Items: childNodes, firstChild, lastChild, parentElement, parentNode properties; appendChild(), hasChildNodes(), removeChild() methods.

canHaveHTML

Value: Boolean. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

While most HTML elements are containers of HTML content, not all are. The canHaveHTML property lets scripts find out whether a particular object can accept HTML content, such as for insertion or replacement by object methods. The value for a p element, for example, is true. The value for a br element is false.

Example

Use The Evaluator (Chapter 13) to experiment with the canHaveHTML property. Enter the following statements into the top text field and observe the results:

```
document.getElementById("input").canHaveHTML
document.getElementById("myP").canHaveHTML
```

The first statement returns false because an input element (the top text field in this case) cannot have nested HTML. But the myP element is a p element that gladly accepts HTML content.

Related Items: appendChild(), insertAdjacentHTML(), insertBefore() methods.

childNodes[]

Value: Array of node objects. Read-Only
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The childNodes property consists of an array of node objects contained by the current object. Note that child nodes consist of both element objects and text nodes. Therefore, depending on the content of the current object, the number of childNodes and children collections may differ.

If you use the childNodes array in a for loop that iterates through a sequence of HTML (or XML) elements, watch out for the possibility that the browser treats source code whitespace (blank lines between elements and even simple carriage returns between elements) as text nodes. This potential problem affects MacIE5 and Moz1. If present, these extra text nodes occur primarily surrounding block elements.

Most looping activity through the childNodes array aims to examine, count, or modify element nodes within the collection. If that is your script's goal, then test each node returned by

the `childNodes` array, and verify that the `nodeType` property is `1` (element) before processing that node. Otherwise, skip over the node. The skeletal structure of such a loop follows:

```
for (var i = 0; i < myElem.childNodes.length; i++) {
   if (myElem.childNodes[i].nodeType == 1) {
      statements to work on element node i
   }
}
```

The presence of these "phantom" text nodes also impacts the nodes referenced by the `firstChild` and `lastChild` properties, described later in this chapter.

Example

Listing 15-3 contains an example of how you might code a function that "walks" the child nodes of a given node. The `walkChildNodes()` function shown in the listing accumulates and returns a hierarchical list of child nodes from the point of view of the document's HTML element (the default) or any element whose ID you pass as a string parameter. This function is embedded in The Evaluator so that you can inspect the child node hierarchy of that page or (when using `evaluator.js` for debugging as described in Chapter 45 on the CD-ROM) the node hierarchy within any page you have under construction. Try it out in The Evaluator by entering the following statements into the top text field:

```
walkChildNodes()
walkChildNodes(getElementById("myP"))
```

The results of this function show the nesting relationships among all child nodes within the scope of the initial object. It also shows the act of drilling down to further `childNodes` collections until all child nodes are exposed and catalogued. Text nodes are labeled accordingly. The first 15 characters of the actual text are placed in the results to help you identify the nodes when you compare the results against your HTML source code.

Listing 15-3: **Collecting Child Nodes**

```
function walkChildNodes(objRef, n) {
   var obj;
   if (objRef) {
      if (typeof objRef == "string") {
         obj = document.getElementById(objRef);
      } else {
         obj = objRef;
      }
   } else {
      obj = (document.body.parentElement) ?
         document.body.parentElement : document.body.parentNode;
   }
   var output = "";
   var indent = "";
   var i, group, txt;
   if (n) {
      for (i = 0; i < n; i++) {
         indent += "+---";
      }
   } else {
      n = 0;
      output += "Child Nodes of <" + obj.tagName;
```

```
            output += ">\n====================\n";
        }
        group = obj.childNodes;
        for (i = 0; i < group.length; i++) {
            output += indent;
            switch (group[i].nodeType) {
                case 1:
                    output += "<" + group[i].tagName;
                    output += (group[i].id) ? " ID=" + group[i].id : "";
                    output += (group[i].name) ? " NAME=" + group[i].name : "";
                    output += ">\n";
                    break;
                case 3:
                    txt = group[i].nodeValue.substr(0,15);
                    output += "[Text:\"" + txt.replace(/[\r\n]/g,"<cr>");
                    if (group[i].nodeValue.length > 15) {
                        output += "...";
                    }
                    output += "\"]\n";
                    break;
                case 8:
                    output += "[!COMMENT!]\n";
                    break;
                default:
                    output += "[Node Type = " + group[i].nodeType + "]\n";
            }
            if (group[i].childNodes.length > 0) {
              output += walkChildNodes(group[i], n+1);
            }
        }
        return output;
    }
```

Related Items: `nodeName`, `nodeType`, `nodeValue`, `parentNode` properties; `cloneNode()`, `hasChildNodes()`, `removeNode()`, `replaceNode()`, `swapNode()` methods.

children

Value: Array of element objects. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `children` property consists of an array of element objects contained by the current object. Unlike the `childNodes` property, `children` does not take into account text nodes but rather focuses strictly on the HTML (and XML) element containment hierarchy from the point of view of the current object. Children exposed to the current object are immediate children only. If you want to get all element objects nested within the current object (regardless of how deeply nested they are), use the `all` collection instead.

Example

Listing 15-4 shows how you can use the `children` property to "walk" the child nodes of a given node. This function accumulates and returns a hierarchical list of child elements from the point of view of the document's HTML element (the default) or any element whose ID you pass as a string parameter. This function is embedded in The Evaluator so that you can inspect the parent–child hierarchy of that page or (when using `evaluator.js` for debugging as described in Chapter 45 on the CD-ROM) the element hierarchy within any page you have

under construction. Try it out in The Evaluator in IE5+ by entering the following statements into the top text field:

```
walkChildren()
walkChildren("myP")
```

The results of this function show the nesting relationships among all parent and child elements within the scope of the initial object. It also shows the act of drilling down to further `children` collections until all child elements are exposed and catalogued. The element tags also display their `id` and/or `name` attribute values if any are assigned to the elements in the HTML source code.

Listing 15-4: Collecting Child Elements

```
function walkChildren(objRef, n) {
   var obj;
   if (objRef) {
      if (typeof objRef == "string") {
         obj = document.getElementById(objRef);
      } else {
         obj = objRef;
      }
   } else {
      obj = document.body.parentElement;
   }
   var output = "";
   var indent = "";
   var i, group;
   if (n) {
      for (i = 0; i < n; i++) {
         indent += "+---";
      }
   } else {
      n = 0;
      output += "Children of <" + obj.tagName;
      output += ">\n====================\n";
   }
   group = obj.children;
   for (i = 0; i < group.length; i++) {
      output += indent + "<" + group[i].tagName;
      output += (group[i].id) ? " ID=" + group[i].id : "";
      output += (group[i].name) ? " NAME=" + group[i].name : "";
      output += ">\n";
      if (group[i].children.length > 0) {
         output += walkChildren(group[i], n+1);
      }
   }
   return output;
}
```

Related Items: `canHaveChildren`, `firstChild`, `lastChild`, `parentElement` properties; `appendChild()`, `removeChild()`, `replaceChild()` methods.

`className`

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

A *class name* is an identifier that is assigned to the `class` attribute of an element. To associate a CSS rule with several elements in a document, assign the same identifier to the `class` attributes of those elements, and use that identifier (preceded by a period) as the CSS rule's selector. An element's `className` property enables the application of different CSS rules to that element under script control. Listing 15-5 shows an example of such a script.

Example

The style of an element toggles between "on" and "off" in Listing 15-5 by virtue of setting the element's `className` property alternatively to an existing stylesheet class selector name and an empty string. When you set the `className` to an empty string, the default behavior of the `h1` element governs the display of the first header. A click of the button forces the stylesheet rule to override the default behavior in the first `h1` element.

Listing 15-5: Working with the className Property

```
<html>
   <head>
      <title>className Property</title>
      <style type="text/css">
      .special {font-size:16pt; color:red}
      </style>
      <script type="text/javascript">
      function toggleSpecialStyle(elemID) {
         var elem = (document.all) ? document.all(elemID) :
            document.getElementById(elemID);
         if (elem.className == "") {
            elem.className = "special";
         } else {
            elem.className = "";
         }
      }
      </script>
   </head>
   <body>
      <h1>className Property Lab</h1>
      <hr />
      <form name="input">
         <input type="button" value="Toggle Class Name"
         onclick="toggleSpecialStyle('head1')" />
      </form>
      <br />

      <h1 id="head1">ARTICLE I</h1>
      <p>Congress shall make no law respecting an establishment of religion, or
         prohibiting the free exercise thereof; or abridging the freedom of
         speech, or of the press; or the right of the people peaceably to
         assemble, and to petition the government for a redress of
         grievances.</p>
      <h1>ARTICLE II</h1>
```

Continued

Listing 15-5 *(continued)*

```
      <p>A well regulated militia, being necessary to the security of a free
         state, the right of the people to keep and bear arms, shall not be
         infringed.</p>
   </body>
</html>
```

You can also create multiple versions of a style rule with different class selector identifiers and apply them at will to a given element.

Related Items: `rule`, `stylesheet` objects (Chapter 26); `id` property.

clientHeight
clientWidth

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN7, Moz1+, Safari1+

These two properties by and large reveal the pixel height and width of the content within an element whose stylesheet rule includes height and width settings. In theory, these measures do not take into account any margins, borders, or padding that you add to an element by way of stylesheets. In practice, however, different combinations of borders, margins, and padding influence these values in unexpected ways. One of the more reliable applications of the `clientHeight` property enables you to discover, for example, where the text of an overflowing element ends. To read the rendered dimensions of an element, you are better served across browsers with the `offsetHeight` and `offsetWidth` properties.

For the `document.body` object, the `clientHeight` and `clientWidth` properties return the inside height and width of the window or frame (plus or minus a couple of pixels). These take the place of desirable, but nonexistent, window properties in IE.

Internet Explorer 5+ expands the number of objects that employ these properties to include virtually all objects that represent HTML elements. For IE4, these properties apply only to the following objects: `body`, `button`, `caption`, `div`, `embed`, `fieldset`, `legend`, `marquee`, `table`, `td`, `textarea`, `th`, and `tr`.

Values for these properties in Mozilla-based browsers are zero, except for `document.body`, which measures the browser's current content area.

Example

Listing 15-6 for IE includes an example of how to dynamically size content on a page based upon the client area width and height. This example calls upon the `clientHeight` and `clientWidth` properties of a `div` element that contains a paragraph element. Only the width of the `div` element is specified in its stylesheet rule, which means that the paragraph's text wraps inside that width and extends as deeply as necessary to show the entire paragraph. The `clientHeight` property describes that depth. The `clientHeight` property then calculates where a logo image should be positioned immediately after `div`, regardless of the length of the text. As a bonus, the `clientWidth` property helps the script center the image horizontally with respect to the paragraph's text.

Listing 15-6: **Using clientHeight and clientWidth Properties**

```
<html>
   <head>
      <title>clientHeight and clientWidth Properties</title>
      <script type="text/javascript">
      function showLogo() {
         var paragraphW = document.getElementById("myDIV").clientWidth;
         var paragraphH = document.getElementById("myDIV").clientHeight;
         // correct for Windows/Mac discrepancies
         var paragraphTop = (document.getElementById("myDIV").clientTop) ?
            document.getElementById("myDIV").clientTop :
            document.getElementById("myDIV").offsetTop;
         var logoW = document.getElementById("logo").style.pixelWidth;
         // center logo horizontally against paragraph
         document.getElementById("logo").style.pixelLeft =
            (paragraphW-logoW) / 2;
         // position image immediately below end of paragraph
         document.getElementById("logo").style.pixelTop =
            paragraphTop + paragraphH;
         document.getElementById("logo").style.visibility = "visible";
      }
      </script>
   </head>
   <body>
      <button onclick="showLogo()">Position and Show Logo Art</button>
      <div id="logo" style="position:absolute; width:120px; visibility:hidden">
         <img alt="image" src="logo.gif" />
      </div>
      <div id="myDIV" style="width:200px">
         <p>Lorem ipsum dolor sit amet, consectetaur adipisicing elit, sed do
            eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
            adminim veniam, quis nostrud exercitation ullamco laboris nisi ut
            aliquip ex ea commodo consequat. Duis aute irure dolor in
            reprehenderit involuptate velit esse cillum dolore eu fugiat nulla
            pariatur. Excepteur sint occaecat cupidatat non proident.</p>
      </div>
   </body>
</html>
```

To assist in the vertical positioning of the logo, the `offsetTop` property of the `div` object provides the position of the start of the `div` with respect to its outer container (the `body`). Unfortunately, MacIE uses the `clientTop` property to obtain the desired dimension. That measure (assigned to the `paragraphTop` variable), plus the `clientHeight` of the `div`, provides the top coordinate of the image.

Related Items: `offsetHeight`, `offsetWidth` properties.

clientLeft
clientTop

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

***elementObject*.clientLeft**

The purpose and names of the `clientLeft` and `clientTop` properties are confusing at best. Unlike the `clientHeight` and `clientWidth` properties, which apply to the content of an element, the `clientLeft` and `clientTop` properties return essentially no more information than the thickness of a border around an element — provided the element is positioned. If you do not specify a border or do not position the element, the values are zero (although the `document.body` object can show a couple of pixels in each direction without explicit settings). If you are trying to read the left and top coordinate positions of an element, the `offsetLeft` and `offsetTop` properties are more valuable in WinIE; as shown in Listing 15-6, however, the `clientTop` property returns a suitable value in MacIE. Virtually all elements have the `clientLeft` and `clientTop` properties in IE5+; in IE4, the properties apply only to the `body`, `button`, `caption`, `embed`, `fieldset`, `legend`, `marquee`, and `textarea` objects.

Related Items: `offsetLeft`, `offsetTop` properties.

contentEditable

Value: Boolean. Read/Write
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

IE5.5 introduced the concept of editable HTML content on a page. Element tags can include a `contenteditable` attribute, whose value is echoed via the `contentEditable` property of the element. The default value for this property is `inherit`, which means that the property inherits whatever setting this property has in the hierarchy of HTML containers outward to the body. If you set the `contentEditable` property to `true`, that element and all nested elements set to inherit the value become editable; conversely, a setting of `false` turns off the option to edit the content.

Example

Listing 15-7 demonstrates how to use the `contentEditable` property to allow a piece of text to be edited. When you click the button of a freshly loaded page, the `toggleEdit()` function captures the opposite of the current editable state via the `isContentEditable` property of the `div` that is subject to edit. You switch on editing for that element in the next statement by assigning the new value to the `contentEditable` property of the `div`. For added impact, turn the text of the `div` to red to provide additional user feedback about what is editable on the page. You can also switch the button label to one that indicates the action invoked by the next click on it.

Listing 15-7: Using the contentEditable Property

```
<html>
   <head>
      <style type="text/css">
      .normal {color: black}
      .editing {color: red}
      </style>
      <script type="text/javascript">
      function toggleEdit() {
         var newState = !editableText.isContentEditable;
         editableText.contentEditable = newState;
         editableText.className = (newState) ? "editing" : "normal";
         editBtn.innerText = (newState) ? "Disable Editing" : "Enable Editing";
      }
      </script>
      <title>
      </title>
```

```
   </head>
   <body>
      <h1>Editing Contents</h1>
      <hr />
      <p>Turn on editing to modify the following text:</p>
      <div id="editableText">
         Edit this text on the fly....
      </div>
      <p><button id="editBtn" onclick="toggleEdit()"
         onfocus="this.blur()">Enable Editing</button></p>
   </body>
</html>
```

Related Item: `isContentEditable` property.

currentStyle

Value: `style` object. Read-Only
Compatibility: WinIE5+, MacIE5+, NN-, Moz-, Safari-

Every element has style attributes applied to it, even if those attributes are the browser's default settings. Because an element's `style` object reflects only those properties whose corresponding attributes are explicitly set via CSS statements, you cannot use the `style` property of an `element` object to view default style settings applied to an element. That's where the `currentStyle` property comes in.

This property returns a read-only `style` object that contains values for every possible `style` property applicable to the element. If a `style` property is explicitly set via CSS statement or script adjustment, the current reading for that property is also available here. Thus, a script can inquire about any property to determine if it should change to meet some scripted design goal. For example, if you surround some text with an `<em>` tag, the browser by default turns that text into an italic font style. This setting is not reflected in the element's `style` object (`fontStyle` property) because the italic setting was not set via CSS; in contrast, the `element` object's `currentStyle.fontStyle` property reveals the true, current `fontStyle` property of the element as italic.

Example

To change a `style` property setting, access it via the element's `style` object. Use The Evaluator (Chapter 13) to compare the properties of the `currentStyle` and `style` objects of an element. For example, an unmodified copy of The Evaluator contains an `em` element whose ID is `"myEM"`. Enter `document.getElementById("myEM").style` into the bottom property listing text box and press Enter. Notice how most of the property values are empty. Now enter `document.getElementById("myEM").currentStyle` into the property listing text box and press Enter. Every property has a value associated with it.

Related Items: `runtimeStyle`, `style` objects (Chapter 26); `window.getComputedStyle()` for W3C DOM browsers (Chapter 16).

dataFld
dataFormatAs
dataSrc

Value: String. Read/Write
Compatibility: WinIE4+, MacIE5, NN-, Moz-, Safari-

The dataFld, dataFormatAs, and dataSrc properties (along with more element-specific properties such as dataPageSize and recordNumber) are part of the Internet Explorer data-binding facilities based on ActiveX controls. The Win32 versions of IE4 and later have several ActiveX objects built into the browsers that facilitate the direct communication between a Web page and a data source. Data sources include text files, XML data, HTML data, and external databases (MacIE supports text files only). Data binding is a very large topic, much of which extends more to discussions about Microsoft Data Source Objects (DSOs), ODBC, and JDBC — subjects well beyond the scope of this book. But data binding is a powerful tool and can be of use even if you are not a database guru. Therefore, this discussion of the three primary properties — dataFld, dataFormatAs, and dataSrc — briefly covers data binding through Microsoft's *Tabular Data Control DSO*. This allows any page to access, sort, display, and filter (but not update) data downloaded into a Web page from an external text file (commonly comma- or tab-delimited data).

You can load data from an external text file into a document with the help of the Tabular Data Control (TDC). You retrieve the data by specifying the TDC object within an <object> tag set and specifying additional parameters such as the URL of the text file and field delimiter characters. The object element can go anywhere within the body of your document. (I tend to put it at the bottom of the code so that all normal page rendering happens before the control loads.) Retrieving the data simply brings it into the browser and does not, on its own, render the data on the page.

If you haven't worked with embedded objects in IE, the classid attribute value might seem a bit strange. The most perplexing part to some is the long value of numeric data signifying the Globally Unique Identifier (GUID) for the object. You must enter this value exactly as shown in the following example for the proper ActiveX TDC to run. The HTML syntax for this object is as follows:

```
<object id="objName" classid="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83">
   <param name="DataURL" value="URL">
   [additional optional parameters]
</object>
```

Table 15-3 lists the parameters available for the TDC. Only the DataURL parameter is required; others — such as FieldDelim, UseHeader, RowDelim, and EscapeChar — may be helpful depending on the nature of the data source.

Table 15-3: Tabular Data Control Parameters

Parameter	*Description*
CharSet	Character set of the data source file. Default is latin1.
DataURL	URL of data source file (relative or absolute).
EscapeChar	Character used to "escape" delimiter characters that are part of the data. Default is empty. A common value is "\".
FieldDelim	Delimiter character between fields within a record. Default is comma (,). For a Tab character, use a value of .
Language	ISO language code of source data. Default is en-us.
TextQualifier	Optional character surrounding a field's data. Default is empty.
RowDelim	Delimiter character between records. Default is newline (NL).
UseHeader	Set to true if the first row of data in the file contains field names. Default is false.

The value you assign to the `object` element's `id` attribute is the identifier that your scripts use to communicate with the data after the page and data completely load. You can therefore have as many uniquely named TDCs loaded in your page as there are data source files you want to access at once.

The initial binding of the data to HTML elements usually comes when you assign values to the `datasrc` and `datafld` attributes of the elements. The `datasrc` attribute points to the `dso` identifier (matching the `id` attribute of the `object` element, preceded with a hash symbol), while the `datafld` attribute points to the name of the field whose data should be extracted. When you use data binding with an interactive element such as a table, multiple records are displayed in consecutive rows of the table (more about this in a moment).

Adjust the `dataSrc` and `dataFld` properties if you want the same HTML element (other than a table) to change the data that it displays. These properties apply to a subset of HTML elements that can be associated with external data: `a`, `applet`, `body`, `button`, `div`, `frame`, `iframe`, `img`, `input` (most types), `label`, `marquee`, `object`, `param`, `select`, `span`, and `textarea` objects.

In some cases, your data source may store chunks of HTML-formatted text for rendering inside an element. Unless directed otherwise, the browser renders a data source field as plain text — even if the content contains HTML formatting tags. But if you want the HTML to be observed during rendering, you can set the `dataFormatAs` property (or, more likely, the `dataformatas` attribute of the tag) to `HTML`. The default value is `text`.

Example

Listing 15-8 is a simple document that has two TDC objects associated with it. The external files are different formats of the U.S. Bill of Rights document. One file is a traditional, tab-delimited data file consisting of only two records. The first record is a tab-delimited sequence of field names (named `"Article1"`, `"Article2"`, and so on); the second record is a tab-delimited sequence of article content defined in HTML:

```
<h1>ARTICLE I</h1><p>Congress shall make...</p>
```

The second file is a raw text file consisting of the full Bill of Rights with no HTML formatting attached.

When you load Listing 15-8, only the first article of the Bill of Rights appears in a blue-bordered box. Buttons enable you to navigate to the previous and next articles in the series. Because the data source is a traditional, tab-delimited file, the `nextField()` and `prevField()` functions calculate the name of the next source field and assign the new value to the `dataFld` property. All of the data is already in the browser after the page loads, so cycling through the records is as fast as the browser can reflow the page to accommodate the new content.

Listing 15-8: **Binding Data to a Page**

```
<html>
   <head>
      <title>Data Binding</title>
      <style type="text/css">
      #display {width:500px; border:10px ridge blue; padding:20px}
      .hiddenControl {display:none}
      </style>
      <script type="text/javascript">
      function nextField() {
```

Continued

Listing 15-8 *(continued)*

```
            var elem = document.getElementById("display");
            var fieldName = elem.dataFld;
            var currFieldNum = parseInt(fieldName.substring(7,
               fieldName.length),10);
            currFieldNum = (currFieldNum == 10) ? 1 : ++currFieldNum;
            elem.dataFld = "Article" + currFieldNum;
         }
         function prevField() {
            var elem = document.getElementById("display");
            var fieldName = elem.dataFld;
            var currFieldNum = parseInt(fieldName.substring(7,
               fieldName.length),10);
            currFieldNum = (currFieldNum == 1) ? 10 : --currFieldNum;
            elem.dataFld = "Article" + currFieldNum;
         }

         function toggleComplete() {
            if (document.getElementById("buttonWrapper").className == "") {
               document.getElementById("display").dataSrc = "#rights_raw";
               document.getElementById("display").dataFld = "column1";
               document.getElementById("display").dataFormatAs = "text";
               document.getElementById("buttonWrapper").className =
                  "hiddenControl";
            } else {
               document.getElementById("display").dataSrc = "#rights_html";
               document.getElementById("display").dataFld = "Article1";
               document.getElementById("display").dataFormatAs = "HTML";
               document.getElementById("buttonWrapper").className = "";
            }
         }
         </script>
      </head>
      <body>
         <h1>U.S. Bill of Rights</h1>
         <form>
            <input type="button" value="Toggle Complete/Individual"
            onclick="toggleComplete()" /> <span id="buttonWrapper" class=""><input
            type="button" value="Prev" onclick="prevField()" /> <input
            type="button" value="Next" onclick="nextField()" /></span>
         </form>
         <div id="display" datasrc="#rights_html" datafld="Article1"
         dataformatas="HTML">
         </div>
         <object id="rights_html"
         classid="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83">
            <param name="DataURL" value="Bill of Rights.txt" />
            <param name="UseHeader" value="True" />
            <param name="FieldDelim" value="&#09;" />
         </object> <object id="rights_raw"
         classid="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83">
            <param name="DataURL" value="Bill of Rights (no format).txt" />
            <param name="FieldDelim" value="\" />
            <param name="RowDelim" value="\" />
         </object>
      </body>
   </html>
```

Another button on the page enables you to switch between the initial piecemeal version of the document and the unformatted version in its entirety. To load the entire document as a single record, the `FieldDelim` and `RowDelim` parameters of the second `object` element eliminate their default values by replacing them with characters that don't appear in the document at all. And because the external file does not have a field name in the file, the default value (`column1` for the lone column in this document) is the data field. Thus, in the `toggleComplete()` function, the `dataSrc` property is changed to the desired `object` element ID, the `dataFld` property is set to the correct value for the data source, and the `dataFormatAs` property is changed to reflect the different intention of the source content (to be rendered as HTML or as plain text). When the display shows the entire document, you can hide the two radio buttons by assigning a `className` value to the `span` element that surrounds the buttons. The `className` value is the identifier of the class selector in the document's stylesheet. When the `toggleComplete()` function resets the `className` property to empty, the default properties (normal inline display style) take hold.

One further example demonstrates the kind of power available to the TDC under script control. Listing 15-9 displays table data from a tab-delimited file of Academy Award information. The data file has eight columns of data, and each column heading is treated as a field name: Year, Best Picture, Best Director, Best Director Film, Best Actress, Best Actress Film, Best Actor, and Best Actor Film. For the design of the page, only five fields from each record appear: Year, Film, Director, Actress, and Actor. Notice in the listing how the HTML for the table and its content is bound to the data source object and the fields within the data.

The "dynamic" part of this example is apparent in how you can sort and filter the data, once loaded into the browser, without further access to the original source data. The TDC object features `Sort` and `Filter` properties that enable you to act on the data currently loaded in the browser. The simplest kind of sorting indicates on which field (or fields via a semicolon-delimited list of field names) the entire data set should be sorted. Leading the name of the sort field is either a plus (to indicate ascending) or minus (descending) symbol. After setting the `data` object's `Sort` property, invoke its `Reset()` method to tell the object to apply the new property. The data in the bound table is immediately redrawn to reflect any changes.

Similarly, you can tell a data collection to display records that meet specific criteria. In Listing 15-9, two select lists and a pair of radio buttons provide the interface to the `Filter` property's settings. For example, you can filter the output to display only those records in which the Best Picture was the same picture of the winning Best Actress's performance. Simple filter expressions are based on field names:

```
dataObj.Filter = "Best Picture" = "Best Actress Film";
```

Listing 15-9: **Sorting Bound Data**

```
<html>
   <head>
      <title>Data Binding - Sorting</title>
      <script type="text/javascript">
      function sortByYear(type) {
         oscars.Sort = (type == "normal") ? "-Year" : "+Year";
         oscars.Reset();
      }
      function filterInCommon(form) {
         var filterExpr1 =
            form.filter1.options[form.filter1.selectedIndex].value;
         var filterExpr2 =
```

Continued

Listing 15-9 *(continued)*

```
            form.filter2.options[form.filter2.selectedIndex].value;
         var operator = (form.operator[0].checked) ? "=" : "<>";
         var filterExpr = filterExpr1 + operator + filterExpr2;
         oscars.Filter = filterExpr;
         oscars.Reset();
      }
      </script>
   </head>
   <body>
      <h1>Academy Awards 1978-1997</h1>
      <form>
         <p>Sort list by year <a href="javascript:sortByYear('normal')">from
            newest to oldest</a> or <a
            href="javascript:sortByYear('reverse')">from oldest to
            newest</a>.</p>
         <p>Filter listings for records whose <select name="filter1"
            onchange="filterInCommon(this.form)">
               <option value="Best Picture">Best Picture</option>
               <option value="Best Director Film">Best Director's Film</option>
               <option value="Best Actress Film">Best Actress' Film</option>
               <option value="Best Actor Film">Best Actor's Film</option>
            </select> <input type="radio" name="operator" checked="checked"
            onclick="filterInCommon(this.form)" />is <input type="radio"
            name="operator" onclick="filterInCommon(this.form)" />is not
            <select name="filter2" onchange="filterInCommon(this.form)">
               <option value="Best Picture">Best Picture</option>
               <option value="Best Director Film">Best Director's Film</option>
               <option value="Best Actress Film">Best Actress' Film</option>
               <option value="Best Actor Film">Best Actor's Film</option>
            </select></p>
      </form>
      <table datasrc="#oscars" border="1" align="center">
         <thead style="background-color:yellow; text-align:center">
            <tr>
               <td>Year</td>
               <td>Film</td>
               <td>Director</td>
               <td>Actress</td>
               <td>Actor</td>
            </tr>
         </thead>
         <tr>
            <td><div id="col1" datafld="Year"></div></td>
            <td><div id="col2" datafld="Best Picture"></div></td>
            <td><div id="col3" datafld="Best Director"></div></td>
            <td><div id="col4" datafld="Best Actress"></div></td>
            <td><div id="col5" datafld="Best Actor"></div></td>
         </tr>
      </table>
      <object id="oscars" classid="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83">
         <param name="DataURL" value="Academy Awards.txt" />
         <param name="UseHeader" value="True" />
         <param name="FieldDelim" value="&#09;" />
      </object>
   </body>
</html>
```

Related Items: `recordNumber`, `table.dataPageSize` properties.

dir

Value: `"ltr" | "rtl"` Read/Write
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The `dir` property (based on the `dir` attribute of virtually every text-oriented HTML element) controls whether an element's text is rendered left-to-right (the default) or right-to-left. By and large, this property (and HTML attribute) is necessary only when you need to override the default directionality of a language's character set as defined by the Unicode standard.

Example

Changing this property value in a standard U.S. version of a browser only makes the right margin the starting point for each new line of text (in other words, the characters are not rendered in reverse order). You can experiment with this in The Evaluator by entering the following statements into the expression evaluation field:

```
document.getElementById("myP").dir = "rtl"
```

Related Item: `lang` property.

disabled

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

While some elements have a `disabled` property in IE4 and IE5, this property is associated with every HTML element in IE5.5+. W3C DOM browsers apply the property only to form control and `style` element objects. Disabling an HTML element (like form elements) usually gives the element a "dimmed" look, indicating that it is not active. A disabled element does not receive any events. It also cannot receive focus, either manually or by script (although disabled text fields in MacIE4 errantly manage to receive focus). But a user can still select and copy a disabled body text element.

If you disable a form control element, the element's data is not submitted to the server with the rest of the form elements. If you need to keep a form control "locked down," but still submit it to the server, use the `form` element's `onsubmit` event handler to enable the form control right before the form is submitted.

Example

Use The Evaluator (Chapter 13) to experiment with the `disabled` property on both form elements (IE4+ and W3C) and regular HTML elements (WinIE5.5+). For IE4+ and W3C browsers, see what happens when you disable the output text area by entering the following statement into the top text box:

```
document.forms[0].output.disabled = true
```

The text area is disabled for user entry, although you can still set the field's `value` property via script (which is how the `true` returned value got there).

If you have WinIE5.5+, disable the `myP` element by entering the following statement into the top text box:

```
document.getElementById("myP").disabled = true
```

The sample paragraph's text turns gray.

Related Item: `isDisabled` property.

document

Value: `document` object. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

In the context of HTML element objects as exposed in IE4+, the `document` property is a reference to the document that contains the object. While it is unlikely that you will need to use this property, `document` may come in handy for complex scripts and script libraries that handle objects in a generic fashion and do not know the reference path to the document containing a particular object. You might need a reference to the document to inspect it for related objects. The W3C version of this property is `ownerDocument`.

Example

The following simplified function accepts a parameter that can be any object in a document hierarchy. The script finds out the reference of the object's containing document for further reference to other objects:

```
function getCompanionFormCount(obj) {
   var ownerDoc = obj.document;
   return ownerDoc.forms.length;
}
```

Because the `ownerDoc` variable contains a valid reference to a `document` object, the `return` statement uses that reference to return a typical property of the document object hierarchy.

Related Item: `ownerDocument` property.

filters[]

Value: Array. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

Filters are IE-specific stylesheet add-ons that offer a greater variety of font rendering (such as drop shadows) and transitions between hidden and visible elements. Each filter specification is a `filter` object. The `filters` property contains an array of `filter` objects defined for the current element. You can apply filters to the following set of elements: `body`, `button`, `img`, `input`, `li`, `marquee`, `ol`, `table`, `td`, `textarea`, `th`, `ul`, and positioned `div` and `span` elements. See Chapter 26 for details about stylesheet filters.

Related Item: `filter` object.

firstChild
lastChild

Value: Node object reference. Read-Only
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

W3C DOM-based document object models are built around an architecture known as a *node map*. Each object defined by HTML is a node in the map. A node has relationships with other nodes in the document — relationships described in family terms of parents, siblings, and children.

A *child node* is an element that is contained by another element. The container is the parent of such a child. Just as an HTML element can contain any number of child elements, so can a parent object have zero or more children. A list of those children (returned as an array) can be read from an object by way of its `childNodes` property:

```
var nodeArray = document.getElementById("elementID").childNodes;
```

While you can use this array (and its `length` property) to get a reference to the first or last child node, the `firstChild` and `lastChild` properties offer shortcuts to those positions. These are helpful when you wish to insert a new child before or after all of the others and you need a reference point for the IE `insertAdjacentElement()` method or other method that adds elements to the document's node list.

Example

Listing 15-10 contains an example of how to use the `firstChild` and `lastChild` properties to access child nodes. These two properties come in handy in this example, which adds and replaces `li` elements to an existing `ol` element. You can enter any text you want to appear at the beginning or end of the list. Using the `firstChild` and `lastChild` properties simplifies access to the ends of the list. For the functions that replace child nodes, the example uses the `replaceChild()` method. Alternatively for IE4+, you can modify the `innerText` property of the objects returned by the `firstChild` or `lastChild` property. This example is especially interesting to watch when you add items to the list: The browser automatically renumbers items to fit the current state of the list.

See the discussion of the `childNodes` property earlier in this chapter about the presence of "phantom" nodes in some browser versions. The problem may influence your use of the `firstChild` and `lastChild` properties.

Listing 15-10: **Using firstChild and lastChild Properties**

```
<html>
   <head>
      <title>firstChild and lastChild Properties</title>
      <script type="text/javascript">
      // helper function for prepend() and append()
      function makeNewLI(txt) {
         var newItem = document.createElement("li");
         newItem.innerHTML = txt;
         return newItem;
      }
      function prepend(form) {
         var newItem = makeNewLI(form.input.value);
         var firstLI = document.getElementById("myList").firstChild;
         document.getElementById("myList").insertBefore(newItem, firstLI);
      }
      function append(form) {
         var newItem = makeNewLI(form.input.value);
         var lastLI = document.getElementById("myList").lastChild;
         document.getElementById("myList").appendChild(newItem);
      }
      function replaceFirst(form) {
         var newItem = makeNewLI(form.input.value);
         var firstLI = document.getElementById("myList").firstChild;
```

Continued

Listing 15-10 *(continued)*

```
            document.getElementById("myList").replaceChild(newItem, firstLI);
         }
         function replaceLast(form) {
            var newItem = makeNewLI(form.input.value);
            var lastLI = document.getElementById("myList").lastChild;
            document.getElementById("myList").replaceChild(newItem, lastLI);
         }
         </script>
      </head>
      <body>
         <h1>firstChild and lastChild Property Lab</h1>
         <hr />
         <form>
            <label>Enter some text to add to or replace in the OL
            element:</label><br />
             <input type="text" name="input" size="50" /><br />
             <input type="button" value="Insert at Top"
            onclick="prepend(this.form)" /> <input type="button"
            value="Append to Bottom" onclick="append(this.form)" /><br />
             <input type="button" value="Replace First Item"
            onclick="replaceFirst(this.form)" /> <input type="button"
            value="Replace Last Item" onclick="replaceLast(this.form)" />
         </form>
         <ol id="myList">
            <li>Initial Item 1</li>
            <li>Initial Item 2</li>
            <li>Initial Item 3</li>
            <li>Initial Item 4</li>
         </ol>
      </body>
</html>
```

Related Items: `nextSibling`, `parentElement`, `parentNode`, `previousSibling` properties; `appendChild()`, `hasChildNodes()`, `removeChild()`, `removeNode()`, `replaceChild()`, `replaceNode()` methods.

height
width

Value: Integer or percentage string. Read/Write and Read-Only
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The `height` and `width` properties described here are not the identically named properties that belong to an element's style. Rather, these properties reflect the values normally assigned to `height` and `width` attributes of elements such as `img`, `applet`, `table`, and so on. As such, these properties are accessed directly from the object (for example, `document.getElementById("myTable").width` in IE4+) rather than through the `style` object (for example, `document.getElementById("myDIV").style.width`). Only elements for which the HTML 4.*x* standard provides `height` and `width` attributes have the corresponding properties.

Values for these properties are either integer pixel values (numbers or strings) or percentage values (strings only). If you need to perform some math on an existing percentage value, use the `parseInt()` function to extract the numeric value for use with math calculations. If an element's `height` and `width` attributes are set as percentage values, you can use the `offsetHeight` and `offsetWidth` properties in many modern browsers to get the rendered pixel dimensions.

Property values are read/write for the `image` object in most recent browser versions because you can resize an `image` object in IE4+ and Moz1+ after the page loads. Properties are read/write for some other objects (such as the `table` object)—but not necessarily all others that support these properties.

In general, you cannot set the value of these properties to something less than is required to render the element. This is particularly true of a table. If you attempt to set the `height` value to less than the amount of pixels required to display the table as defined by its style settings, your changes have no effect (even though the property value retains its artificially low value). For other objects, however, you can set the size to anything you like and the browser scales the content accordingly (images, for example). If you want to see only a segment of an element (in other words, to crop the element), use a stylesheet to set the element's clipping region.

Example

The following example demonstrates how to use the `width` property by increasing the width of a table by 10 percent.

```
var tableW = parseInt(document.getElementById("myTable").width);
document.getElementById("myTable").width = (tableW * 1.1) + "%";
```

Because the initial setting for the `width` attribute of the `table` element is set as a percentage value, the script calculation extracts the number from the percentage width string value. In the second statement, the old number is increased by 10 percent and turned into a percentage string by appending the percentage symbol to the value. The resulting string value is assigned to the `width` property of the table.

Related Items: `clientHeight`, `clientWidth` properties; `style.height`, `style.width` properties.

hideFocus

Value: Boolean. Read/Write
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

In IE for Windows, button types of form controls and links display a dotted rectangle around some part of the element whenever that element has focus. If you set the `tabindex` attribute or `tabIndex` property of any other kinds of elements in IE5+, they, too, display that dotted line when given focus. You can still let an element receive focus, but hide that dotted line by setting the `hideFocus` property of the element object to `true` (default value is `false`).

Hiding focus does not disable the element. In fact, if the element about to receive focus is scrolled out of view, the page scrolls to bring the element into view. Form controls that respond to keyboard action (for example, pressing the spacebar to check or uncheck a checkbox control) also continue to work as normal. For some designers, the focus rectangle harms the design goals of the page. The `hideFocus` property gives them more control over the appearance while maintaining consistency of operation with other pages. There is no corresponding HTML attribute for a tag, so you can use an `onload` event handler in the page to set the `hideFocus` property of desired objects after the page loads.

Example

Use The Evaluator (Chapter 13) to experiment with the `hideFocus` property in WinIE5.5+. Enter the following statement into the top text field to assign a `tabIndex` value to the `myP` element so that, by default, the element receives focus and the dotted rectangle:

```
document.getElementById("myP").tabIndex = 1
```

Press the Tab key several times until the paragraph receives focus. Now, disable the focus rectangle:

```
document.getElementById("myP").hideFocus = true
```

If you now press the Tab key several times, the dotted rectangle does not appear around the paragraph. To prove that the element still receives focus, scroll the page down to the bottom so that the paragraph is not visible (you may have to resize the window). Click one of the focusable elements at the bottom of the page, and then press the Tab key slowly until the Address field toolbar has focus. Press the Tab key once. The page scrolls to bring the paragraph into view, but there is no focus rectangle around the element.

Related Items: `tabIndex` property; `srcollIntoView()` method.

id

Value: String. (See text)
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `id` property returns the identifier assigned to an element's `id` attribute in the HTML code. A script cannot modify the ID of an existing element nor assign an ID to an element that lacks one. But if a script creates a new element object, an identifier may be assigned to it by way of the `id` property.

Example

Rarely do you need to access this property in a script — unless you write an authoring tool that iterates through all elements of a page to extract the IDs assigned by the author. You can retrieve an object reference once you know the object's `id` property (via the `document.getElementById(elemID)` method). But if for some reason your script doesn't know the ID of, say, the second paragraph of a document, you can extract that ID as follows:

```
var elemID = document.getElementsByTagName("p")[1].id;
```

Related Item: `className` property.

innerHTML
innerText

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

One way that Internet Explorer exposes the contents of an element is through the `innerHTML` and `innerText` properties. (Moz1+/NN6+ offer only the `innerHTML` property.) All content defined by these "inner" properties consists of document data that is contained by an element's start and end tags, but not including the tags themselves (see `outerText` and `outerHTML` properties). Setting these inner properties is a common way to modify a portion of a page's content after the page loads.

The `innerHTML` property contains not only the text content for an element as seen on the page, but also every bit of HTML tagging that is associated with that content. (If there are no

tags in the content, the text is rendered as is.) For example, consider the following bit of HTML source code:

```
<p id="paragraph1">"How <em>are</em> you?" he asked.</p>
```

The value of the paragraph object's `innerHTML` property (`document.getElementById("paragraph1").innerHTML`) is:

```
"How <em>are</em> you?" he asked.
```

The browser interprets any HTML tags that you include in a string you assign to an element's `innerHTML` property as tags. This also means that you can introduce entirely new nested elements (or child nodes in the modern terminology) by assigning a slew of HTML content to an element's `innerHTML` property. The document's object model adjusts itself to the newly inserted content.

In contrast, the `innerText` property knows only about the text content of an element container. In the example you just saw, the value of the paragraph's `innerText` property (`document.getElementById("paragraph1").innerText`) is:

```
"How are you?" he asked.
```

It's important to remember that if you assign a string to the `innerText` property of an element and that string contains HTML tags, the tags and their angle brackets appear in the rendered page and are not interpreted as live tags.

Do not modify the `innerHTML` property to adjust the HTML for `frameset`, `html`, `head`, or `title` objects. You may modify table constructions through either `innerHTML` or the various table-related methods that create or delete rows, columns, and cells (see Chapter 37 on the CD-ROM). It is also safe to modify the contents of a cell by setting its `innerHTML` or `innerText` properties.

When the HTML you insert includes a `<script>` tag, be sure to include the `defer` attribute to the opening tag. This even goes for scripts that contain function definitions, which you might consider to be deferred automatically.

Example

Listing 15-11 contains an example of how to use the `innerHTML` and `innerText` properties to dynamically alter the content within a page. The page generated in the listing contains an `h1` element label and a paragraph of text. The purpose is to demonstrate how the `innerHTML` and `innerText` properties differ in their intent. Two text boxes contain the same combination of text and HTML tags that replaces the inner content of the paragraph's label.

If you apply the default content of the first text box to the `innerHTML` property of the `label1` object, the italic style is rendered as such for the first word. In addition, the text in parentheses is rendered with the help of the small stylesheet rule assigned by virtue of the surrounding `<span>` tags. But if you apply that same content to the `innerText` property of the `label` object, the tags are rendered as is.

Use this as a laboratory to experiment with some other content in both text boxes. See what happens when you insert a `<br>` tag within some text of both text boxes.

Listing 15-11: **Using innerHTML and innerText Properties**

```
<html>
   <head>
      <title>innerHTML and innerText Properties</title>
```

Continued

Listing 15-11 *(continued)*

```
      <style type="text/css">
      h1 {font-size:18pt; font-weight:bold; font-family:"Comic Sans MS", Arial,
         sans-serif}
      .small {font-size:12pt; font-weight:400; color:gray}
      </style>
      <script type="text/javascript">
      function setGroupLabelAsText(form) {
         var content = form.textInput.value;
         if (content) {
            document.getElementById("label1").innerText = content;
         }
      }
      function setGroupLabelAsHTML(form) {
         var content = form.HTMLInput.value;
         if (content) {
            document.getElementById("label1").innerHTML = content;
         }
      }
      </script>
   </head>
   <body>
      <form>
         <p><input type="text" name="HTMLInput"
            value="&lt;I&gt;First&lt;/I&gt; Article &lt;SPAN
            CLASS='small'&gt;(of ten)&lt;/span&gt;"
            size="50" /> <input type="button" value="Change Heading HTML"
            onclick="setGroupLabelAsHTML(this.form)" /></p>
         <p><input type="text" name="textInput"
            value="&lt;I&gt;First&lt;/I&gt; Article &lt;SPAN
            CLASS='small'&gt;(of ten)&lt;/span&gt;"
            size="50" /> <input type="button" value="Change Heading Text"
            onclick="setGroupLabelAsText(this.form)" /></p>
      </form>
      <h1 id="label1">
         ARTICLE I
      </h1>
      <p>Congress shall make no law respecting an establishment of religion, or
         prohibiting the free exercise thereof; or abridging the freedom of
         speech, or of the press; or the right of the people peaceably to
         assemble, and to petition the government for a redress of
         grievances.</p>
   </body>
</html>
```

Related Items: `outerHTML`, `outerText` properties; `replaceNode()` method.

isContentEditable

Value: Boolean. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The `isContentEditable` property returns a Boolean value indicating whether a particular element object is set to be editable (see the preceding discussion of the `contentEditable` property). This property is helpful because if a parent element's `contentEditable` property

is set to `true`, a nested element's `contentEditable` property likely is set to its default value `inherit`. But because its parent is editable, the `isContentEditable` property of the nested element returns `true`.

Example

Use The Evaluator (Chapter 13) to experiment with both the `contentEditable` and `isContentEditable` properties on the `myP` and nested `myEM` elements (reload the page to start with a known version). Check the current setting for the `myEM` element by typing the following statement into the top text field:

```
myEM.isContentEditable
```

This value is `false` because no element upward in the element containment hierarchy is set to be editable yet. Next, turn on editing for the surrounding `myP` element:

```
myP.contentEditable = true
```

At this point, the entire `myP` element is editable because its child element is set, by default, to inherit the edit state of its parent. Prove it by entering the following statement into the top text box:

```
myEM.isContentEditable
```

While the `myEM` element is shown to be editable, no change has accrued to its `contentEditable` property:

```
myEM.contentEditable
```

This property value remains the default `inherit`.

You can see an additional example of these two properties in use in Listing 15-7.

Related Item: `contentEditable` property.

isDisabled

Value: Boolean. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The `isDisabled` property returns a Boolean value that indicates whether a particular element object is set to be disabled (see the preceding discussion of the `disabled` property). This property is helpful; if a parent element's `disabled` property is set to `true`, then a nested element's `disabled` property likely is set to its default value of `false`. But because its parent is disabled, the `isDisabled` property of the nested element returns `true`. In other words, the `isDisabled` property returns the actual disabled status of an element regardless of its `disabled` property.

Example

Use The Evaluator (Chapter 13) to experiment with both the `disabled` and `isDisabled` properties on the `myP` and nested `myEM` elements (reload the page to start with a known version). Check the current setting for the `myEM` element by typing the following statement into the top text field:

```
myEM.isDisabled
```

This value is `false` because no element upward in the element containment hierarchy is set for disabling yet. Next, disable the surrounding `myP` element:

```
myP.disabled = true
```

At this point, the entire `myP` element (including its children) is disabled. Prove it by entering the following statement into the top text box:

```
myEM.isDisabled
```

While the `myEM` element is shown as disabled, no change has accrued to its `disabled` property:

```
myEM.disabled
```

This property value remains the default `false`.

Related Item: `disabled` property.

isMultiLine

Value: Boolean. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The `isMultiLine` property returns a Boolean value that reveals whether the element object is capable of occupying or displaying more than one line of text. Importantly, this value does not reveal whether the element actually occupies multiple lines; rather, it indicates the potential of doing so. For example, a text `input` element cannot wrap to multiple lines, so its `isMultiLine` property is `false`. However, a `button` element can display multiple lines of text for its label, so it reports `true` for the `isMultiLine` property.

Example

Use The Evaluator (Chapter 13) to read the `isMultiLine` property for elements on that page. Try the following statements in the top text box:

```
document.body.isMultiLine
document.forms[0].input.isMultiLine
myP.isMultiLine
myEM.isMultiLine
```

All but the text field form control report that they are capable of occupying multiple lines.

isTextEdit

Value: Boolean. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `isTextEdit` property reveals whether an object can have a WinIE `TextRange` object created with its content. (See the `TextRange` object in Chapter 35.) You can create `TextRange` objects only from a limited selection of objects in IE4+ for Windows: `body`, `button`, text type `input`, and `textarea`. This property always returns `false` in MacIE.

Example

Good coding practice dictates that your script check for this property before invoking the `createTextRange()` method on any object. A typical implementation is as follows:

```
if (document.getElementById("myObject").isTextEdit) {
   var myRange = document.getElementById("myObject").createTextRange();
   [more statements that act on myRange]
}
```

Related Items: `createRange()` method; `TextRange` object (Chapter 35).

lang

Value: ISO language code string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The lang property governs the written language system used to render an element's text content when overriding the default browser's language system. The default value for this property is an empty string unless the corresponding lang attribute is assigned a value in the element's tag. Modifying the property value by script control does not appear to have any effect in the current browser implementations.

Example

Values for the lang property consist of strings containing valid ISO language codes. Such codes have, at the minimum, a primary language code (for example, "fr" for French) plus an optional region specifier (for example, "fr-ch" for Swiss French). The code to assign a Swiss German value to an element looks like the following:

```
document.getElementById("specialSpan").lang = "de-ch";
```

language

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

IE4+'s architecture allows for multiple scripting engines to work with the browser. Two engines are included with the basic Windows version browser: JScript (compatible with JavaScript) and Visual Basic Scripting Edition (VBScript). The default scripting engine is JScript. But if you wish to use VBScript or some other scripting language in statements that are embedded within event handler attributes of a tag, you can specifically direct the browser to apply the desired scripting engine to those script statements by way of the language attribute of the tag. The language property provides scripted access to that property. Unless you intend to modify the event handler HTML code and replace it with a statement in VBScript (or any other non-JScript-compatible language installed with your browser), you do not need to modify this property (or read it, for that matter).

Valid values include JScript, javascript, vbscript, and vbs. Third-party scripting engines have their own identifier for use with this value. Because the language attribute was also used in the <script> tag, Internet Explorer 5 observes language="xml" as well.

Related Item: script element object.

lastChild

(See firstChild)

length

Value: Integer. Read-Only and Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The length property returns the number of items in an array or collection of objects. Its most common application is as a boundary condition in a for loop. While arrays and collections commonly use integer values as index values (always starting with zero), the length value is the actual number of items in the group. Therefore, to iterate through all items of the group, the condition expression should include a less-than (<) symbol rather than a less-than-or-equal (<=) symbol, as in the following:

```
for (var i = 0; i < someArray.length; i++) {...}
```

For decrementing through an array (in other words, starting from the last item in the array and working toward the first), the initial expression must initialize the counting variable as the length minus one:

```
for (var i = someArray.length - 1; i >= 0; i--) {...}
```

For most arrays and collections, the `length` property is read-only and governed solely by the number of items in the group. But in more recent versions of the browsers, you can assign values to some object arrays (`areas`, `options`, and the `select` object) to create placeholders for data assignments. See discussions of the `area`, `select`, and `option` element objects for details. A plain JavaScript array can also have its `length` property value modified by script to either trim items from the end of the array or reserve space for additional assignments. See Chapter 30 for more about the `Array` object.

Example

You can try the following sequence of statements in the top text box of The Evaluator to see how the `length` property returns values (and sets them for some objects). Note that some statements work in only some browser versions.

```
(All browsers)  document.forms.length
(All browsers)  document.forms[0].elements.length
(NN3+, IE4+)    document.images.length
(NN4+)          document.layers.length
(IE4+)          document.all.length
(IE5+, W3C)     document.getElementById("myTable").childNodes.length
```

Related Items: `area`, `select`, `option`, and `Array` objects.

localName
namespaceURI
prefix

Value: String. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The three properties, `localName`, `namespaceURI`, and `prefix`, apply to any node in an XML document that associates a namespace URI with an XML tag. Although NN6 exposes all three properties for all element (and node) objects, the properties do not return the desired values. However, Mozilla-based browsers remedy the situation. To better understand what values these three properties represent, consider the following XML content:

```
<x xmlns:bk='http://bigbooks.org/schema'>
    <bk:title>To Kill a Mockingbird</bk:title>
</x>
```

The element whose tag is `<bk:title>` is associated with the Namespace URI defined for the block, and the element's `namespaceURI` property would return the string `http:// bigbooks.org/schema`. The tag name consists of a prefix (before the colon) and the local name (after the colon). In the above example, the `prefix` property for the element defined by the `<bk:title>` tag would be `bk`, while the `localName` property would return `title`. The `localName` property of any node returns the same value as its `nodeName` property value, such as `#text` for a text node.

For more information about XML Namespaces, visit `http://www.w3.org/TR/REC-xml-names`.

Related Items: `scopeName`, `tagUrn` properties.

nextSibling
previousSibling

Value: Object reference. Read-Only
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

A *sibling node* is one that is at the same nested level as another node in the hierarchy of an HTML document. For example, the following `p` element has two child nodes (the `em` and `span` elements). Those two child nodes are siblings of each other.

```
<p>MegaCorp is <em>the</em> source of the <span class="hot">hottest</span>
gizmos.</p>
```

Sibling order is determined solely by the source code order of the nodes. Therefore, in the previous example, the `em` node has no `previousSibling` property. Meanwhile, the `span` node has no `nextSibling` property (meaning that these properties return `null`). These properties provide another way to iterate through all nodes at the same level.

Example

The following function assigns the same class name to all child nodes of an element:

```
function setAllChildClasses(parentElem, className) {
   var childElem = parentElem.firstChild;
   while (childElem.nextSibling) {
      childElem.className = className;
      childElem = childElem.nextSibling;
   }
}
```

This example is certainly not the only way to achieve the same results. Using a `for` loop to iterate through the `childNodes` collection of the parent element is an equally valid approach.

Related Items: `firstChild`, `lastChild`, `childNodes` properties; `hasChildNodes()`, `insertAdjacentElement()` methods.

nodeName

Value: String. Read-Only
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

For HTML and XML elements, the name of a node is the same as the tag name. The `nodeName` property is provided for the sake of consistency with the node architecture specified by the formal W3C DOM standard. The value, just like the `tagName` property, is an all-uppercase string of the tag name (even if the HTML source code is written with lowercase tags).

Some nodes, such as the text content of an element, do not have a tag. The `nodeName` property for such a node is a special value: `#text`. Another kind of node is an attribute of an element. For an attribute, the `nodeName` is the name of the attribute. See Chapter 14 for more about `Node` object properties.

Example

The following function demonstrates one (not very efficient) way to assign a new class name to every `p` element in an IE5+ document:

```
function setAllPClasses(className) {
   for (var i = 0; i < document.all.length; i++) {
      if (document.all[i].nodeName == "P") {
         document.all[i].className = className;
```

```
        }
    }
}
```

A more efficient approach uses the `getElementsByTagName()` method to retrieve a collection of all `p` elements and then iterate through them directly.

Related Item: `tagName` property.

nodeType

Value: Integer. Read-Only
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The W3C DOM specification identifies a series of constant values that denote categories of nodes. Every node has a value that identifies its type, but not all browsers support the `nodeType` property on all node types as objects. Table 15-4 lists the `nodeType` values implemented in recent browsers; all of the values are considered part of the W3C DOM Level 2 specification.

Table 15-4: nodeType Property Values

Value	*Description*	*WinIE*	*MacIE*	*NN/Moz*	*Safari*
1	Element node	5	5	NN6	1
2	Attribute node	6	5	NN6	1
3	Text (`#text`) node	5	5	NN6	1
4	CDATA section node	-	-	-	-
5	Entity reference node	-	-	-	-
6	Entity node	-	-	-	-
7	Processing instruction node	-	-	-	-
8	Comment node	6	5	NN6	-
9	Document node	5	5	NN6	1
10	Document type node	-	-	NN6	1
11	Document fragment node	6	5	NN6	1
12	Notation node	-	-	-	-

The `nodeType` value is automatically assigned to a node, whether the node exists in the document's HTML source code or it is generated on the fly via a script. For example, if you create a new element node through any of the ways available by script (for example, by assigning a string encased in HTML tags to the `innerHTML` property or by explicitly invoking the `document.createElement()` method), the new element assumes a `nodeType` of `1`.

Mozilla-based browsers and Safari go one step further in supporting the W3C DOM specification by implementing a set of `Node` object property constants for each of the `nodeType` values. Table 15-5 lists the entire set as defined in the DOM Level 2 specification. Substituting these constants for `nodeType` integers can improve readability of a script. For example, instead of

```
if (myElem.nodeType == 1) {...}
```

it is much easier to see what's going on with

```
if (myElem.nodeType == Node.ELEMENT_NODE) {...}
```

Table 15-5: W3C DOM nodeType Constants

Reference	*nodeType Value*
Node.ELEMENT_NODE	1
Node.ATTRIBUTE_NODE	2
Node.TEXT_NODE	3
Node.CDATA_SECTION_NODE	4
Node.ENTITY_REFERENCE_NODE	5
Node.ENTITY_NODE	6
Node.PROCESSING_INSTRUCTION_NODE	7
Node.COMMENT_NODE	8
Node.DOCUMENT_NODE	9
Node.DOCUMENT_TYPE_NODE	10
Node.DOCUMENT_FRAGMENT_NODE	11
Node.NOTATION_NODE	12

Example

You can experiment with viewing `nodeType` property values in The Evaluator. The `p` element whose ID is `myP` is a good place to start. The `p` element itself is a `nodeType` of `1`:

```
document.getElementById("myP").nodeType
```

This element has three child nodes: a string of text (`nodeName` `#text`); an `em` element (`nodeName` `em`); and the rest of the text of the element content (`nodeName` `#text`). If you view the `nodeType` of either of the text portions, the value comes back as `3`:

```
document.getElementById("myP").childNodes[0].nodeType
```

Related Item: `nodeName` property.

nodeValue

Value: Number, string, or null. Read/Write
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

For a text node, the `nodeValue` property consists of the actual text for that node. Such a node cannot contain any further nested elements, so the `nodeValue` property offers another way of reading and modifying what Internet Explorer implements as an element's `innerText` property (but in the W3C DOM you must reference the child text node of an element to get or set its node value).

Of the node types implemented in the W3C DOM–capable browsers, only the text and attribute types have readable values. An element's node value returns a `null` value. For an attribute node, the `nodeValue` property consists of the value assigned to that attribute. According to the W3C DOM standard, attribute values should be reflected as strings. WinIE5, however, returns values of type `Number` when the value is all numeric characters. Even if you assign a string version of a number to such a `nodeValue` property, it is converted to a `Number` type internally. NN6 and MacIE5 return `nodeValue` values as strings in all cases (and convert numeric assignments to strings).

Example

You can use the `nodeValue` property to carry out practical tasks. As an example, `nodeValue` can be used to increase the width of a `textarea` object by 10 percent. The `nodeValue` is converted to an integer before performing the math and reassignment:

```
function widenCols(textareaElem) {
   var colWidth = parseInt(textareaElem.attributes["cols"].nodeValue, 10);
   textareaElem.attributes["cols"].nodeValue = (colWidth * 1.1);
}
```

As another example, you can replace the text of an element, assuming that the element contains no further nested elements:

```
function replaceText(elem, newText) {
   if (elem.childNodes.length == 1 && elem.firstChild.nodeType == 3) {
      elem.firstChild.nodeValue = newText;
   }
}
```

The function builds in one final verification that the element contains just one child node and that it is a text type. An alternative version of the assignment statement of the second example uses the `innerText` property in IE with identical results:

```
elem.innerText = newText;
```

Related Items: `attributes`, `innerText`, `nodeType` properties.

offsetHeight
offsetWidth

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

These properties, which ostensibly report the height and width of any element, have had a checkered history due to conflicts between interpretations of the CSS "box model" by Microsoft and the W3C. Both properties were invented by Microsoft for IE4. Although they are not part of any W3C standard, other modern browsers, including Mozilla-based browsers and Safari, implement the properties because they're so valuable to scripters.

When IE6 is set to standards-compatible mode (by DOCTYPE switching described in Chapter 14), the properties measure the pixel dimensions of the element's content, regardless of borders, margins, or padding. This is the W3C box model, which agrees with other W3C-compatible browsers. But in IE6 quirks mode and all older versions of WinIE, the measures incorrectly include border and padding thicknesses.

Be aware that for a normal block-level element whose height and width are not specified, the `offsetHeight` is determined by the actual height of the content after all text flows. But the `offsetWidth` always extends the full width of the containing element. Therefore, the `offsetWidth` property does not reveal the rendered width of text content that is narrower than the full parent element width. For example, a `p` element consisting of only a few words may report an `offsetWidth` of many hundreds of pixels because the paragraph's block extends the full width of the `body` element that represents the containing parent of the `p` element. To find out the actual width of text within a full-width, block-level element, wrap the text within an inline element (such as a `span`) and inspect the `offsetWidth` property of the `span`.

Example

With IE4+, you can substitute the `offsetHeight` and `offsetWidth` properties for `clientHeight` and `clientWidth` in Listing 15-6. The reason is that the two elements in question have their widths hard-wired in stylesheets. Thus, the `offsetWidth` property follows that lead rather than observing the default width of the parent (BODY) element.

With IE5+ and W3C browsers, you can use The Evaluator to inspect the `offsetHeight` and `offsetWidth` property values of various objects on the page. Enter the following statements into the top text box:

```
document.getElementById("myP").offsetWidth
document.getElementById("myEM").offsetWidth
document.getElementById("myP").offsetHeight
document.getElementById("myTable").offsetWidth
```

Related Items: `clientHeight`, `clientWidth` properties.

offsetLeft
offsetTop

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `offsetLeft` and `offsetTop` properties can suffer from the same version vagaries that afflict `offsetHeight` and `offsetWidth` properties when borders, margins, and padding are associated with an element and DOCTYPE switching is a factor. However, the `offsetLeft` and `offsetTop` properties are valuable in providing pixel coordinates of an element within the positioning context of the parent element — even when the elements are not positioned explicitly.

The `offsetLeft` and `offsetTop` properties for positioned elements in MacIE do not return the same values as the `style.left` and `style.top` properties of the same element. See Listing 39-17 on the CD-ROM for an example of how to correct these discrepancies without having to hard-wire the precise pixel differences in your code.

The element used as a coordinate context for these properties is whatever element the `offsetParent` property returns. This means that to determine the precise position of any element, you may have to add some code that iterates through the `offsetParent` hierarchy until that property returns `null`.

Although the `offsetLeft` and `offsetTop` properties are not part of the W3C DOM specification, they are supported across most browsers because they are convenient for some scriptable Dynamic HTML tasks. Through these two properties, a script can read the pixel coordinates of any block-level or inline element. Measurements are made relative to the `body` element, but this may change in the future. See the discussion later in this chapter about the `offsetParent` property.

Example

The following IE script statements utilize all four "offset" dimensional properties to size and position a `div` element so that it completely covers a `span` element located within a `p` element. This can be for a fill-in-the-blank quiz that provides text entry fields elsewhere on the page. As the user gets an answer correct, the blocking `div` element is hidden to reveal the correct answer.

```
document.all.blocker.style.pixelLeft = document.all.span2.offsetLeft
document.all.blocker.style.pixelTop = document.all.span2.offsetTop
```

```
document.all.blockImg.height = document.all.span2.offsetHeight
document.all.blockImg.width = document.all.span2.offsetWidth
```

Because the `offsetParent` property for the `span` element is the `body` element, the positioned `div` element can use the same positioning context (it's the default context, anyway) for setting the `pixelLeft` and `pixelTop` style properties. (Remember that positioning properties belong to an element's `style` object.) The `offsetHeight` and `offsetWidth` properties can read the dimensions of the `span` element (the example has no borders, margins, or padding to worry about) and assign them to the dimensions of the image contained by the blocker `div` element.

This example is also a bit hazardous in some implementations. If the text of `span2` wraps to a new line, the new `offsetHeight` value has enough pixels to accommodate both lines. But the `blockImg` and `blocker div` elements are block-level elements that render as a simple rectangle. In other words, the `blocker` element doesn't turn into two separate strips to cover the pieces of `span2` that spread across two lines.

Related Items: `clientLeft`, `clientTop`, `offsetParent` properties.

offsetParent

Value: Object reference. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `offsetParent` property returns a reference to the object that acts as a positioning context for the current element. Values for the `offsetLeft` and `offsetTop` properties are measured relative to the top-left corner of the `offsetParent` object.

The returned object is usually, but not always, the next outermost block-level container. For most document elements, the `offsetParent` object is the `document.body` object (with exceptions for some elements in some browsers).

Table cells, for example, have different `offsetParent` elements in different browsers:

Browser	*td offsetParent*
WinIE4	`tr`
WinIE5+/NN7+/Moz1+	`table`
MacIE	`table`
NN6	`body`

The property behaves predictably for positioned elements in most browsers. For example, a first-level positioned element's `offsetParent` element is the `body`; the `offsetParent` of a nested positioned element (for example, one absolute-positioned `div` inside another) is the next outer container (in other words, the positioning context of the inner element).

Example

You can use the `offsetParent` property to help you locate the position of a nested element on the page. Listing 15-12 demonstrates how a script can "walk" up the hierarchy of `offsetParent` objects in IE for Windows to assemble the location of a nested element on a page. The goal of the exercise in Listing 15-12 is to position an image at the upper-left corner of the second table cell. The entire table is centered on the page.

The `onload` event handler invokes the `setImagePosition()` function. The function first sets a Boolean flag that determines whether the calculations should be based on the client or offset

sets of properties. WinIE4 and MacIE5 rely on client properties, while WinIE5+ works with the offset properties. The discrepancies even out, however, with the `while` loop. This loop traverses the `offsetParent` hierarchy starting with the `offsetParent` of the cell out to, but not including, the `document.body` object. The `body` object is not included because that is the positioning context for the image. In IE5, the `while` loop executes only once because just the `table` element exists between the cell and the body; in IE4, the loop executes twice to account for the `tr` and `table` elements up the hierarchy. Finally, the cumulative values of left and top measures are applied to the positioning properties of the `div` object's style and the image is made visible.

Listing 15-12: **Using the offsetParent Property**

```
<html>
   <head>
      <title>offsetParent Property</title>
      <script type="text/javascript">
      function setImagePosition(){
         var x = 0;
         var y = 0;
         var offsetPointer = document.getElementById("myCell"); // cElement;
         while (offsetPointer) {
            x += offsetPointer.offsetLeft;
            y += offsetPointer.offsetTop;
            offsetPointer = offsetPointer.offsetParent;
         }
         // correct for MacIE body margin factors
         if (navigator.userAgent.indexOf("Mac") != -1 &&
            typeof document.body.leftMargin != "undefined") {
            x += document.body.leftMargin;
            y += document.body.topMargin;
         }
         document.getElementById("myDIV").style.left = x + "px";
         document.getElementById("myDIV").style.top = y + "px";
         document.getElementById("myDIV").style.visibility = "visible";
      }
      </script>
   </head>
   <body onload="setImagePosition()">
      <h1>The offsetParent Property</h1>
      <hr />
      <p>After the document loads, the script positions a small image in the
         upper left corner of the second table cell.</p>
      <table border="1" align="center">
         <tr>
            <td>This is the first cell</td>
            <td id="myCell">This is the second cell.</td>
         </tr>
      </table>
      <img id="myDIV" alt="image" src="end.gif" height="12" width="12"
         style="position:absolute; visibility:hidden; height:12px;
         width:12px" />
   </body>
</html>
```

Related Items: `offsetLeft`, `offsetTop`, `offsetHeight`, `offsetWidth` properties.

outerHTML
outerText

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

One way that Internet Explorer exposes an entire element to scripting is by way of the `outerHTML` and `outerText` properties. The primary distinction between these two properties is that `outerHTML` includes the element's start and end tags whereas `outerText` includes only rendered text that belongs to the element (including text from any nested elements).

The `outerHTML` property contains not only the text content for an element as seen on the page, but also every bit of HTML tagging associated with that content. For example, consider the following bit of HTML source code:

```
<p id="paragraph1">"How <em>are</em> you?" he asked.</p>
```

The value of the `p` object's `outerHTML` property (`document.all.paragraph1. outerHTML`) is exactly the same as that of the source code.

The browser interprets any HTML tags in a string that you assign to an element's `outerHTML` property. This means that you can delete (set the property to an empty string) or replace an entire tag with this property. The document's object model adjusts itself to whatever adjustments you make to the HTML in this manner.

In contrast, the `outerText` property knows only about the text content of an element container. In the preceding example, the value of the paragraph's `outerText` property (`document.all.paragraph1.innerText`) is:

```
"How are you?" he asked.
```

If this looks familiar, it's because in most cases the `innerText` and `outerText` properties of an existing element return the exact same strings.

Example

Listing 15-13 demonstrates how to use the `outerHTML` and `outerText` properties in IE to access and modify Web page content dynamically. The page generated by Listing 15-13 (WinIE4+ only) contains an `h1` element label and a paragraph of text. The purpose is to demonstrate how the `outerHTML` and `outerText` properties differ in their intent. Two text boxes contain the same combination of text and HTML tags that replaces the element that creates the paragraph's label.

If you apply the default content of the first text box to the `outerHTML` property of the `label1` object, the `h1` element is replaced by a `span` element whose `class` attribute acquires a different stylesheet rule defined earlier in the document. Notice that the ID of the new `span` element is the same as the original `h1` element. This allows the script attached to the second button to address the object. But this second script replaces the element with the raw text (including tags). The element is now gone, and any attempt to change the `outerHTML` or `outerText` properties of the `label1` object causes an error because there is no longer a `label1` object in the document.

Use this laboratory to experiment with some other content in both text boxes.

Listing 15-13: **Using outerHTML and outerText Properties**

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

```
      <title>outerHTML and outerText Properties</title>
      <style type="text/css">
      h1 {font-size:18pt; font-weight:bold; font-family:"Comic Sans MS", Arial,
         sans-serif}
      .heading {font-size:20pt; font-weight:bold; font-family:"Arial Black",
         Arial, sans-serif}
      </style>
      <script type="text/javascript">
      function setGroupLabelAsText(form) {
         var content = form.textInput.value;
         if (content) {
            document.getElementById("label1").outerText = content;
         }
      }
      function setGroupLabelAsHTML(form) {
         var content = form.HTMLInput.value;
         if (content) {
            document.getElementById("label1").outerHTML = content;
         }
      }
      </script>
   </head>
   <body>
      <form>
         <p><input type="text" name="HTMLInput"
            value="&lt;SPAN ID='label1' CLASS='heading'&gt;Article the
            First&lt;/SPAN&gt;"
            size="55" /> <input type="button" value="Change Heading HTML"
            onclick="setGroupLabelAsHTML(this.form)" /></p>
         <p><input type="text" name="textInput"
            value="&lt;SPAN ID='label1' CLASS='heading'&gt;Article the
            First&lt;/SPAN&gt;"
            size="55" /> <input type="button" value="Change Heading Text"
            onclick="setGroupLabelAsText(this.form)" /></p>
      </form>
      <h1 id="label1">ARTICLE I</h1>
      <p>Congress shall make no law respecting an establishment of religion, or
         prohibiting the free exercise thereof; or abridging the freedom of
         speech, or of the press; or the right of the people peaceably to
         assemble, and to petition the government for a redress of
         grievances.</p>
   </body>
</html>
```

Related Items: `innerHTML`, `innerText` properties; `replaceNode()` method.

ownerDocument

Value: Document object reference. Read-Only
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The `ownerDocument` property belongs to any element or node in the W3C DOM. The property's value is a reference to the document node that ultimately contains the element or node. If a script encounters a reference to an element or node (perhaps it has been passed as a parameter to a function), the object's `ownerDocument` property provides a way to build references to other objects in the same document or to access properties and methods of the document objects. IE's proprietary version of this property is simply `document`.

Example

Use The Evaluator (Chapter 13) to explore the `ownerDocument` property. Enter the following statement into the top text box:

```
document.body.childNodes[5].ownerDocument
```

The result is a reference to the `document` object. You can use that to inspect a property of the document, as shown in the following statement you should enter into the top text box:

```
document.body.childNodes[5].ownerDocument.URL
```

This returns the `document.URL` property for the document that owns the child node.

Related Item: `document` object.

parentElement

Value: Element object reference or `null`. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `parentElement` property returns a reference to the next outermost HTML element from the current element. This parent–child relationship of elements is often, but not always, the same as a parent–child node relationship (see `parentNode` property later in this chapter). The difference is that the `parentElement` property deals only with HTML elements as reflected as document objects, whereas a node is not necessarily an HTML element (for example, an attribute or text chunk).

There is also a distinction between `parentElement` and `offsetParent` properties. The latter returns an element that may be many generations removed from a given element but is the immediate parent with regard to positioning context. For example, a `td` element's `parentElement` property is most likely its enclosing `tr` element, but a `td` element's `offsetParent` property is its `table` element.

A script can "walk" the element hierarchy outward from an element with the help of the `parentElement` property. The top of the parent chain is the `html` element. Its `parentElement` property returns `null`.

Example

You can experiment with the `parentElement` property in The Evaluator. The document contains a `p` element named `myP`. Type each of the following statements from the left column into the upper expression evaluation text box and press Enter to see the results.

Expression	*Result*
`document.getElementById("myP").tagName`	`p`
`document.getElementById("myP").parentElement`	`[object]`
`document.getElementById("myP").parentElement.tagName`	`body`
`document.getElementById("myP").parentElement.parentElement`	`[object]`
`document.getElementById("myP").parentElement.parentElement.tagName`	`html`
`document.getElementById("myP").parentElement.parentElement.parentElement`	`null`

Related Items: `offsetParent`, `parentNode` properties.

parentNode

Value: Node object reference or `null`. Read-Only
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The `parentNode` property returns a reference to the next outermost node that is reflected as an object belonging to the document. For a standard element object, the `parentNode` property is the same as IE's `parentElement` because both objects happen to have a direct parent–child node relationship as well as a parent–child element relationship.

Other kinds of content, however, can be nodes. This includes text fragments within an element. A text fragment's `parentNode` property is the next outermost node or element that encompasses that fragment. A text node object in IE does not have a `parentElement` property.

Example

Use The Evaluator to examine the `parentNode` property values of both an element and a non-element node. Begin with the following two statements and watch the results of each:

```
document.getElementById("myP").parentNode.tagName
document.getElementById("myP").parentElement.tagName    (IE only)
```

Now examine the properties from the point of view of the first text fragment node of the `myP` paragraph element:

```
document.getElementById("myP").childNodes[0].nodeValue
document.getElementById("myP").childNodes[0].parentNode.tagName
document.getElementById("myP").childNodes[0].parentElement    (IE only)
```

Notice (in IE) that the text node does not have a `parentElement` property.

Related Items: `childNodes`, `nodeName`, `nodeType`, `nodeValue`, `parentElement` properties.

parentTextEdit

Value: Element object reference or `null`. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

Only a handful of objects in IE's object model are capable of creating text ranges (see the `TextRange` object in Chapter 35). To find an object's next outermost container capable of generating a text range, use the `parentTextEdit` property. If an element is in the hierarchy, that element's object reference is returned. Otherwise (for example, `document.body.parentTextEdit`), the value is `null`. MacIE always returns a value of `null` because the browser doesn't support the `TextRange` object.

Example

Listing 15-14 contains an example that demonstrates how to use the `parentTextEdit` property to create a text range. The page resulting from Listing 15-14 contains a paragraph of Latin text and three radio buttons that select the size of a paragraph chunk: one character, one word, or one sentence. If you click anywhere within the large paragraph, the `onclick` event handler invokes the `selectChunk()` function. The function first examines which of the radio buttons is selected to determine how much of the paragraph to highlight (select) around the point at which the user clicks.

After the script employs the `parentTextEdit` property to test whether the clicked element has a valid parent capable of creating a text range, it calls upon the property again to help create the text range. From there, `TextRange` object methods shrink the range to a single insertion point, move that point to the spot nearest the cursor location at click time, expand the selection to encompass the desired chunk, and select that bit of text.

Notice one workaround for the `TextRange` object's `expand()` method anomaly: If you specify a sentence, IE doesn't treat the beginning of a `p` element as the starting end of a sentence automatically. A camouflaged (white text color) period is appended to the end of the previous element to force the `TextRange` object to expand only to the beginning of the first sentence of the targeted `p` element.

Listing 15-14: **Using the parentTextEdit Property**

```
<html>
   <head>
      <title>parentTextEdit Property</title>
      <style type="text/css">
      p {cursor:hand}
      </style>
      <script type="text/javascript">
      function selectChunk() {
         var chunk, range;
         for (var i = 0; i < document.forms[0].chunk.length; i++) {
            if (document.forms[0].chunk[i].checked) {
               chunk = document.forms[0].chunk[i].value;
               break;
            }
         }
         var x = window.event.clientX;
         var y = window.event.clientY;
         if (window.event.srcElement.parentTextEdit) {
            range = window.event.srcElement.parentTextEdit.createTextRange();
            range.collapse();
            range.moveToPoint(x, y);
            range.expand(chunk);
            range.select();
         }
      }
      </script>
   </head>
   <body bgcolor="white">
      <form>
         <p>Choose how much of the paragraph is to be selected when you click
            anywhere in it:<br />
             <input type="radio" name="chunk" value="character"
            checked="checked" />Character <input type="radio" name="chunk"
            value="word" />Word <input type="radio" name="chunk"
            value="sentence" />Sentence <font color="white">.</font></p>
      </form>
      <p onclick="selectChunk()">Lorem ipsum dolor sit amet, consectetaur
         adipisicing
         elit, sed do eiusmod tempor incididunt ut labore et dolore magna
         aliqua. Ut
         enim adminim veniam, quis nostrud exercitation ullamco laboris nisi ut
         aliquip ex ea commodo consequat. Duis aute irure dolor in
         reprehenderit involuptate velit esse cillum dolore eu fugiat nulla
         pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
         culpa qui officia deserunt mollit anim id est laborum.</p>
   </body>
</html>
```

Related Items: `isTextEdit` property; `TextRange` object (Chapter 35).

previousSibling

(See `nextSibling`)

readyState

Value: String (integer for OBJECT object). Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

A script can query an element in IE to find out if it has loaded all ancillary data (for example, external image files or other media files) before other statements act on that object or its data. The `readyState` property lets you know the loading status of an element.

Table 15-6 lists the possible values and their meanings.

Table 15-6: readyState Property Values

HTML Value	*OBJECT Value*	*Description*
`complete`	4	Element and data fully loaded
`interactive`	3	Data may not be loaded fully, but user can interact with element
`loaded`	2	Data is loaded, but object may be starting up
`loading`	1	Data is loading
`uninitialized`	0	Object has not started loading data yet

For most HTML elements, this property always returns `complete`. Most of the other states are used by elements such as `img`, `embed`, and `object`, which load external data and even start other processes (such as ActiveX controls) to work.

One word of caution: Do not expect the `readyState` property to reveal if an object exists yet in the document (for example, `uninitialized`). If the object does not exist, it cannot have a `readyState` property — the result is a script error for an undefined object. If you want to run a script only after every element and its data are fully loaded, trigger the function by way of the `onload` event handler for the `body` element or the `onreadystatechange` event handler for the object (and check that the `readyState` property is `complete`).

Example

To witness a `readyState` property other than `complete` for standard HTML, you can try examining the property in a script that immediately follows an `<img>` tag:

```
...
<img id="myImg" src="someImage.gif">
<script type="text/javaScript">
alert(document.getElementById("myImg").readyState);
</script>
...
```

Putting this fragment into a document that is accessible across a slow network helps. If the image is not in the browser's cache, you might get the `uninitialized` or `loading` result. The former means that the `img` object exists, but it has not started receiving the image data

from the server yet. If you reload the page, chances are that the image will load instantaneously from the cache and the `readyState` property will report `complete`.

Related Items: `onreadystatechange` event handler.

`recordNumber`

Value: Integer or `null`. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

Virtually every object has a `recordNumber` property, but it applies only to elements used in Internet Explorer data binding to represent repeated data. For example, if you display 30 records from an external data store in a table, the `tr` element in the table is represented only once in the HTML. However, the browser repeats the table row (and its component cells) to accommodate all 30 rows of data. If you click a row, you can use the `recordNumber` property of the `tr` object to see which record was clicked. A common application of this facility is in data binding situations that allow for updating records. For example, script a table so that clicking on an uneditable row of data displays that record's data in editable text boxes elsewhere on the page. If an object is not bound to a data source, or it is a non-repeating object bound to a data source, the `recordNumber` property is `null`.

Example

Listing 15-15 shows how to use the `recordNumber` property to navigate to a specific record in a sequence of data. The data source is a small, tab-delimited file consisting of 20 records of Academy Award data. Thus, the table that displays a subset of the fields is bound to the data source object. Also bound to the data source object are three `span` objects embedded within a paragraph near the top of the page. As the user clicks a row of data, three fields from that clicked record are placed into the bound `span` objects.

The script part of this page is a mere single statement. When the user triggers the `onclick` event handler of the repeated `tr` object, the function receives as a parameter a reference to the `tr` object. The data store object maintains an internal copy of the data in a `recordset` object. One of the properties of this `recordset` object is the `AbsolutePosition` property, which is the integer value of the current record that the data object points to (it can point to only one row at a time, and the default row is the first row). The statement sets the `AbsolutePosition` property of the `recordset` object to the `recordNumber` property for the row that the user clicks. Because the three `span` elements are bound to the same data source, they are immediately updated to reflect the change to the data object's internal pointer to the current record. Notice, too, that the third `span` object is bound to one of the data source fields not shown in the table. You can reach any field of a record because the Data Source Object holds the entire data source content.

Listing 15-15: Using the Data Binding recordNumber Property

```
<html>
   <head>
      <title>Data Binding (recordNumber)</title>
      <style type="text/css">
      .filmTitle {font-style:italic}
      </style>
      <script type="text/javascript">
      // set recordset pointer to the record clicked on in the table.
      function setRecNum(row) {
```

```
            document.oscars.recordset.AbsolutePosition = row.recordNumber;
         }
         </script>
      </head>
      <body>
         <p><b>Academy Awards 1978-1997</b> (Click on a table row to extract data
            from one record.)</p>
         <p>The award for Best Actor of <span datasrc="#oscars"
            datafld="Year"></span>
             went to <span datasrc="#oscars" datafld="Best Actor"></span>
             for his outstanding achievement in the film <span
            class="filmTitle"
            datasrc="#oscars" datafld="Best Actor Film"></span>.</p>
         <table border="1" datasrc="#oscars" align="center">
            <thead style="background-color:yellow; text-align:center">
               <tr>
                  <td>Year</td>
                  <td>Film</td>
                  <td>Director</td>
                  <td>Actress</td>
                  <td>Actor</td>
               </tr>
            </thead>
            <tr id="repeatableRow" onclick="setRecNum(this)">
               <td><div id="col1" datafld="Year"></div></td>
               <td><div class="filmTitle" id="col2" datafld="Best
               Picture"></div></td>
               <td><div id="col3" datafld="Best Director"></div></td>
               <td><div id="col4" datafld="Best Actress"></div></td>
               <td><div id="col5" datafld="Best Actor"></div></td>
            </tr>
         </table>
         <object id="oscars" classid="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83">
            <param name="DataURL" value="Academy Awards.txt" />
            <param name="UseHeader" value="True" />
            <param name="FieldDelim" value="&#09;" />
         </object>
      </body>
   </html>
```

Related Items: `dataFld`, `dataSrc` properties; `table`, `tr` objects (Chapter 37 on the CD-ROM).

runtimeStyle

Value: `style` object. Read-Only
Compatibility: WinIE5+, MacIE5+, NN-, Moz-, Safari-

You can determine the browser default settings for stylesheet attributes with the help of the `runtimeStyle` property. The `style` object that this property returns contains all style attributes and the default settings at the time the page loads. This property does not reflect values assigned to elements by stylesheets in the document or by scripts. The default values returned by this property differ from the values returned by the `currentStyle` property. The latter includes data about values that are not assigned explicitly by stylesheets, yet are influenced by the default behavior of the browser's rendering engine. In contrast, the `runtimeStyle` property shows unassigned style values as empty or zero.

Example

To change a style property setting, access it via the element's `style` object. Use The Evaluator (Chapter 13) to compare the properties of the `runtimeStyle` and `style` objects of an element. For example, an unmodified copy of The Evaluator contains an `em` element whose ID is `"myEM"`. Enter both

```
document.getElementById("myEM").style.color
```

and

```
document.getElementById("myEM").runtimeStyle.color
```

into the top text field in turn. Initially, both values are empty. Now assign a color to the `style` property via the upper text box:

```
document.getElementById("myEM").style.color = "red"
```

If you now type the two earlier statements into the upper box, you can see that the `style` object reflects the change, while the `runtimeStyle` object still holds onto its original (empty) value.

Related Items: `currentStyle` property; `style` object (Chapter 26).

scopeName

Value: String. Read-Only
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `scopeName` property is associated primarily with XML that is embedded within a document. When you include XML, you can specify one or more XML Namespaces that define the "owner" of a custom tag name, thus aiming toward preventing conflicts of identical custom tags from different sources in a document. (See Chapter 41 on the CD-ROM for more about XML objects.)

The XML Namespace is assigned as an attribute of the `<html>` tag that surrounds the entire document:

```
<html xmlns:fred='http://www.someURL.com'>
```

After that, the Namespace value precedes all custom tags linked to that Namespace:

```
<fred:FIRST_Name id="fredFirstName"/>
```

To find out the Namespace "owner" of an element, you can read the `scopeName` property of that element. For the preceding example, the `scopeName` returns `fred`. For regular HTML elements, the returned value is always `HTML`. The `scopeName` property is available only in Win32 and UNIX flavors of IE5+. The comparable property in the W3C DOM is `localName`.

Example

If you have a sample document that contains XML and a namespace spec, you can use `document.write()` or `alert()` methods to view the value of the `scopeName` property. The syntax is

```
document.getElementById("elementID").scopeName
```

Related Item: `tagUrn` property.

scrollHeight
scrollWidth

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN7+, Moz1+, Safari1+

The scrollHeight and scrollWidth properties contain the pixel measures of an object, regardless of how much of the object is visible on the page. Therefore, if the browser window displays a vertical scrollbar, and the body extends below the bottom of the viewable space in the window, the scrollHeight takes into account the entire height of the body as if you were to scroll downward and see the entire element. For most elements that don't have their own scrollbars, the scrollHeight and scrollWidth properties have the same values as the clientHeight and clientWidth properties.

Example

Use The Evaluator (Chapter 13) to experiment with these two properties of the textarea object, which displays the output of evaluations and property listings. To begin, enter the following into the bottom one-line text field to list the properties of the body object:

```
document.body
```

This displays a long list of properties for the body object. Now enter the following property expression in the top one-line text field to see the scrollHeight property of the output textarea when it holds the dozens of lines of property listings:

```
document.getElementById("output").scrollHeight
```

The result, some number probably in the hundreds, is now displayed in the output textarea. This means that you can scroll the content of the output element vertically to reveal that number of pixels. Click the Evaluate button once more. The result, 13 or 14, is a measure of the scrollHeight property of the textarea that had only the previous result in it. The scrollable height of that content was only 13 or 14 pixels, the height of the font in the textarea. The scrollWidth property of the output textarea is fixed by the width assigned to the element's cols attribute (as calculated by the browser to determine how wide to make the text area on the page).

Related Items: clientHeight , clientWidth properties; window.scroll() method.

scrollLeft
scrollTop

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN7+, Moz1+, Safari1+

If an element is scrollable (in other words, it has its own scrollbars), you can find out how far the element is scrolled in the horizontal and vertical direction via the scrollLeft and scrollTop properties. These values are pixels. For non-scrollable elements, these values are always zero — even if they are contained by elements that are scrollable. For example, if you scroll a browser window (or frame in a multiframe environment) vertically, the scrollTop property of the body object is whatever the pixel distance is between the top of the object (now out of view) and the first visible row of pixels of the element. But the scrollTop value of a table that is in the document remains at zero.

Netscape browsers prior to version 7 (Moz1) treat scrolling of a body element from the point of view of the window. If you want to find out the scrolled offset of the current page in these browsers, use window.scrollX and window.scrollY.

Scripts that involve tracking mouse events in IE need to take into account the scrollLeft and scrollTop properties of the body to compensate for scrolling of the page. See the Event object in Chapter 25.

Example

Use The Evaluator (Chapter 13) to experiment with these two properties of the `textarea` object, which displays the output of evaluations and property listings. To begin, enter the following into the bottom one-line text field to list the properties of the `body` object:

```
document.body
```

This displays a long list of properties for the `body` object. Use the `textarea`'s scrollbar to page down a couple of times. Now enter the following property expression in the top one-line text field to see the `scrollTop` property of the output `textarea` after you scroll:

```
document.getElementById("output").scrollTop
```

The result, some number, is now displayed in the output `textarea`. This means that the content of the `output` element was scrolled vertically. Click the Evaluate button once more. The result, `0`, is a measure of the `scrollTop` property of the `textarea` that had only the previous result in it. There wasn't enough content in the `textarea` to scroll, so the content was not scrolled at all. The `scrollTop` property, therefore, is zero. The `scrollLeft` property of the output is always zero because the `textarea` element is set to wrap any text that overflows the width of the element. No horizontal scrollbar appears in this case, and the `scrollLeft` property never changes.

Related Items: `clientLeft`, `clientTop` properties; `window.scroll()` method.

sourceIndex

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `sourceIndex` property returns the numeric index (zero-based) of the object within the entire document, which is the group of all elements in the document.

Example

While the operation of this property is straightforward, the sequence of elements exposed by the `document.all` property may not be. To that end, you can use The Evaluator (Chapter 13) to experiment in IE4+ with the values that the `sourceIndex` property returns to see how the index values of the `document.all` collection follow the source code.

To begin, reload The Evaluator. Enter the following statement in the top text box to set a preinitialized global variable:

```
a = 0
```

When you evaluate this expression, a zero should appear in the Results box. Next, enter the following statement into the top text box:

```
document.all[a].tagName + " [" + a++ + "]"
```

There are a lot of plus signs in this statement, so be sure you enter it correctly. As you successively evaluate this statement (repeatedly click the Evaluate button), the global variable (`a`) is incremented, thus enabling you to "walk through" the elements in source code order. The `sourceIndex` value for each HTML tag appears in square brackets in the Results box. You generally begin with the following sequence:

```
html [0]
head [1]
title [2]
```

You can continue until there are no more elements, at which point an error message appears because the value of `a` exceeds the number of elements in the `document.all` array. Compare your findings against the HTML source code view of The Evaluator.

Related Item: `item()` method.

style

Value: `style` object reference. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `style` property is the gateway to an element's stylesheet settings. The property's value is a `style` object whose properties enable you to read and write the stylesheet settings for the element. While scripts do not usually manipulate the `style` object as a whole, it is quite common in a Dynamic HTML page for scripts to get or set multiple properties of the `style` object to effect animation, visibility, and all appearance parameters of the element. Note that style properties returned through this object are only those that are explicitly set by the element's `style` attribute or by script.

You can find significant differences in the breadth of properties of the `style` object in different versions of IE and NN. See Chapter 26 for more details on the `style` object.

Example

Most of the action with the `style` property has to do with the `style` object's properties, so you can use The Evaluator here to simply explore the lists of `style` object properties available on as many DHTML-compatible browsers as you have running. To begin, enter the following statement into the lower, one-line text box to inspect the `style` property for the `document.body` object:

```
document.body.style
```

Now inspect the `style` property of the table element that is part of the original version of The Evaluator. Enter the following statement into the lower text box:

```
document.getElementById("myTable").style
```

In both cases, the values assigned to the `style` object's properties are quite limited by default.

Related Items: `currentStyle`, `runtimeStyle` properties; `style` object (Chapter 26).

tabIndex

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `tabIndex` property controls where in the tabbing sequence the current object receives focus. This property obviously applies only to elements that can receive focus. IE5+ permits giving focus to more elements than most other browsers; but for all browsers compatible with this property, the primary elements for which you may want to control focus (namely form input elements) are covered.

In general, browsers treat form elements as focusable elements by default. Nonform elements usually don't receive focus unless you specifically set their `tabIndex` properties (or `tabindex` tag attributes). If you set the `tabIndex` property of one form element to `1`, that element is first in the tabbing order. Meanwhile, the rest fall into source code tabbing order on successive presses of the Tab key. If you set two elements to, say, `1`, the tabbing proceeds in source code order for those two elements and then onto the rest of the elements in source code order starting with the top of the page.

In Internet Explorer, you can remove an element from tabbing order entirely by setting its `tabIndex` property to `-1`. Users can still click those elements to make changes to form element settings, but tabbing bypasses the element.

Example

Listing 15-16 contains a sample script that demonstrates how to control the tab order of a form via the `tabIndex` property. This example not only demonstrates the way you can modify the tabbing behavior of a form on the fly but also how to force form elements out of the tabbing sequence entirely in IE. In this page, the upper form (named `lab`) contains four elements. Scripts invoked by buttons in the lower form control the tabbing sequence. Notice that the `tabindex` attributes of all lower form elements are set to `-1`, which means that these control buttons are not part of the tabbing sequence in IE.

When you load the page, the default tabbing order for the `lab` form control elements (default setting of zero) takes charge. If you start pressing the Tab key, the precise results at first depend on the browser you use. In IE, the Address field is first selected; next the Tab sequence gives focus to the window (or frame, if this page were in a frameset); finally the tabbing reaches the `lab` form. Continue pressing the Tab key and watch how the browser assigns focus to each of the element types. In NN6+, however, you must click anywhere on the content to get the Tab key to start working on form controls.

The sample script inverts the tabbing sequence with the help of a `for` loop that initializes two variables that work in opposite directions as the looping progresses. This gives the last element the lowest `tabIndex` value. The `skip2()` function simply sets the `tabIndex` property of the second text box to `-1`, removing it from the tabbing entirely (IE only). Notice, however, that you can click in the field and still enter text. (See the `disabled` property earlier in this chapter to see how to prevent field editing.) NN6+ does not provide a `tabIndex` property setting that forces the browser to skip over a form control. You should disable the control instead.

Listing 15-16: **Controlling the tabIndex Property**

```
<html>
   <head>
      <title>tabIndex Property</title>
      <script type="text/javascript">
      function invert() {
         var form = document.lab;
         for (var i = 0, j = form.elements.length; i < form.elements.length;
            i++, j--) {
            form.elements[i].tabIndex = j;
         }
      }

      function skip2() {
         if (navigator.userAgent.indexOf("MSIE") != -1) {
            document.lab.text2.tabIndex = -1;
         } else {
            alert("Not available.");
         }
      }

      function resetTab() {
         var form = document.lab;
         for (var i = 0; i < form.elements.length; i++) {
```

```
            form.elements[i].tabIndex = 0;
         }
      }
      </script>
   </head>
   <body>
      <h1>tabIndex Property Lab</h1>
      <hr />
      <form name="lab">
         Text box no. 1: <input type="text" name="text1" /><br />
         Text box no. 2: <input type="text" name="text2" /><br />
         <input type="button" value="A Button" /><br />
         <input type="checkbox" />And a checkbox
      </form>
      <hr />
      <form name="control">
         <input type="button" value="Invert Tabbing Order" tabindex="-1"
         onclick="invert()" /><br />
         <input type="button" value="Skip Text box no. 2 (IE Only)"
         tabindex="-1" onclick="skip2()" /><br />
         <input type="button" value="Reset to Normal Order" tabindex="-1"
         onclick="resetTab()" />
      </form>
   </body>
</html>
```

The final function, `resetTab()`, sets the `tabIndex` property value to zero for all `lab` form elements; this restores the default order.

Related Items: `blur()`, `focus()` methods.

tagName

Value: String. Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `tagName` property returns a string of the HTML or XML tag name belonging to the object. All `tagName` values are returned in all uppercase characters, even if the source code is written in all lowercase or a mixture. This consistency makes it easier to perform string comparisons. For example, you can create a generic function that contains a `switch` statement to execute actions for some tags and not others. The skeleton of such a function looks like the following:

```
function  processObj(objRef) {
   switch (objRef.tagName) {
   case "TR":
      [statements to deal with table row object]
      break;
   case "TD":
      [statements to deal with table cell object]
      break;
   case "COLGROUP":
      [statements to deal with column group object]
      break;
   default:
      [statements to deal with all other object types]
   }
}
```

Example

You can also see the `tagName` property in action in the example associated with the `sourceIndex` property discussed earlier. In that example, the `tagName` property is read from a sequence of objects in source code order.

Related Items: `nodeName` property; `getElementsByTagName()` method.

tagUrn

(See `scopeName`)

title

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The W3C standard states that you should use the `title` property (and `title` attribute) in an "advisory" role. Most browsers (although not Safari 1.0) interpret this role as text assigned to tooltips that pop up momentarily while the cursor rests atop an element. The advantage of having this property available for writing is that your scripts can modify an element's tooltip text in response to other user interaction on the page. A tooltip can provide brief help about the behavior of icons or links on the page. It can also convey a summary of key facts from the destination of a link, thus enabling a visitor to see vital information without having to navigate to the other page.

As with setting the status bar, I don't recommend using tooltips for conveying mission-critical information to the user. Not all users are patient enough to let the pointer pause for the tooltip to appear. On the other hand, a user may be more likely to notice a tooltip once it appears rather than a status bar message (even though the latter appears instantaneously).

Example

Listing 15-17 provides a glimpse at how you can use the `title` property to establish tooltips for a page. A simple paragraph element has its `title` attribute set to `"First Time!"`, which is what the tooltip displays if you roll the pointer atop the paragraph and pause after the page loads. But an `onmouseover` event handler for that element increments a global variable counter in the script, and the `title` property of the paragraph object is modified with each mouseover action. The `count` value is made part of a string assigned to the `title` property. Notice that there is not a live connection between the `title` property and the variable; instead, the new value explicitly sets the `title` property.

Listing 15-17: Controlling the title Property

```
<html>
   <head>
      <title>title Property</title>
      <script type="text/javascript">
      // global counting variable
      var count = 0;

      function setToolTip(elem) {
         elem.title = "You have previously rolled atop this paragraph " +
            count + " time(s).";
      }

      function incrementCount(elem) {
```

```
            count++;
            setToolTip(elem);
         }
         </script>
      </head>
      <body>
         <h1>title Property Lab</h1>
         <hr />
         <p id="myP" title="First Time!" onmouseover="incrementCount(this)">Roll
            the mouse over this paragraph a few times.<br />
            Then pause atop it to view the tooltip.</p>
      </body>
   </html>
```

Related Item: `window.status` property.

uniqueID

Value: String. Read-Only
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

You can let the WinIE5+ browser generate an identifier (`id` property) for a dynamically generated element on the page with the aid of the `uniqueID` property. You should use this feature with care because the ID it generates at any given time may differ from the ID generated the next time the element is created in the page. Therefore, you should use the `uniqueID` property when your scripts require an unknown element to have an `id` property but the algorithms are not expecting any specific identifier.

To guarantee that an element gets only one ID assigned to it while the object exists in memory, assign the value via the `uniqueID` property of that same object — not some other object. Once you retrieve the `uniqueID` property of an object, the property's value stays the same no matter how often you access the property again. In general, you assign the value returned by the `uniqueID` property to the object's `id` property for other kinds of processing. (For example, the parameter of a `getElementById()` method requires the value assigned to the `id` property of an object.)

Example

Listing 15-18 demonstrates the recommended syntax for obtaining and applying a browser-generated identifier for an object. After you enter some text into the text box and click the button, the `addRow()` function appends a row to the table. The left column displays the identifier generated via the table row object's `uniqueID` property. IE5+ generates identifiers in the format `"ms__id`*n*`"`, where *n* is an integer starting with zero for the current browser session. Because the `addRow()` function assigns `uniqueID` values to the row and the cells in each row, the integer for each row is three greater than the previous one. There is no guarantee that future generations of the browser will follow this format, so do not rely on the format or sequence in your scripts.

Listing 15-18: Using the uniqueID Property

```
<html>
   <head>
      <title>Inserting an WinIE5+ Table Row</title>
      <script type="text/javascript">
```

Continued

Listing 15-18 *(continued)*

```
      function addRow(item1) {
         if (item1) {
            // assign long reference to shorter var name
            var theTable = document.getElementById("myTable");
            // append new row to the end of the table
            var newRow = theTable.insertRow(theTable.rows.length);
            // give the row its own ID
            newRow.id = newRow.uniqueID;

            // declare cell variable
            var newCell;

            // an inserted row has no cells, so insert the cells
            newCell = newRow.insertCell(0);
            // give this cell its own id
            newCell.id = newCell.uniqueID;
            // display the row's id as the cell text
            newCell.innerText = newRow.id;
            newCell.bgColor = "yellow"
            // re-use cell var for second cell insertion
            newCell = newRow.insertCell(1);
            newCell.id = newCell.uniqueID;
            newCell.innerText = item1;
         }
      }
      </script>
   </head>
   <body>
      <table id="myTable" border="1">
         <tr>
            <th>Row ID</th>
            <th>Data</th>
         </tr>
         <tr id="firstDataRow">
            <td>firstDataRow</td>
            <td>Fred</td>
         </tr>
         <tr id="secondDataRow">
            <td>secondDataRow</td>
            <td>Jane</td>
         </tr>
      </table>
      <hr />
      <form>
         Enter text to be added to the table:<br />
         <input type="text" name="input" size="25" /><br />
         <input type='button' value='Insert Row'
         onclick='addRow(this.form.input.value)' />
      </form>
   </body>
</html>
```

Related Items: `id` property; `getElementById()` method.

Methods

addBehavior("*URL*")

Returns: Integer ID.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `addBehavior()` method imports an external Internet Explorer behavior and attaches it to the current object, thereby extending the properties and/or methods of that object. See Chapter 48 on the CD-ROM for details on IE behaviors. The sole parameter of the `addBehavior()` method is a URL pointer to the behavior component's code. This component may be in an external file (with an `.htc` extension), in which case the parameter can be a relative or absolute URL. IE also includes a library of built-in (default) behaviors, whose URLs are in the following format:

```
#default#behaviorName
```

Here, *`behaviorName`* is one of the default behaviors (see Chapter 48 on the CD-ROM). If the behavior is imported into the document via the `object` tag, the `addBehavior()` method parameter is the ID of that element in the following format:

```
#objectID
```

When you add a behavior, the loading of the external code occurs asynchronously. This means that even though the method returns a value instantly, the behavior is not necessarily ready to work. Only when the behavior is fully loaded can it respond to events or allow access to its properties and methods. Behaviors loaded from external files observe domain security rules.

Example

Listing 15-19a shows what a behavior file looks like. It is the file used to demonstrate the `addBehavior()` method in Listing 15-19b. The behavior component and the HTML page that loads it must come from the same server and domain; they also must load via the same protocol (for example, `http://`, `https://`, and `file://` are mutually exclusive, mismatched protocols).

Listing 15-19a: **The makeHot.htc Behavior Component**

```
<PUBLIC:ATTACH EVENT="onmousedown" ONEVENT="makeHot()" />
<PUBLIC:ATTACH EVENT="onmouseup" ONEVENT="makeNormal()" />
<PUBLIC:PROPERTY NAME="hotColor" />
<PUBLIC:METHOD NAME="setHotColor" />
<SCRIPT LANGUAGE="JScript">
var oldColor;
var hotColor = "red";

function setHotColor(color) {
   hotColor = color;
}

function makeHot() {
   if (event.srcElement == element) {
      oldColor = style.color;
      runtimeStyle.color = hotColor;
   }
}
```

Continued

Listing 15-19a *(continued)*

```
function makeNormal() {
   if (event.srcElement == element) {
      runtimeStyle.color = oldColor   ;
   }
}
</SCRIPT>
```

The object to which the component is attached is a simple paragraph object, shown in Listing 15-19b. When the page loads, the behavior is not attached so clicking the paragraph text has no effect.

When you turn on the behavior by invoking the `turnOn()` function, the `addBehavior()` method attaches the code of the `makeHot.htc` component to the `myP` object. At this point, the `myP` object has one more property, one more method, and two more event handlers that are written to be made public by the component's code. If you want the behavior to apply to more than one paragraph in the document, you have to invoke the `addBehavior()` method for each paragraph object.

After the behavior file is instructed to start loading, the `setInitialColor()` function is called to set the new color property of the paragraph to the user's choice from the `select` list. But this can happen only if the component is fully loaded. Therefore, the function checks the `readyState` property of `myP` for completeness before invoking the component's function. If IE is still loading the component, the function is invoked again in 500 milliseconds.

As long as the behavior is loaded, you can change the color used to turn the paragraph "hot." The function first ensures that the component is loaded by checking that the object has the new color property. If it does, then (as a demonstration of how to expose and invoke a component method) the method of the component is invoked. You can also simply set the property value.

Listing 15-19b: **Using addBehavior() and removeBehavior()**

```
<html>
   <head>
      <title>addBehavior() and removeBehavior() Methods</title>
      <script type="text/javascript">
      var myPBehaviorID;

      function turnOn() {
         myPBehaviorID =
            document.getElementById("myP").addBehavior("makeHot.htc");
         setInitialColor();
      }

      function setInitialColor() {
         if (document.getElementById("myP").readyState == "complete") {
            var select = document.forms[0].colorChoice;
            var color = select.options[select.selectedIndex].value;
            document.getElementById("myP").setHotColor(color);
         } else {
            setTimeout("setInitialColor()", 500);
```

```
        }
      }

      function turnOff() {
         document.getElementById("myP").removeBehavior(myPBehaviorID);
      }

      function setColor(select, color) {
         if (document.getElementById("myP").hotColor) {
            document.getElementById("myP").setHotColor(color);
         } else {
            alert("This feature is not available. Turn on the Behavior
            first.");
            select.selectedIndex = 0;
         }
      }

      function showBehaviorCount() {
         var num = document.getElementById("myP").behaviorUrns.length;
         var msg = "The myP element has " + num + " behavior(s). ";
         if (num > 0) {
            msg += "Name(s): \r\n";
            for (var i = 0; i < num; i++) {
               msg += document.getElementById("myP").behaviorUrns[i] + "\r\n";
            }
         }
         alert(msg);
      }
      </script>
   </head>
   <body>
      <h1>addBehavior() and removeBehavior() Method Lab</h1>
      <hr />
      <p id="myP">This is a sample paragraph. After turning on the behavior, it
         will turn your selected color when you mouse down anywhere in this
         paragraph.</p>
      <form>
         <input type="button" value="Switch On Behavior" onclick="turnOn()" />
         Choose a 'hot' color: <select name="colorChoice"
         onchange="setColor(this, this.value)">
            <option value="red">red</option>
            <option value="blue">blue</option>
            <option value="cyan">cyan</option>
         </select>
         <br />
         <input type="button" value="Switch Off Behavior"
         onclick="turnOff()" />
         <p><input type="button" value="Count the URNs"
            onclick="showBehaviorCount()" /></p>
      </form>
   </body>
</html>
```

To turn off the behavior, the `removeBehavior()` method is invoked. Notice that the `removeBehavior()` method is associated with the `myP` object, and the parameter is the ID of the behavior added earlier. If you associate multiple behaviors with an object, you can remove one without disturbing the others because each has its own unique ID.

Related Items: `readyState` property; `removeBehavior()` method; behaviors (Chapter 48 on the CD-ROM).

`addEventListener("eventType", listenerFunc, useCapture)`
`removeEventListener("eventType", listenerFunc, useCapture)`

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The W3C DOM's event mechanism accommodates both event bubbling and trickling (see Chapter 25). While the new mechanism supports the long-standing notion of binding an event to an element by way of HTML attributes (for example, the old `onclick` event handler), it encourages binding events by registering an event listener with an element. (In browsers that support the W3C event model, other ways of binding events — such as event handler attributes — are internally converted to registered events.)

To tell the DOM that an element should "listen" for a particular kind of event, use the `addEventListener()` method on the element object. The method requires three parameters. The first is a string version of the event type for which the element should listen. Event type strings do not include the well-used "on" prefix of event handlers. Instead, the names consist only of the event and are usually in all lowercase (except for some special system-wide events preceded by DOM). Table 15-7 shows all the events recognized by the W3C DOM specification (including some new DOM ones that are not yet implemented in browsers).

Table 15-7: W3C DOM Event Listener Types

abort	error
blur	focus
change	load
click	mousedown
DOMActivate	mousemove
DOMAttrModified	mouseout
DOMCharacterDataModified	mouseover
DOMFocusIn	mouseup
DOMFocusOut	reset
DOMNodeInserted	resize
DOMNodeInsertedIntoDocument	scroll
DOMNodeRemoved	select
DOMNodeRemovedFromDocument	submit
DOMSubtreeModified	unload

Note that the event types specified in the DOM Level 2 are more limited than the wide range of events defined in IE4+. Also, the W3C temporarily tabled the issue of keyboard events until DOM Level 3. Fortunately, most W3C-compatible browsers implement keyboard events in a fashion that likely will appear as part of the W3C DOM Level 3.

The second parameter of the `addEventListener()` method is a reference to the JavaScript function to be invoked. This is the same form used to assign a function to an event property of an object (for example, *objReference*`.onclick = `*someFunction*), and it should *not* be a quoted string. This approach also means that you cannot specify parameters in the function

call. Therefore, functions that need to reference forms or form control elements must build their own references (with the help of the event object's property that says which object is the event's target).

By default, the W3C DOM event model has events bubble upward through the element container hierarchy starting with the target object of the event (for example, the button being clicked). However, if you specify `true` for the third parameter of the `addEventListener()` method, event capture is enabled for this particular event type whenever the current object is the event target. This means that any other event type targeted at the current object bubbles upward unless it, too, has an event listener associated with the object and the third parameter is set to `true`.

Using the `addEventListener()` method requires that the object to which it is attached already exists. Therefore, you most likely will use the method inside an initialization function triggered by the `onload` event handler for the page. (The `document` object can use `addEventListener()` for the load event immediately because the `document` object exists early in the loading process.)

A script can also eliminate an event listener that was previously added by script. The `removeEventListener()` method takes the same parameters as `addEventListener()`, which means that you can turn off one listener without disturbing others. In fact, because you can add two listeners for the same event and listener function (one set to capture and one not — a rare occurrence, indeed), the three parameters of the `removeEventListener()` enable you to specify precisely which listener to remove from an object.

Unlike the event capture mechanism of NN4, the W3C DOM event model does not have a "global" capture mechanism for an event type regardless of target. And with respect to Internet Explorer, the `addEventListener()` method is closely analogous to the IE5+ `attachEvent()` method. Also, event capture in IE5+ is enabled via the separate `setCapture()` method. Both the W3C and IE event models use their own syntaxes to bind objects to event handling functions, so the actual functions may be capable of serving both models with browser version branching required only for event binding. See Chapter 25 for more about event handling with these two event models.

Example

Listing 15-20 provides a compact workbench to explore and experiment with the basic W3C DOM event model. When the page loads, no event listeners are registered with the browser (except for the control buttons, of course). But you can add an event listener for a `click` event in bubble and/or capture mode to the `body` element or the `p` element that surrounds the `span` holding the line of text. If you add an event listener and click the text, you see a readout of the element processing the event and information indicating whether the event phase is bubbling (3) or capture (1). With all event listeners engaged, notice the sequence of events being processed. Remove listeners one at a time to see the effect on event processing.

Listing 15-20: **W3C Event Lab**

```
<html>
   <head>
      <title>W3C Event Model Lab</title>
      <style type="text/css">
      td {text-align:center}
      </style>
```

Continued

Listing 15-20 *(continued)*

```
   <script type="text/javascript">
   // add event listeners
   function addBubbleListener(elemID) {
      document.getElementById(elemID).addEventListener("click", reportEvent,
         false);
   }
   function addCaptureListener(elemID) {
      document.getElementById(elemID).addEventListener("click", reportEvent,
         true);
   }
   // remove event listeners
   function removeBubbleListener(elemID)  {
      document.getElementById(elemID).removeEventListener("click",
         reportEvent, false);
   }
   function removeCaptureListener(elemID) {
      document.getElementById(elemID).removeEventListener("click",
         reportEvent, true);
   }
   // display details about any event heard
   function reportEvent(evt) {
      var elem = (evt.target.nodeType == 3) ? evt.target.parentNode :
         evt.target;
      if (elem.id == "mySPAN") {
         var msg = "Event processed at " + evt.currentTarget.tagName +
         " element (event phase = " + evt.eventPhase + ").\n";
         document.controls.output.value += msg;
      }
   }
   // clear the details textarea
   function clearTextArea() {
      document.controls.output.value = "";
   }
   </script>
</head>
<body id="myBODY">
   <h1>W3C Event Model Lab</h1>
   <hr />
   <p id="myP"><span id="mySPAN">This paragraph (a SPAN element nested
   inside a P element) can be set to listen for "click" events.</span></p>
   <hr />
   <form name="controls" id="controls">
      <p>Examine click event characteristics: <input type="button"
         value="Clear" onclick="clearTextArea()" /><br />
         <textarea name="output" cols="80" rows="6" wrap="virtual">
         </textarea></p>
      <table cellpadding="5" border="1">
         <caption style="font-weight:bold">Control Panel</caption>
         <tr style="background-color:#ffff99">
            <td rowspan="2">"Bubble"-type click listener:</td>
            <td><input type="button" value="Add to BODY" onclick=
               "addBubbleListener('myBODY')" /></td>
            <td><input type="button" value="Remove from BODY" onclick=
               "removeBubbleListener('myBODY')" /></td>
         </tr>
         <tr style="background-color:#ffff99">
```

```
                <td><input type="button" value="Add to P" onclick=
                   "addBubbleListener('myP')" /></td>
                <td><input type="button" value="Remove from P" onclick=
                   "removeBubbleListener('myP')" /></td>
             </tr>
             <tr style="background-color:#ff9999">
                <td rowspan="2">"Capture"-type click listener:</td>
                <td><input type="button" value="Add to BODY" onclick=
                   "addCaptureListener('myBODY')" /></td>
                <td><input type="button" value="Remove from BODY" onclick=
                   "removeCaptureListener('myBODY')" /></td>
             </tr>
             <tr style="background-color:#ff9999">
                <td><input type="button" value="Add to P" onclick=
                   "addCaptureListener('myP')" /></td>
                <td><input type="button" value="Remove from P" onclick=
                   "removeCaptureListener('myP')" /></td>
             </tr>
          </table>
       </form>
   </body>
</html>
```

Related Items: `attachEvent()`, `detachEvent()`, `dispatchEvent()`, `fireEvent()`, `removeEventListener()` methods.

appendChild(*elementObject*)

Returns: Node object reference.
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The `appendChild()` method inserts an element or text node (defined by other code that comes before it) as the new, last child of the current element. Aside from the more obvious application of adding a new child element to the end of a sequence of child nodes, the `appendChild()` method is also practical for building element objects and their content before appending, replacing, or inserting the element into an existing document. The `document.createElement()` method generates a reference to an element of whatever tag name you assign as that method's parameter.

The `appendChild()` method returns a reference to the appended node object. This reference differs from the object that is passed as the method's parameter because the returned value represents the object as part of the document rather than as a freestanding object in memory.

Example

Listing 15-21 contains an example that shows how to use the `appendChild()` method in concert with `removeChild()` and `replaceChild()` to modify child elements in a document. Because many W3C DOM browsers treat source code carriage returns as text nodes (and thus child nodes of their parent), the HTML for the affected elements in Listing 15-21 is shown without carriage returns between elements.

The `append()` function creates a new `li` element and then uses the `appendChild()` method to attach the text box text as the displayed text for the item. The nested expression, `document.createTextNode(form.input.value)`, evaluates to a legitimate node that is appended to the new `li` item. All of this occurs before the new `li` item is added to the document. In the final statement of the function, `appendChild()` is invoked from the vantage point of the `ul` element — thus adding the `li` element as a child node of the `ul` element.

Invoking the `replaceChild()` method in the `replace()` function utilizes some of the same code. The main difference is that the `replaceChild()` method requires a second parameter: a reference to the child element to be replaced. This demonstration replaces the final child node of the `ul` list, so the function takes advantage of the `lastChild` property of all elements to get a reference to that final nested child. That reference becomes the second parameter to `replaceChild()`.

Listing 15-21: **Various Child Methods**

```
<html>
   <head>
      <title>appendChild(), removeChild(), and replaceChild() Methods</title>
      <script type="text/javascript">
      function append(form) {
         if (form.input.value) {
            var newItem = document.createElement("LI");
            newItem.appendChild(document.createTextNode(form.input.value));
            document.getElementById("myUL").appendChild(newItem);
         }
      }

      function replace(form) {
         if (form.input.value) {
            var newItem = document.createElement("LI");
            var lastChild = document.getElementById("myUL").lastChild;
            newItem.appendChild(document.createTextNode(form.input.value));
            document.getElementById("myUL").replaceChild(newItem, lastChild);
         }
      }

      function restore() {
         var oneChild;
         var mainObj = document.getElementById("myUL");
         while (mainObj.childNodes.length > 2) {
            oneChild = mainObj.lastChild;
            mainObj.removeChild(oneChild);
         }
      }
      </script>
   </head>
   <body>
      <h1>Child Methods</h1>
      <hr />
      Here is a list of items:
      <ul id="myUL"><li>First Item</li><li>Second Item</li></ul>
      <form>
         Enter some text to add/replace in the list: <input type="text"
         name="input" size="30" /><br />
          <input type="button" value="Append to List"
         onclick="append(this.form)" /> <input type="button"
         value="Replace Final Item" onclick="replace(this.form)" /> <input
         type="button" value="Restore List" onclick="restore()" />
      </form>
   </body>
</html>
```

The final part of the demonstration uses the `removeChild()` method to peel away all children of the `ul` element until just the two original items are left standing. Again, the `lastChild` property comes in handy as the `restore()` function keeps removing the last child until only two remain.

Related Items: `removeChild()`, `replaceChild()` methods; nodes and children (Chapter 14).

`applyElement(elementObject[, type])`

Returns: Nothing.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `applyElement()` method enables you to insert a new element as the parent or child of the current object. An important feature of this method is that the new object is wrapped around the current object (if the new element is to become the parent) or the current object's content (if the new element is to become a child). When the new element becomes a child, all previous children are nested further by one generation to become immediate children of the new element. You can imagine how the resulting action of this method affects the containment hierarchy of the current element, so you must be careful in how you use the `applyElement()` method.

One parameter, a reference to the object to be applied, is required. This object may be generated from constructions such as `document.createElement()` or from one of the child or node methods that returns an object. The second parameter is optional, and it must be one of the following values:

Parameter Value	*Description*
`Outside`	New element becomes the parent of the current object
`Inside`	New element becomes the immediate child of the current object

If you omit the second parameter, the default value (`outside`) is assumed. Listing 15-22 shows how the `applyElement()` method is used both with and without default values.

Example

To help you visualize the impact of the `applyElement()` method with its different parameter settings, Listing 15-22 enables you to apply a new element (an `em` element) to a `span` element inside a paragraph. At any time, you can view the HTML of the entire `p` element to see where the `em` element is applied as well as its impact on the element containment hierarchy for the paragraph.

After you load the page, inspect the HTML for the paragraph before doing anything else. Notice the `span` element and its nested `font` element, both of which surround the one-word content. If you apply the `em` element inside the `span` element (click the middle button), the `span` element's first (and only) child element becomes the `em` element; the `font` element is now a child of the new `em` element.

Listing 15-22: **Using the applyElement() Method**

```
<html>
   <head>
      <title>applyElement() Method</title>
```

Continued

Listing 15-22 *(continued)*

```
      <script type="text/javascript">
      function applyOutside() {
         var newItem = document.createElement("EM");
         newItem.id = newItem.uniqueID;
         document.getElementById("mySpan").applyElement(newItem);
      }

      function applyInside() {
         var newItem = document.createElement("EM");
         newItem.id = newItem.uniqueID;
         document.getElementById("mySpan").applyElement(newItem, "inside");
      }

      function showHTML() {
         alert(document.getElementById("myP").outerHTML);
      }
      </script>
   </head>
   <body>
      <h1>applyElement() Method</h1>
      <hr />
      <p id="myP">A simple paragraph with a <span id="mySpan"><font
         size="+1">special</font></span> word in it.</p>
      <form>
         <input type="button" value="Apply &lt;EM&gt; Outside"
         onclick="applyOutside()" /> <input type="button"
         value="Apply &lt;EM&gt; Inside" onclick="applyInside()" /> <input
         type="button" value="Show &lt;P&gt; HTML..."
         onclick="showHTML()" /><br />
          <input type="button" value="Restore Paragraph"
         onclick="location.reload()" />
      </form>
   </body>
</html>
```

The visible results of applying the `em` element inside and outside the `span` element in this case are the same. But you can see from the HTML results that each element impacts the element hierarchy quite differently.

Related Items: `insertBefore()`, `appendChild()`, `insertAdjacentElement()` methods.

`attachEvent("`*`eventName`*`", `*`functionRef`*`)`
`detachEvent("`*`eventName`*`", `*`functionRef`*`)`

Returns: Boolean.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `attachEvent()` method is used primarily within code that specifies IE behaviors (see Chapter 48 on the CD-ROM). But you can also use it in regular scripting as yet another way to bind an event handler to an object. The following example characterizes the more typical approach to assigning an event handler:

```
myObject.onmousedown = setHilite;
```

The version with attachEvent() is as follows:

```
myObject.attachEvent("onmousedown", setHilite);
```

Both parameters are required. The first parameter is a string version (case-insensitive) of the event name. The second is a reference to the function to be invoked when the event fires for this object. A *function reference* is an unquoted, case-sensitive identifier for the function without any parentheses (which also means that you cannot pass parameters in this function call).

There is a subtle benefit to using attachEvent() over the event property binding approach. When you use attachEvent(), the method returns a Boolean value of true if the event binding succeeds. IE triggers a script error if the function reference fails, so don't rely on a returned value of false to catch these kinds of errors. Also, there is no validation that the object recognizes the event name.

If you have used attachEvent() to bind an event handler to an object's event, you can disconnect that binding with the detachEvent() method. The parameters are the same as for attachEvent(). The detachEvent() method cannot unbind events whose associations are established via tag attributes or event property settings.

The W3C DOM event model provides functionality similar to these IE-only methods: addEventListener() and removeEventListener().

Example

Use The Evaluator (Chapter 13) to create an anonymous function that is called in response to an onmousedown event of the first paragraph on the page. Begin by assigning the anonymous function to global variable a (already initialized in The Evaluator) in the upper text box:

```
a = new Function("alert('Function created at " + (new Date()) + "')")
```

The quote marks and parentheses can get jumbled easily, so enter this expression carefully. When you enter the expression successfully, the Results box shows the function's text. Now assign this function to the onmousedown event of the myP element by entering the following statement into the upper text box:

```
document.getElementById("myP").attachEvent("onmousedown", a)
```

The Results box displays true when successful. If you mouse down on the first paragraph, an alert box displays the date and time that the anonymous function was created (when the new Date() expression was evaluated).

Now, disconnect the event relationship from the object by entering the following statement into the upper text box:

```
document.getElementById("myP").detachEvent("onmousedown", a)
```

Related Items: addEventListener(), detachEvent(), dispatchEvent(), fireEvent(), removeEventListener() methods; Event binding (Chapter 14).

blur()
focus()

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The blur() method removes focus from an element, while the focus() method gives focus to an element. Even though the blur() and focus() methods have been around since the earliest scriptable browsers, not every focusable object has enjoyed these methods since the

beginning. Browsers prior to IE4 and NN6 limited these methods primarily to the window object and form control elements.

Windows

For window objects, the blur() method (NN3+, IE4+) pushes the referenced window to the back of all other open windows. If other browser suite windows (such as e-mail or news reader windows) are open, the window receiving the blur() method is placed behind these windows as well.

Caution

The window.blur() method does not adjust the stacking order of the current window in Mozilla-based browsers (thus the "Put Me in Back" button in Listing 15-23 doesn't work in those browsers). But a script in a window can invoke the focus() method of another window to bring that other window to the front (provided a scriptable linkage, such as the window.opener property, exists between the two windows).

The minute you create another window for a user in your Web site environment, you must pay attention to window layer management. With browser windows so easily activated by the slightest mouse click, a user can lose a smaller window behind a larger one in a snap. Most inexperienced users don't think to check the Windows taskbar or browser menu bar (if the browser is so equipped) to see if a smaller window is still open and then activate it. If that subwindow is important to your site design, then you should present a button or other device in each window that enables users to safely switch among windows. The window.focus() method brings the referenced window to the front of all the windows.

Rather than supply a separate button on your page to bring a hidden window forward, you should build your window-opening functions in such a way that if the window is already open, the function automatically brings that window forward (as shown in Listing 15-23). This removes the burden of window management from your visitors.

The key to success with this method is making sure that your references to the desired windows are correct. Therefore, be prepared to use the window.opener property to refer to the main window if a subwindow needs to bring the main window back into focus.

Form control elements

The blur() and focus() methods apply primarily to text-oriented form controls: text input, select, and textarea elements.

Just as a camera lens blurs when it goes out of focus, a text object "blurs" when it loses focus — when someone clicks or tabs out of the field. Under script control, blur() deselects whatever may be selected in the field, and the text insertion pointer leaves the field. The pointer does not proceed to the next field in tabbing order, as it does if you perform a blur by tabbing out of the field manually.

For a text object, having focus means that the text insertion pointer is flashing in that text object's field. Giving a field focus is like opening it up for human editing.

Setting the focus of a text field or textarea does not, by itself, enable you to place the cursor at any specified location in the field. The cursor usually appears at the beginning of the text. To prepare a field for entry to remove the existing text, use both the focus() and select() methods in series.

There is a caveat about using focus() and select() together to preselect the content of a text field for immediate editing: Many versions of Internet Explorer fail to achieve the desired results due to an internal timing problem. You can work around this problem (and remain

compatible with other browsers) by initiating the focus and selection actions through a `setTimeout()` method. See Chapter 43 on the CD-ROM on data validation for an example.

A common design requirement is to position the insertion pointer at the end of a text field or `textarea` so that a user can begin appending text to existing content immediately. This is possible in IE4+ with the help of the `TextRange` object. The following script fragment moves the text insertion pointer to the end of a `textarea` element whose ID is `myTextarea`:

```
var range = document.getElementById("myTextarea").createTextRange();
range.move("textedit");
range.select();
```

You should be very careful in combining `blur()` or `focus()` methods with `onblur` and `onfocus` event handlers — especially if the event handlers display alert boxes. Many combinations of these events and methods can cause an infinite loop in which it is impossible to dismiss the alert dialog box completely. On the other hand, there is a useful combination for older browsers that don't offer a `disabled` property for text boxes. The following text field event handler can prevent users from entering text in a text field:

```
onfocus = "this.blur()";
```

Some operating systems and browsers enable you to give focus to elements such as buttons (including radio and checkbox buttons) and hypertext links (encompassing both `a` and `area` elements). Typically, once such an element has focus, you can accomplish the equivalent of a mouse click on the element by pressing the spacebar on the keyboard. This is helpful for accessibility to those who have difficulty using a mouse.

An unfortunate side effect of button focus in Win32 environments is that the focus highlight (a dotted rectangle) remains around the button after a user clicks it and until another object gets focus. You can eliminate this artifact for browsers and objects that implement the `onmouseup` event handler by including the following event handler in your buttons:

```
onmouseup = "this.blur()";
```

IE5.5+ recognizes the often undesirable effect of that dotted rectangle and lets scripts set the `hideFocus` property of an element to `true` to keep that rectangle hidden while still giving the element focus. It is a trade-off for the user, however, because there is no visual feedback about which element has focus.

Other elements

For other kinds of elements that support the `focus()` method, you can bring an element into view in lieu of the `scrollIntoView()` method. Link (`a`) and `area` elements in Windows versions of IE display the dotted rectangle around them after a user brings focus to them. To eliminate that artifact, use the same

```
onmouseup = "this.blur()";
```

event handler (or IE5.5+ `hideFocus` property) as just described for form controls.

Example

Listing 15-23 contains an example of using the `focus()` and `blur()` methods to tinker with changing the focus of windows. This example creates a two-window environment; from each window, you can bring the other window to the front. The main window uses the object returned by `window.open()` to assemble the reference to the new window. In the subwindow (whose content is created entirely on the fly by JavaScript), `self.opener` is summoned to refer to the original window, while `self.blur()` operates on the subwindow itself. Blurring one window and focusing on another window yields the same result of sending the window to the back of the pile.

Listing 15-23: **The window.focus() and window.blur() Methods**

```
<html>
   <head>
      <title>Window Focus() and Blur()</title>
      <script type="text/javascript">
      // declare global variable name
      var newWindow = null;

      function makeNewWindow() {
         // check if window already exists
         if (!newWindow || newWindow.closed) {
            // store new window object in global variable
            newWindow = window.open("","","width=250,height=250");
            // pause briefly to let IE3 window finish opening
            setTimeout("fillWindow()",100);
         } else {
            // window already exists, so bring it forward
            newWindow.focus();
         }
      }

      // assemble new content and write to subwindow
      function fillWindow() {
         var newContent = "<html><head><title>Another Sub
         Window<\/title><\/head>";
         newContent += "<body bgColor='salmon'>";
         newContent += "<h1>A Salmon-Colored Subwindow.<\/h1>";
         newContent += "<form><input type='button' value='Bring Main to Front'
         onclick='self.opener.focus()'>";
         newContent += "<form><input type='button' value='Put Me in Back'
         onclick='self.blur()'>";
         newContent += "<\/form><\/body><\/html>";
         // write HTML to new window document
         newWindow.document.write(newContent);
         newWindow.document.close();
      }
      </script>
   </head>
   <body>
      <h1>Window focus() and blur() Methods</h1>
      <hr />
      <form>
         <input type="button" name="newOne" value="Show New Window"
         onclick="makeNewWindow()" />
      </form>
   </body>
</html>
```

A key ingredient to the success of the `makeNewWindow()` function in Listing 15-23 is the first conditional expression. Because `newWind` is initialized as a `null` value when the page loads, that is its value the first time through the function. But after you open the subwindow the first time, `newWind` is assigned a value (the subwindow object) that remains intact even if the user closes the window. Thus, the value doesn't revert to `null` by itself. To catch the possibility that

the user has closed the window, the conditional expression also sees if the window is closed. If it is, a new subwindow is generated, and that new window's reference value is reassigned to the `newWind` variable. On the other hand, if the window reference exists and the window is not closed, the `focus()` method brings that subwindow to the front.

You can see the `focus()` method for a text object in action in Chapter 25's description of the `select()` method for text objects.

Related Items: `window.open()`, `document.formObject.textObject.select()` methods.

`clearAttributes()`

Returns: Nothing.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `clearAttributes()` method removes all attributes from an element except the `name` and `id` values. Thus, styles and event handlers are removed, as are custom attributes assigned in either the HTML source code or later by script. You should know that the `clearAttributes()` method does not alter the length of the element's `attributes` collection because the collection always contains all possible attributes for an element. (See the `attributes` property for elements earlier in this chapter.)

This method is handy if you wish to construct an entirely new set of attributes for an element and prefer to start out with a blank slate. Be aware, however, that unless your scripts immediately assign new attributes to the element, the appearance of the element reverts to its completely unadorned form until you assign new attributes. This means that even positioned elements find their way back to their source code order until you assign a new positioning style. If you simply want to change the value of one or more attributes of an element, it is faster to use the `setAttribute()` method or adjust the corresponding properties.

To accomplish a result in NN6+/Moz1+ that simulates that of IE5+'s `clearAttributes()`, you must iterate through all attributes of an element and remove those attributes (via the `removeAttribute()` method) whose names are other than `id` and `name`.

Example

Use The Evaluator (Chapter 13) to examine the attributes of an element before and after you apply `clearAttributes()`. To begin, display the HTML for the table element on the page by entering the following statement into the upper text field:

```
myTable.outerHTML
```

Notice the attributes associated with the `<table>` tag. Look at the rendered table to see how attributes such as `border` and `width` affect the display of the table. Now, enter the following statement in the top text box to remove all removable attributes from this element:

```
myTable.clearAttributes()
```

First, look at the table. The border is gone, and the table is rendered only as wide as is necessary to display the content with no cell padding. Finally, view the results of the `clearAttributes()` method in the `outerHTML` of the table again:

```
myTable.outerHTML
```

The source code file has not changed, but the object model in the browser's memory reflects the changes you made.

Related Items: `attributes` property; `getAttribute()`, `setAttribute()`, `removeAttribute()`, `mergeAttributes()`, and `setAttributeNode()` methods.

click()

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN2+, Moz1+, Safari1+

The `click()` method lets a script perform nearly the same action as clicking an element. Prior to NN4 and IE4, the `click()` method invoked on a button did not trigger the `onclick` event handler for the object. This has significant impact if you expect the `onclick` event handler of a button to function even if a script performs the "click." For earlier browser versions, you have to invoke the event handler statements directly. Also, just because a script is "clicking" a button, not all buttons in all platforms change their appearance in response. For example, NN4 on the Mac does not change the state of a checkbox when clicked remotely.

If you want to script the action of "clicking" a button, you can safely invoke the resulting event handler function directly. And if the element is a radio button or checkbox, handle the change of state directly (for example, set the `checked` property of a checkbox) rather than expect the browser to take care of it for you.

Example

Use The Evaluator (Chapter 13) to experiment with the `click()` method. The page includes various types of buttons at the bottom. You can "click" the checkbox, for example, by entering the following statement in the topmost text field:

```
document.myForm2.myCheckbox.click()
```

If you use a recent browser version, you most likely can see the checkbox change states between checked and unchecked each time you execute the statement.

Related Item: `onclick` event handler.

cloneNode(*deepBoolean*)

Returns: Node object reference.
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The `cloneNode()` method makes an exact copy of the current node object. This copy does not have a parent node or other relationship with any element once the copy exists (of course, the original node remains in place). The clone also does not become part of the document's object model (the node tree) unless you explicitly insert or append the node somewhere on the page. The copy includes all element attributes, including the `id` attribute. Because the value returned by the `cloneNode()` method is a genuine `Node` object, you can operate on it with any `Node` object methods while it is still in the non-document object state.

The Boolean parameter of the `cloneNode()` method controls whether the copy of the node includes all child nodes (`true`) or just the node itself (`false`). For example, if you clone a paragraph element by itself, the clone consists only of the raw element (equivalent of the tag pair, including attributes in the start tag) and none of its content. But including child nodes makes sure that all content within that paragraph element is part of the copy. This parameter is optional in IE5 (defaulting to `false`), but it is required in other W3C-compatible browsers.

Example

Use The Evaluator (Chapter 13) to clone, rename, and append an element found in The Evaluator's source code. Begin by cloning the paragraph element named `myP` along with all of its content. Enter the following statement into the topmost text field:

```
a = document.getElementById("myP").cloneNode(true)
```

The variable a now holds the clone of the original node, so you can change its id attribute at this point by entering the following statement:

```
a.setAttribute("id", "Dolly")
```

If you want to see the properties of the cloned node, enter a into the lower text field. The precise listing of properties you see depends on the browser you're using; in either case, you should be able to locate the id property, whose value is now Dolly.

As a final step, append this newly named node to the end of the body element by entering the following statement into the topmost text field:

```
document.body.appendChild(a)
```

You can now scroll down to the bottom of the page and see a duplicate of the content. But because the two nodes have different id attributes, they cannot confuse scripts that need to address one or the other.

Related Items: Node object (Chapter 14); appendChild(), removeChild(), removeNode(), replaceChild(), and replaceNode() methods.

componentFromPoint(*x*,*y*)

Returns: String.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The componentFromPoint() method assists in some event-related tasks. You can use it for a kind of collision detection (in other words, to determine whether an event occurs inside or outside of a particular element). If the element has scrollbars, the method can provide additional information about the event such as precisely which component of the scrollbar the user activates.

A key aspect of this method is that you invoke it on any element that you want to use as the point of reference. For example, if you want to find out if a mouseup event occurs in an element whose ID is myTable, invoke the method as follows:

```
var result = document.getElementById("myTable").componentFromPoint(
   event.clientX, event.clientY);
```

Parameters passed to the method are *x* and *y* coordinates. These coordinates do not have to come from an event, but the most likely scenario links this method with an event of some kind. Mouse events (other than onclick) work best.

The value returned by the method is a string that provides details about where the coordinate point is with respect to the current element. If the coordinate point is inside the element's rectangle, the returned value is an empty string. Conversely, if the point is completely outside of the element, the returned value is the string "outside". For scrollbar pieces, the list of possible returned values is quite lengthy (as shown in Table 15-8).

Table 15-8: Returned Values for componentFromPoint()

Returned String	*Element Component at Coordinate Point*
scrollbarDown	Scrollbar down arrow
scrollbarHThumb	Scrollbar thumb on horizontal bar
scrollbarLeft	Scrollbar left arrow

Continued

Table 15-8 *(continued)*

Returned String	*Element Component at Coordinate Point*
`scrollbarPageDown`	Scrollbar page-down region
`scrollbarPageLeft`	Scrollbar page-left region
`scrollbarPageRight`	Scrollbar page-right region
`scrollbarPageUp`	Scrollbar page-up region
`scrollbarRight`	Scrollbar right arrow
`scrollbarUp`	Scrollbar up arrow
`scrollbarVThumb`	Scrollbar thumb on vertical bar
`handleBottom`	Resize handle at bottom
`handleBottomLeft`	Resize handle at bottom left
`handleBottomRight`	Resize handle at bottom right
`handleLeft`	Resize handle at left
`handleRight`	Resize handle at right
`handleTop`	Resize handle at top
`handleTopLeft`	Resize handle at top left
`handleTopRight`	Resize handle at top right

You do not have to use this method for most collision or event detection, however. The `event` object's `srcElement` property returns a reference to whatever object receives the event.

Example

Listing 15-24 demonstrates how the `componentFromPoint()` method can be used to determine exactly where a mouse event occurred. As presented, the method is associated with a `textarea` object that is specifically sized to display both vertical and horizontal scrollbars. As you click various areas of the `textarea` and the rest of the page, the status bar displays information about the location of the event with the help of the `componentFromPoint()` method.

The script utilizes a combination of the `event.srcElement` property and the `componentFromPoint()` method to help you distinguish how you can use each one for different types of event processing. The `srcElement` property is used initially as a filter to decide whether the status bar will reveal further processing about the `textarea` element's event details.

The `onmousedown` event handler in the `body` element triggers all event processing. IE events bubble up the hierarchy (and no events are cancelled in this page), so all `mousedown` events eventually reach the `body` element. Then, the `whereInWorld()` function can compare each `mousedown` event from any element against the text area's geography.

Listing 15-24: **Using the componentFromPoint() Method**

```
<html>
   <head>
      <title>componentFromPoint() Method</title>
      <script type="text/javascript">
```

```
      function whereInWorld(elem) {
         var x = event.clientX;
         var y = event.clientY;
         var component =
            document.getElementById("myTextarea").componentFromPoint(x,y);
         if (window.event.srcElement == document.getElementById("myTextarea")){
            if (component == "") {
               status = "mouseDown event occurred inside the element";
            } else {
               status = "mouseDown occurred on the element\'s " + component;
            }
         } else {
            status = "mouseDown occurred " + component + " of the element";
         }
      }
      </script>
   </head>
   <body onmousedown="whereInWorld()">
      <h1>componentFromPoint() Method</h1>
      <hr />
      <p>Tracking the mouseDown event relative to the textarea object. View
         results in status bar.</p>
      <form>
         <textarea name="myTextarea" wrap="off" cols="12" rows="4">
            This is Line 1
            This is Line 2
            This is Line 3
            This is Line 4
            This is Line 5
            This is Line 6
         </textarea>
      </form>
   </body>
</html>
```

Related Item: `event` object.

`contains(`*`elementObjectReference`*`)`

Returns: Boolean.
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `contains()` method reports whether the current object contains another known object within its HTML containment hierarchy. Note that this is not geographical collision detection of overlapping elements, but rather the determination of whether one element is nested somewhere within another.

The scope of the `contains()` method extends as deeply within the current object's hierarchy as is necessary to locate the object. In essence, the `contains()` method examines all of the elements that are part of an element's `all` array. Therefore, you can use this method as a shortcut replacement for a `for` loop that examines each nested element of a container for the existence of a specific element.

The parameter to the `contains()` method is a reference to an object. If you have only the element's ID as a string to go by, you can use the `document.getElementById()` method to generate a valid reference to the nested element.

***elementObject*.contains()**

An element always contains itself.

Example

Using The Evaluator (Chapter 13), see how the `contains()` method responds to the object combinations in each of the following statements as you enter them into the upper text box:

```
document.body.contains(document.all.myP)
document.all.myP.contains(document.all.item("myEM"))
document.all.myEM.contains(document.all.myEM)
document.all.myEM.contains(document.all.myP)
```

Feel free to test other object combinations within this page.

Related Items: `item()`, `document.getElementById()` methods.

`detachEvent()`

(See `attachEvent()`)

`dispatchEvent(`*`eventObject`*`)`

Returns: Boolean.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `dispatchEvent()` method allows a script to fire an event aimed at any object capable of supporting that event. This is the W3C event model way of generalizing mechanisms that earlier browsers sometimes mimic with object methods such as `click()` and `focus()`.

The process of generating one of these events is similar to the way a script generates a new node and inserts that node somewhere in the document object model. For events, however, the object that is created is an `Event` object, which is generated via the `document.createEvent()` method. An event generated in this manner is simply a specification about an event. Use properties of an event object to supply specifics about the event, such as its coordinates or mouse button. Then dispatch the event to a target object by invoking that target object's `dispatchEvent()` method and passing the newly created `Event` object as the sole parameter.

Interpreting the meaning of the Boolean value that the `dispatchEvent()` method returns is not straightforward. The browser follows the dispatched event through whatever event propagation is in effect for that object and event type (either bubbling or capture). If any of the event listener functions that are triggered by this dispatched event invoke the `preventDefault()` method, the `dispatchEvent()` method returns `false` to indicate that the event did not trigger the native action of the object; otherwise, the method returns `true`. Notice that this returned value indicates nothing about propagation type or how many event listeners run as a result of dispatching this event.

Although the `dispatchEvent()` method is implemented in NN6, the browser does not yet provide a way to generate new events from scratch. And if you attempt to redirect an existing event to another object via the `dispatchEvent()` method, the browser is prone to crashing. In other words, Mozilla-based browsers are much better candidates for scripts that utilize `dispatchEvent()`.

Example

Listing 15-25 demonstrates how to programmatically fire events using the W3C DOM `dispatchEvent()` method. Notice the syntax in the `doDispatch()` function for creating and

initializing a new mouse event, supported most reliably in Mozilla-based browsers. The behavior is identical to that of Listing 15-26, which demonstrates the IE5.5+ equivalent: `fireEvent()`.

Listing 15-25: **Using the dispatchEvent() Method**

```
<html>
   <head>
      <title></title>
      <style type="text/css">
      #mySPAN {font-style:italic}
      </style>
      <script type="text/javascript">
      // assemble a couple event object properties
      function getEventProps(evt) {
         var msg = "";
         var elem = evt.target;
         msg += "event.target.nodeName: " + elem.nodeName + "\n";
         msg += "event.target.parentNode: " + elem.parentNode.id + "\n";
         msg += "event button: " + evt.button;
         return msg;
      }

      // onClick event handlers for body, myP, and mySPAN
      function bodyClick(evt) {
         var msg = "Click event processed in BODY\n\n";
         msg += getEventProps(evt);
         alert(msg);
         checkCancelBubble(evt);
      }
      function pClick(evt) {
         var msg = "Click event processed in P\n\n";
         msg += getEventProps(evt);
         alert(msg);
         checkCancelBubble(evt);
      }
      function spanClick(evt) {
         var msg = "Click event processed in SPAN\n\n";
         msg += getEventProps(evt);
         alert(msg);
         checkCancelBubble(evt);
      }

      // cancel event bubbling if checkbox is checked
      function checkCancelBubble(evt) {
         if (document.controls.bubbleOn.checked) {
            evt.stopPropagation();
         }
      }

      // assign onClick event handlers to three elements
      function init() {
         document.body.onclick = bodyClick;
         document.getElementById("myP").onclick = pClick;
         document.getElementById("mySPAN").onclick = spanClick;
```

Continued

Listing 15-25 *(continued)*

```
      }

      // invoke fireEvent() on object whose ID is passed as parameter
      function doDispatch(objID, evt) {
         // create empty mouse event
         var newEvt = document.createEvent("MouseEvents");
         // initialize as click with button ID 3
         newEvt.initMouseEvent("click", true, true, window, 0, 0, 0,
            0, 0, false, false, false, false, 3, null);
         // send event to element passed as param
         document.getElementById(objID).dispatchEvent(newEvt);
         // don't let button clicks bubble
         evt.stopPropagation();
      }
      </script>
   </head>
   <body id="myBODY" onload="init()">
      <h1>fireEvent() Method</h1>
      <hr />
      <p id="myP">This is a paragraph <span id="mySPAN">(with a nested
         SPAN)</span> that receives click events.</p>
      <hr />
      <p><b>Control Panel</b></p>
      <form name="controls">
         <p><input type="checkbox" name="bubbleOn"
            onclick="event.stopPropagation()" />Cancel event bubbling.</p>
         <p><input type="button" value="Fire Click Event on BODY"
            onclick="doDispatch('myBODY', event)" /></p>
         <p><input type="button" value="Fire Click Event on myP"
            onclick="doDispatch('myP', event)" /></p>
         <p><input type="button" value="Fire Click Event on mySPAN"
            onclick="doDispatch('mySPAN', event)" /></p>
      </form>
   </body>
</html>
```

Related Item: `fireEvent()` method.

`fireEvent("eventType"[, eventObjectRef])`

Returns: Boolean.
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

While some objects have methods that emulate physical events (for example, the `click()` and `focus()` methods), WinIE5.5+ generalizes the mechanism by letting a script direct any valid event to any object. The `fireEvent()` method is the vehicle.

One required parameter is the event type, formatted as a string. IE event types are coded just like the property names for event handlers (for example, `onclick`, `onmouseover`, and so on).

A second, optional parameter is a reference to an existing `event` object. This object can be an event that some user or system action triggers (meaning that the `fireEvent()` method is in a function invoked by an event handler). The existing event can also be an object created by the IE5.5+ `document.createEventObject()` method. In either case, the purpose of providing an

existing `event` object is to set the properties of the `event` object that the `fireEvent()` method creates. The event type is defined by the method's first parameter, but if you have other properties to set (for example, coordinates or a keyboard key code), then those properties are picked up from the existing object. Here is an example of a sequence that creates a new `mousedown` event, stuffs some values into its properties, and then fires the event at an element on the page:

```
var newEvent = document.createEventObject();
newEvent.clientX = 100;
newEvent.clientY = 30;
newEvent.cancelBubble = false;
newEvent.button = 1;
document.getElementById("myElement").fireEvent("onmousedown", newEvent);
```

Events generated by the `fireEvent()` method are just like regular IE `window.event` objects, and they have several important `event` object properties that the browser presets. Importantly, `cancelBubble` is set to `false` and `returnValue` is set to `true` — just like a regular user- or system-induced event. This means that if you want to prevent event bubbling and/or prevent the default action of the event's source element, then the event handler functions must set these `event` object properties just like normal event handling in IE.

The `fireEvent()` method returns a Boolean value that the `returnValue` property of the event determines. If the `returnValue` property is set to `false` during event handling, the `fireEvent()` method returns `false`. Under normal processing, the method returns `true`.

Although the W3C DOM Level 2 event model includes the `dispatchEvent()` method to accommodate script-generated events (and `Event` object methods to create event objects), Microsoft has so far elected to ignore the standard recommendation. While there is some similarity between the basic operations of `fireEvent()` and `dispatchEvent()`, the two methods diverge significantly in advanced applications (for example, the way events can propagate and the W3C notion of an `Event` object).

Example

Listing 15-26 contains script code that shows how to programmatically fire events using the `fireEvent()` method. Three buttons in the example page enable you to direct a click event to each of the three elements that have event handlers defined for them. The events fired this way are artificial, generated via the `createEventObject()` method. For demonstration purposes, the `button` property of these scripted events is set to `3`. This property value is assigned to the `event` object that eventually gets directed to an element. With event bubbling left on, the events sent via `fireEvent()` behave just like the physical clicks on the elements. Similarly, if you disable event bubbling, the first event handler to process the event cancels bubbling, and no further processing of that event occurs. Notice that event bubbling is cancelled within the event handlers that process the event. To prevent the clicks of the checkbox and action buttons from triggering the `body` element's `onclick` event handlers, event bubbling is turned off for the buttons right away.

Listing 15-26: **Using the fireEvent() Method**

```
<html>
   <head>
      <title></title>
      <style type="text/css">
      #mySPAN {font-style:italic}
      </style>
```

Continued

Listing 15-26 *(continued)*

```
    <script type="text/javascript">
    // assemble a couple event object properties
    function getEventProps() {
       var msg = "";
       var elem = event.srcElement;
       msg += "event.srcElement.tagName: " + elem.tagName + "\n";
       msg += "event.srcElement.id: " + elem.id + "\n";
       msg += "event button: " + event.button;
       return msg;
    }

    // onClick event handlers for body, myP, and mySPAN
    function bodyClick() {
       var msg = "Click event processed in BODY\n\n";
       msg += getEventProps();
       alert(msg);
       checkCancelBubble();
    }
    function pClick() {
       var msg = "Click event processed in P\n\n";
       msg += getEventProps();
       alert(msg);
       checkCancelBubble();
    }
    function spanClick() {
       var msg = "Click event processed in SPAN\n\n";
       msg += getEventProps();
       alert(msg);
       checkCancelBubble();
    }

    // cancel event bubbling if checkbox is checked
    function checkCancelBubble() {
       event.cancelBubble = document.controls.bubbleOn.checked;
    }

    // assign onClick event handlers to three elements
    function init() {
       document.body.onclick = bodyClick;
       document.getElementById("myP").onclick = pClick;
       document.getElementById("mySPAN").onclick = spanClick;
    }

    // invoke fireEvent() on object whose ID is passed as parameter
    function doFire(objID) {
       var newEvt = document.createEventObject();
       newEvt.button = 3;
       document.all(objID).fireEvent("onclick", newEvt);
       // don't let button clicks bubble
       event.cancelBubble = true;
    }
    </script>
</head>
<body id="myBODY" onload="init()">
```

```
      <h1>fireEvent() Method</h1>
      <hr />
      <p id="myP">This is a paragraph <span id="mySPAN">(with a nested
         SPAN)</span> that receives click events.</p>
      <hr />
      <p><b>Control Panel</b></p>
      <form name="controls">
         <p><input type="checkbox" name="bubbleOn"
            onclick="event.cancelBubble=true" />Cancel event bubbling.</p>
         <p><input type="button" value="Fire Click Event on BODY"
            onclick="doFire('myBODY')" /></p>
         <p><input type="button" value="Fire Click Event on myP"
            onclick="doFire('myP')" /></p>
         <p><input type="button" value="Fire Click Event on mySPAN"
            onclick="doFire('mySPAN')" /></p>
      </form>
   </body>
</html>
```

Related Item: `dispatchEvent()` method.

`focus()`

(See `blur()`)

`getAdjacentText("position")`

Returns: String.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `getAdjacentText()` method enables you to extract copies of plain-text components of an element object (in other words, without any HTML tag information). The sole parameter is one of four case-insensitive string constant values that indicate from where, in relation to the current object, the text should be extracted. The values are:

Parameter Value	*Description*
`beforeBegin`	Text immediately in front of the element's tag, back to the preceding tag
`afterBegin`	Text that begins inside the element tag, up to the next tag (whether it be a nested element or the element's end tag)
`beforeEnd`	Text immediately in front of the element's end tag, back to the preceding tag (whether it be a nested element or the element's start tag)
`afterEnd`	Text immediately following the element's end tag, forward until the next tag

If the current object has no nested elements, then the `afterBegin` and `beforeEnd` versions both return the same as the object's `innerText` property. When the current object is encased immediately within another element (for example, a `td` element inside a `tr` element), there is no text before the element's beginning or after the element's end so these values are returned as empty strings.

The strings returned from this method are roughly equivalent to values of text fragment nodes in the W3C DOM, but IE5+ treats these data pieces only as string data types rather than as text node types. W3C DOM equivalents for the four versions are:

```
document.getElementById("objName").previousSibling.nodeValue
document.getElementById("objName").firstChild.nodeValue
document.getElementById("objName").lastChild.nodeValue
document.getElementById("objName").nextSibling.nodeValue
```

Example

Use The Evaluator (Chapter 13) to examine all four adjacent text possibilities for the `myP` and nested `myEM` elements in that document. Enter each of the following statements into the upper text box, and view the results:

```
document.getElementById("myP").getAdjacentText("beforeBegin")
document.getElementById("myP").getAdjacentText("afterBegin")
document.getElementById("myP").getAdjacentText("beforeEnd")
document.getElementById("myP").getAdjacentText("afterEnd")
```

The first and last statements return empty strings because the `myP` element has no text fragments surrounding it. The `afterBegin` version returns the text fragment of the `myP` element up to, but not including, the EM element nested inside. The `beforeEnd` string picks up after the end of the nested EM element and returns all text to the end of `myP`.

Now, see what happens with the nested `myEM` element:

```
document.getElementById("myEM").getAdjacentText("beforeBegin")
document.getElementById("myEM").getAdjacentText("afterBegin")
document.getElementById("myEM").getAdjacentText("beforeEnd")
document.getElementById("myEM").getAdjacentText("afterEnd")
```

Because this element has no nested elements, the `afterBegin` and `beforeEnd` strings are identical — the same value as the `innerText` property of the element.

Related Items: `childNodes`, `data`, `firstChild`, `lastChild`, `nextSibling`, `nodeValue`, and `previousSibling` properties.

getAttribute("*attributeName*"[, *caseSensitivity*])

Returns: See text.
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `getAttribute()` method returns the value assigned to a specific attribute of the current object. You can use this method as an alternative to retrieving properties of an object, particularly when your script presents you with the attribute name as a string (in contrast to a fully formed reference to an object and its property). Thus, the following example statements yield the same data:

```
var mult = document.getElementById("mySelect").multiple;
var mult = document.getElementById("mySelect").getAttribute("multiple");
```

Returned value types from `getAttribute()` are either strings (including attribute values assigned as unquoted numeric values) or Booleans (for example, the `multiple` property of a `select` element object).

The W3C DOM Level 2 standard recommends `getAttribute()` and `setAttribute()` for reading and writing element object attribute values, rather than reading and writing those values by way of their corresponding properties. While using these methods is certainly advisable for XML elements, the same DOM standard sends conflicting signals by defining all

kinds of properties for HTML element objects. Browsers, of course, will support access via properties well into the future, so don't feel obligated to change your ways just yet.

All browsers that support the `getAttribute()` method require one parameter, which is a string of the attribute name. By default, this parameter is not case-sensitive. Note that this has impact on custom attributes that you might assign to HTML or XML elements in your documents. Attribute names are automatically converted to lowercase when they are turned into properties of the object. Therefore, you must avoid reusing attribute names, even if you use different case letters in the source code assignments.

IE includes an optional extension to the method in the form of a second parameter that enables you to be more specific about the case-sensitivity of the first parameter. The default value of the second parameter is `false`, which means that the first parameter is not case-sensitive. A value of `true` makes the first parameter case-sensitive. This matters only if you use `setAttribute()` to add a parameter to an existing object and if the IE version of that method insists on case-sensitivity. The default behavior of `setAttribute()` respects the case of the attribute name. See also the discussion of the `setAttribute()` method later in this chapter with regard to `setAttribute()`'s influence over the IE `attributes` property.

Example

Use The Evaluator (Chapter 13) to experiment with the `getAttribute()` method for the elements in the page. For IE4, use the `document.all` notation. IE5+ and W3C browsers understand the W3C standard `getElementById()` method of addressing an element. You can enter the following sample statements into the top text box to view attribute values.

IE4:

```
document.all.myTable.getAttribute("width")
document.all.myTable.getAttribute("border")
```

IE5+/W3C:

```
document.getElementById("myTable").getAttribute("width")
document.getElementById("myTable").getAttribute("border")
```

Related Items: `attributes` property; `document.createAttribute()`, `setAttribute()` methods.

`getAttributeNode("`*`attributeName`*`")`

Returns: Attribute node object.
Compatibility: WinIE6+, MacIE-, NN6+, Moz1+, Safari1+

In the W3C DOM, an attribute is an object that inherits all the properties of a `Node` object (see Chapter 14). As its name implies, an attribute object represents a name–value pair of an attribute that is explicitly defined inside an element's tag. The ability to treat attributes as node objects is far more important when working with XML than HTML, but it is helpful to understand attribute nodes within the context of the W3C DOM object-oriented view of a document. Importantly, attribute nodes specifically are not recognized as nodes of a document hierarchy. Therefore, an attribute node is not a child node of the element that defines the attribute.

The "nodeness" of attributes comes into play when addressing the contents of an object's `attributes` property. The W3C `attributes` property builds on the DOM's formal structure by returning an object known (internally) as a *named node map*. Like an array, the named node map has a `length` property (facilitating `for` loop interation through the map), plus several methods that allow for inserting, removing, reading, or writing attribute name–value pairs within the node map.

***elementObject*.getAttributeNode()**

An attribute object inherits all the properties of the `Node` object. Table 15-9 lists the properties of an attribute object.

Table 15-9: Attribute Object Properties of W3C DOM–Compatible Browsers

`attributes`
`childNodes`
`firstChild`
`lastChild`
`name`
`nextSibling`
`nodeName`
`nodeType`
`nodeValue`
`ownerDocument`
`parentNode`
`previousSibling`
`specified`
`Value`

All of this is a long way to explain the W3C DOM `getAttributeNode()` method, which returns a W3C DOM attribute object. The sole parameter of the method is a case-insensitive string version of the attribute's name. You can then use any of the properties shown in Table 15-9 to get or set attribute values. Of course, HTML attributes are generally exposed as properties of HTML elements, so it is usually easier to read or write the object's properties directly.

Example

Use The Evaluator (Chapter 13) to explore the `getAttributeNode()` method in NN6. The Results `textarea` element provides several attributes to check out. Because the method returns an object, enter the following statements into the bottom text field so you can view the properties of the attribute node object returned by the method:

```
document.getElementById("output").getAttributeNode("cols")
document.getElementById("output").getAttributeNode("rows")
document.getElementById("output").getAttributeNode("wrap")
document.getElementById("output").getAttributeNode("style")
```

All (except the last) statements display a list of properties for each attribute node object. The last statement, however, returns nothing because the `style` attribute is not specified for the element.

Related Items: `attributes` property; `getAttribute()`, `removeAttributeNode()`, `setAttributeNode()` methods.

`getAttributeNodeNS("`*`namespaceURI`*`", "`*`localName`*`")`

Returns: Attribute node object.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

This method returns a W3C DOM attribute object. The first parameter of the method is a URI string matching a URI assigned to a label in the document. The second parameter is the local name portion of the attribute you are getting.

Related Items: `attributes`, `namespaceURI`, `localName` properties; `getAttributeNode()`, `setAttributeNodeNS()` methods.

`getAttributeNS("namespaceURI", "localName")`

Returns: See text.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

This method returns the value assigned to a specific attribute of the current object when that attribute's name is defined by way of an XML namespace definition within the document. The first parameter of the method is a URI string matching a URI assigned to a namespace label in a tag defined earlier in the document. The second parameter is the local name portion of the attribute whose value you are getting.

Returned value types from `getAttributeNS()` are either strings (including attribute values assigned as unquoted numeric values) or Booleans (for example, the `multiple` property of a `select` element object).

Related Items: `attributes`, `namespaceURI`, `localName` properties; `getAttribute()`, `getAttributeNodeNS()`, `setAttributeNodeNS()` methods.

`getBoundingClientRect()`

Returns: `TextRectangle` object.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

IE5+ assigns to every content-holding element a rectangle that describes the space that the element occupies on the page. This rectangle is called a *bounding rectangle*, and it is expressed in the WinIE5+ object model as a `TextRectangle` object (even when the content is an image or some other kind of object). A `TextRectangle` object has four properties (`top`, `left`, `bottom`, and `right`) that are the pixel coordinates that define the rectangle. The `getBoundingClientRect()` method returns a `TextRectangle` object that describes the bounding rectangle of the current object. You can access an individual measure of an object's bounding rectangle, as in the following example:

```
var parTop = document.getElementById("myP").getBoundingClientRect().top;
```

For elements that consist of text, such as paragraphs, the dimensions of individual `TextRectangle`s for each line of text in the element influence the dimensions of the bounding rectangle. For example, if a paragraph contains two lines, and the second line extends only halfway across the width of the first line, the width of the second line's `TextRectangle` object is only as wide as the actual text in the second line. But because the first line extends close to the right margin, the width of the encompassing bounding rectangle is governed by that wider, first line `TextRectangle`. Therefore, an element's bounding rectangle is as wide as its widest line and as tall as the sum of the height of all `TextRectangle` objects in the paragraph.

Another method, `getClientRects()`, enables you to obtain a collection of line-by-line `TextRectangle` objects for an element.

Example

Listing 15-27 employs both the `getBoundingClientRect()` and `getClientRects()` methods in a demonstration of how they differ. A set of elements are grouped within a `span` element named `main`. The group consists of two paragraphs and an unordered list.

Two controls enable you to set the position of an underlying highlight rectangle to any line of your choice. A checkbox enables you to set whether the highlight rectangle should be only as wide as the line or the full width of the bounding rectangle for the entire `span` element.

All the code is located in the `hilite()` function. The `select` and checkbox elements invoke this function. Early in the function, the `getClientRects()` method is invoked for the `main` element to capture a snapshot of all `TextRectangle`s for the entire element. This array comes in handy when the script needs to get the coordinates of a rectangle for a single line, as chosen in the `select` element.

Whenever the user chooses a number from the `select` list and the value is less than the total number of `TextRectangle` objects in `clientRects`, the function begins calculating the size and location of the underlying yellow highlighter. When the Full Width checkbox is checked, the left and right coordinates are obtained from the `getBoundingClientRect()` method because the entire `span` element's rectangle is the space you're interested in; otherwise, you pull the `left` and `right` properties from the chosen rectangle in the `clientRects` array.

Next comes the assignment of location and dimension values to the `hiliter` object's `style` property. The top and bottom are always pegged to whatever line is selected, so the `clientRects` array is polled for the chosen entry's `top` and `bottom` properties. The previously calculated `left` value is assigned to the `hiliter` object's `pixelLeft` property, while the width is calculated by subtracting the `left` from the `right` coordinates. Notice that the `top` and `left` coordinates also take into account any vertical or horizontal scrolling of the entire body of the document. If you resize the window to a smaller size, line wrapping throws off the original line count. However, an invocation of `hilite()` from the `onresize` event handler applies the currently chosen line number to whatever content falls in that line after resizing.

Listing 15-27: **Using getBoundingClientRect()**

```
<html>
   <head>
      <title>getClientRects() and getBoundClientRect() Methods</title>
      <script type="text/javascript">
      function hilite() {
         var hTop, hLeft, hRight, hBottom, hWidth;
         var select = document.forms[0].choice;
         var n = parseInt(select.options[select.selectedIndex].value) - 1;
         var clientRects = document.getElementById("main").getClientRects();
         var mainElem = document.getElementById("main");
         if (n >= 0 && n < clientRects.length) {
            if (document.forms[0].fullWidth.checked) {
               hLeft = mainElem.getBoundingClientRect().left;
               hRight = mainElem.getBoundingClientRect().right;
            } else {
               hLeft = clientRects[n].left;
               hRight = clientRects[n].right;
            }
            document.getElementById("hiliter").style.pixelTop =
               clientRects[n].top + document.body.scrollTop;
            document.getElementById("hiliter").style.pixelBottom =
               clientRects[n].bottom;
            document.getElementById("hiliter").style.pixelLeft =
               hLeft + document.body.scrollLeft;
            document.getElementById("hiliter").style.pixelWidth =
               hRight - hLeft;
            document.getElementById("hiliter").style.visibility = "visible";
         } else if (n > 0) {
```

```
            alert("The content does not have that many lines.");
            document.getElementById("hiliter").style.visibility = "hidden";
         }
      }
      </script>
   </head>
   <body onresize="hilite()">
      <h1>getClientRects() and getBoundClientRect() Methods</h1>
      <hr />
      <form>
         Choose a line to highlight: <select name="choice" onchange="hilite()">
            <option value="1">1</option>
            <option value="2">2</option>
            <option value="3">3</option>
            <option value="4">4</option>
            <option value="5">5</option>
            <option value="6">6</option>
            <option value="7">7</option>
            <option value="8">8</option>
            <option value="9">9</option>
            <option value="10">10</option>
            <option value="11">11</option>
            <option value="12">12</option>
            <option value="13">13</option>
            <option value="14">14</option>
            <option value="15">15</option>
         </select><br />
          <input name="fullWidth" type="checkbox" onclick="hilite()" /> Full
         Width (bounding rectangle)
      </form>
      <span id="main">
      <p>Lorem ipsum dolor sit amet, consectetaur adipisicing
         elit, sed do eiusmod tempor incididunt ut labore et dolore magna
         aliqua. Ut enim adminim veniam, quis nostrud exercitation
         ullamco:</p>
      <ul>
         <li>laboris</li>
         <li>nisi</li>
         <li>aliquip ex ea commodo</li>
      </ul>
      <p>Duis aute irure dolor in reprehenderit involuptate velit esse cillum
         dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
         proident, sunt in culpa qui officia deseruntmollit anim id est laborum
         Et harumd und lookum like Greek to me, dereud facilis est er expedit
         distinct.</p></span>
      <div id="hiliter"
      style="position:absolute; background-color:yellow; z-index:-1;
      visibility:hidden">
      </div>
   </body>
</html>
```

Because the `z-index` style property of the `hiliter` element is set to `-1`, the element always appears beneath the primary content on the page. If the user selects a line number beyond the current number of lines in the main element, the `hiliter` element is hidden.

Related Items: `getClientRects()` method; `TextRectangle` object (Chapter 35).

***elementObject*.getClientRects()**

getClientRects()

Returns: Array of `TextRectangle` objects.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `getClientRects()` method returns an array of all `TextRectangle` objects that fall within the current object the moment the method is invoked. Each `TextRectangle` object has its own `top`, `left`, `bottom`, and `right` coordinate properties. You can then, for example, loop through all objects in this array to calculate the pixel width of each line. If you want to find out the aggregate height and/or maximum width of the entire collection, you can use the `getBoundingClientRect()` method as a shortcut.

Example

See Listing 15-27, which demonstrates the differences between `getClientRects()` and `getBoundingClientRect()` and shows how you can use the two together.

Related Items: `getBoundingClientRect()` method; `TextRectangle` object (Chapter 35).

getElementsByTagName("*tagName*")

Returns: Array of element objects.
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The `getElementsByTagName()` method returns an array of all elements contained by the current object whose tags match the tag name supplied as the sole parameter to the method. The tag name parameter must be in the form of a string and is case-insensitive. The group of elements returned in the array includes only those elements that are within the containment scope of the current object. Therefore, if you have two table objects in a document and you invoke the `getElementsByTagName("td")` method on one of them, the list of returned table cell elements is confined to those cells within the current table object. The current element is not included in the returned array.

The method accepts a wildcard character (`"*"`) as a parameter to the `getElementsByTagName()` method. The resulting array of elements is nearly identical to what IE4+ returns via the `document.all` collection.

Example

Use The Evaluator (Chapter 13) to experiment with the `getElementsByTagName()` method. Enter the following statements one at a time into the upper text box and study the results:

```
document.body.getElementsByTagName("div")
document.body.getElementsByTagName("div").length
document.getElementById("myTable").getElementsByTagName("td").length
```

Because the `getElementsByTagName()` method returns an array of objects, you can use one of those returned values as a valid element reference:

```
document.getElementsByTagName("form")[0].getElementsByTagName("input").length
```

Related Items: `getElementByTagNameNS()`, `getElementById()`, `tags()` methods.

getElementsByTagNameNS("*namespaceURI*", "*localName*")

Returns: Array of element objects.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

This method returns an array of all elements contained by the current object (within an XML document) as specified in the two parameters. The first parameter of the method is a URI string matching a URI assigned to a label in the document. The second parameter is the local name portion of the attribute whose value you are getting.

Returned value types from `getAttributeNS()` are either strings (including attribute values assigned as unquoted numeric values) or Booleans (for example, the `multiple` property of a `select` element object).

Related Items: `attributes`, `namespaceURI`, `localName` properties; `getElementsByTagNameNS()`, `getElementById()`, `tags()` methods.

`getExpression("attributeName")`

Returns: String.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `getExpression()` method returns the text of the expression that was assigned to an element's attribute via the `setExpression()` method. The returned value is not the value of the expression, but rather the expression itself. If you want to find out the current value of the expression (assuming that the variables used are within the scope of your script), you can use the `eval()` function on the call to `getExpression()`. This action converts the string to a JavaScript expression and returns the evaluated result.

One parameter, a string version of the attribute name, is required.

Example

See Listing 15-32 for the `setExpression()` method. This listing demonstrates the kinds of values returned by `getExpression()`.

Related Items: `document.recalc()`, `removeExpression()`, `setExpression()` methods.

`hasAttribute("attributeName")`

Returns: Boolean.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `hasAttribute()` method returns `true` if the current object has an attribute whose name matches the sole parameter; it returns `false` otherwise.

Related Items: `hasAttributeNS()` and `hasAttributes()` methods.

`hasAttributeNS("namespaceURI", "localName")`

Returns: Boolean.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `hasAttributeNS()` method returns `true` if the current object has an attribute as identified by the two parameters; it returns `false` otherwise. The first parameter of the method is a URI string matching a URI assigned to a label in the document. The second parameter is the local name portion of the attribute whose value you are getting.

Related Items: `attributes`, `namespaceURI`, `localName` properties; `hasAttribute()` and `hasAttributes()` methods.

`hasAttributes()`

Returns: Boolean.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

***elementObject*.hasAttributes()**

The `hasAttributes()` method returns `true` if the current object has any attributes explicitly assigned within the tag; it returns `false` otherwise.

Related Items: `hasAttribute()` and `hasAttributeNS()` methods.

`hasChildNodes()`

Returns: Boolean.
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The `hasChildNodes()` method returns `true` if the current object has child nodes nested within; it returns `false` otherwise. A child node is not necessarily the same as a child element, so the following two expressions return `true` when the current object has at least one child node:

```
document.getElementById("myObject").hasChildNodes()
document.getElementById("myObject").childNodes.length > 0
```

You cannot use the second expression interchangeably with the following statement (which uses the IE-only `children` property):

```
document.getElementById("myObject").children.length > 0
```

You generally use the `hasChildNodes()` method in a conditional expression to make sure such nodes exist before performing operations on them:

```
if (document.getElementById("myObject").hasChildNodes() {
    statements that apply to child nodes
}
```

Example

Use The Evaluator (Chapter 13) to experiment with the `hasChildNodes()` method. If you enter the following statement into the topmost text box:

```
document.getElementById("myP").hasChildNodes()
```

the returned value is `true`. You can find out how many nodes there are by getting the `length` of the `childNodes` array:

```
document.getElementById("myP").childNodes.length
```

This expression reveals a total of three nodes: the two text nodes and the `em` element between them. Check out whether the first text node has any children:

```
document.getElementById("myP").childNodes[0].hasChildNodes()
```

The response is `false` because text fragments do not have any nested nodes. But check out the `em` element, which is the second child node of the `myP` element:

```
document.getElementById("myP").childNodes[1].hasChildNodes()
```

The answer is `true` because the `em` element has a text fragment node nested within it. Sure enough, the statement

```
document.getElementById("myP").childNodes[1].childNodes.length
```

yields a node count of `1`. You can also go directly to the `em` element in your references:

```
document.getElementById("myEM").hasChildNodes()
document.getElementById("myEM").childNodes.length
```

If you want to see the properties of the text fragment node inside the em element, enter the following into the lower text box:

```
document.getElementById("myEM").childNodes[0]
```

You can see that the data and nodeValue properties for the text fragment return the text "all".

Related Items: childNodes property; appendChild(), removeChild(), replaceChild() methods.

insertAdjacentElement("*location*", *elementObject*)

Returns: Object.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The insertAdjacentElement() method inserts an element object (coming from a variety of sources) in a specific position relative to the current object. Both parameters are required. The first must be one of four possible case-insensitive locations for the insertion, shown in the following table:

Location	*Description*
beforeBegin	Before the current element's start tag
afterBegin	After the start tag, but before any nested content
beforeEnd	Before the end tag, but after all other nested content
afterEnd	After the end tag

These locations are relative to the current object. The element type of the current object (a block-level or inline element) has great bearing on how the inserted element is rendered. For example, suppose you create a b element (using document.createElement()) and assign some inner text to it. You then use insertAdjacentElement() in an effort to insert this b element before some text in a p element. Because a p element is a block-level element, the location beforeBegin places the new b element before the start tag of the p element. This means, however, that the bold text appears in a text line above the start of the p element because a <p> tag begins a new block at the left margin of its container (unless instructed otherwise by stylesheets). The resulting HTML looks like the following:

```
<b>The new element.</b><p>The original paragraph element.</p>
```

To make the new b element a part of the p element — but in front of the existing p element's content — use the afterBegin location. The resulting HTML looks like the following:

```
<p><b>The new element.</b>The original paragraph element.</p>
```

To complete the demonstration of the four location types, the following is the result of the beforeEnd location:

```
<p>The original paragraph element. <b>The new element.</b></p>
```

and this is the result of the afterEnd location:

```
<p>The original paragraph element.</p><b>The new element.</b>
```

The object to be inserted is a reference to an element object. The object reference can come from any expression that evaluates to an element object or, more likely, from the result of the document.createElement() method. Bear in mind that the object generated

by `document.createElement()` initially has no content, and all attribute values are set to default values. Moreover, the object is passed to `insertAdjacentElement()` by reference, which means that there is only one instance of that object. If you attempt to insert that object in two places with two statements, the object is moved from the first location to the second. If you need to copy an existing object so that the original is not moved or otherwise disturbed by this method, use the `cloneNode()` method to specify the `true` parameter to capture all nested content of the node.

Example

Use The Evaluator (Chapter 13) to experiment with the `insertAdjacentElement()` method. The goal of the experiment is to insert a new `h1` element above the `myP` element.

All actions require you to enter a sequence of statements in the topmost text box. Begin by storing a new element in the global variable `a`:

```
a = document.createElement("h1")
```

Give the new object some text:

```
a.innerText = "New Header"
```

Now, insert this element before the start of the `myP` object:

```
myP.insertAdjacentElement("beforeBegin", a)
```

Notice that you have not assigned an `id` property value to the new element. But because the element was inserted by reference, you can modify the inserted object by changing the object stored in the `a` variable:

```
a.style.color = "red"
```

The inserted element is also part of the document hierarchy, so you can access it through hierarchy references such as `myP.previousSibling`.

The parent element of the newly inserted element is the `body`. Thus, you can inspect the current state of the HTML for the rendered page by entering the following statement into the topmost text box:

```
document.body.innerHTML
```

If you scroll down past the first form, you can find the `<h1>` element that you added along with the `style` attribute.

Related Items: `document.createElement()`, `applyElement()` methods.

`insertAdjacentHTML("location", "HTMLtext")`
`insertAdjacentText("location", "text")`

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

These two methods insert HTML or straight text at a location relative to the current element. They are intended for use after a page loads, rather than inserting content while the page loads (in which case you can use `document.write()` wherever you need evaluated content to appear on the page).

The first parameter must be one of four possible case-insensitive locations for the insertion, shown in the following table:

Location	*Description*
beforeBegin	Before the current element's start tag
afterBegin	After the start tag, but before any nested content
beforeEnd	Before the end tag, but after all other nested content
afterEnd	After the end tag

These locations yield the same results as described in the insertAdjacentElement() function discussed earlier.

Whether you use insertAdjacentHTML() or insertAdjacentText() depends on the nature of your content and what you want the browser to do with it. If the content contains HTML tags that you want the browser to interpret and render as if it were part of the page source code, use the insertAdjacentHTML() method. All tags become objects in the document's object model. But if you want only to display some text (including HTML tags in their "raw" form), use insertAdjacentText(). The rendering engine does not interpret any tags included in the string passed as the second parameter. Instead, these tags are displayed as characters on the page. This distinction is identical to the one between the innerHTML and innerText properties.

The difference between insertAdjacentHTML() and insertAdjacentElement() is the nature of the content that you insert. The former enables you to accumulate the HTML as a string, while the latter requires the creation of an element object. Also, the two methods in this section work with IE4+ (including Mac versions), whereas insertAdjacentElement() requires the newer object model of WinIE5+.

If the HTML you pass as the second parameter of insertAdjacentHTML() contains <script> tags, you must set the defer attribute in the opening tag. This prevents script statements from executing as you insert them.

Example

Use The Evaluator (Chapter 13) to experiment with these two methods. The example here demonstrates the result of employing both methods in an attempt to add some HTML to the beginning of the myP element.

Begin by assigning a string of HTML code to the global variable a:

```
a = "<b id='myB'>Important News!</b>"
```

Because this HTML is to go on the same line as the start of the myP paragraph, use the afterBegin parameter for the insert method:

```
myP.insertAdjacentHTML("afterBegin", a)
```

Notice that there is no space after the exclamation mark of the inserted HTML. But to prove that the inserted HTML is genuinely part of the document's object model, you can now insert the text of a space after the b element whose ID is myB:

```
myB.insertAdjacentText("afterEnd", " ")
```

Each time you evaluate the preceding statement (by repeatedly clicking the Evaluate button or pressing Enter with the cursor in the topmost field), an additional space is added.

You should also see what happens when the string to be inserted with `insertAdjacentText()` contains HTML tags. Reload The Evaluator and enter the following two statements into the top-most field, evaluating each one in turn:

```
a = "<b id='myB'>Important News!</b>"
myP.insertAdjacentText("afterBegin", a)
```

The HTML is not interpreted but is displayed as plain text. There is no object named `myB` after executing this latest insert method.

Related Items: `innerText`, `innerHTML`, `outerText`, `outerHTML` properties; `insertAdjacentElement()`, `replaceAdjacentText()` methods.

`insertBefore(`*`newChildNodeObject, referenceChildNode`*`)`

Returns: `Node` object.
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The `insertBefore()` method is the W3C DOM syntax for inserting a new child node into an existing element. Node references for both parameters must be valid `Node` objects (including those that `document.createElement()` generates).

The behavior of this method might seem counterintuitive at times. If you include the second parameter (a reference to an existing child node of the current element — optional in IE), the new child node is inserted before that existing one. But if you omit the second parameter (or its value is `null`), the new child node is inserted as the last child of the current element — in which case, the method acts the same as the `appendChild()` method. The true power of this method is summoned when you specify that second parameter; from the point of view of a parent element, you can drop a new child into any spot among its existing children. If an inserted node already exists in the document tree, it will be removed from its previous position.

Bear in mind that the `insertBefore()` method works from a parent element. Internet Explorer provides additional methods, such as `insertAdjacentElement()`, to operate from the perspective of what will become a child element.

Example

Listing 15-28 demonstrates how the `insertBefore()` method can insert child elements (`li`) inside a parent (`ol`) at different locations, depending on the second parameter. A text box enables you to enter your choice of text and/or HTML for insertion at various locations within the `ol` element. If you don't specify a position, the second parameter of `insertBefore()` is passed as `null` — meaning that the new child node is added to the end of the existing children. But choose a spot from the select list where you want to insert the new item. The value of each `select` list option is an index of one of the first three child nodes of the `ol` element.

Listing 15-28: Using the insertBefore() Method

```
<html>
   <head>
      <title>insertBefore() Method</title>
      <script type="text/javascript">
      function doInsert(form) {
         if (form.newText) {
            var newChild = document.createElement("LI");
            newChild.innerHTML = form.newText.value;
            var choice =
               form.itemIndex.options[form.itemIndex.selectedIndex].value;
```

```
            var insertPoint = (isNaN(choice)) ?
               null : document.getElementById("myUL").childNodes[choice];
            document.getElementById("myUL").insertBefore(newChild,
               insertPoint);
         }
      }
      </script>
   </head>
   <body>
      <h1>insertBefore() Method</h1>
      <hr />
      <form onsubmit="return false">
         <p>Enter text or HTML for a new list item: <input type="text"
            name="newText" size="40" value="" /></p>
         <p>Before which existing item? <select name="itemIndex">
               <option value="null">None specified</option>
               <option value="0">1</option>
               <option value="1">2</option>
               <option value="2">3</option>
            </select></p>
         <input type="button" value="Insert Item"
         onclick="doInsert(this.form)" />
      </form>
      <ol id="myUL">
         <li>Originally the First Item</li>
         <li>Originally the Second Item</li>
         <li>Originally the Third Item</li>
      </ol>
   </body>
</html>
```

Related Items: `appendChild()`, `replaceChild()`, `removeChild()`, `insertAdjacentElement()` methods.

`isSupported("feature", "version")`

Returns: Boolean.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `isSupported()` method returns `true` if the current node supports required portions of the specified W3C DOM module and version; it returns `false` otherwise. The first parameter accepts any of the following case-sensitive DOM module name strings: `Core`, `XML`, `HTML`, `Views`, `StyleSheets`, `CSS`, `CSS2`, `Events`, `UIEvents`, `MouseEvents`, `MutationEvents`, `HTMLEvents`, `Range`, `Traversal`. The second parameter accepts a string representation of the major and minor DOM module version, such as `"2.0"` for DOM Level 2.

Example

Use The Evaluator (Chapter 13) to experiment with the `isSupported()` method. If you have multiple versions of NN6 or later and Mozilla, try the following (and others) to see how the support for various modules has evolved:

```
document.body.isSupported("CSS", "2.0")
document.body.isSupported("CSS2", "2.0")
document.body.isSupported("Traversal", "2.0")
```

If you have access to Safari, try the same methods there to see the differences in modules supported compared to Mozilla-based browsers.

***elementObjectCollection*.item()**

item(*index* | "*index*" [, *subIndex*])

Returns: Object.
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `item()` method works with most objects that are themselves collections of other objects. In W3C DOM terminology, these kinds of objects are known as *named node lists* (for objects such as nodes and attributes) or *HTML collections* (for objects such as elements of a form). You may call the `item()` method with a single numeric parameter that is the index value of the desired object within the collection. If you know the index number of the item, you can use JavaScript array syntax instead. The following two statements return the same object reference:

```
document.getElementById("myTable").childNodes.item(2)
document.getElementById("myTable").childNodes[2]
```

The method also supports a string of the ID of an object within the collection. (Integer values are required for the `attributes`, `rules`, and `TextRectangle` objects, however.) Additionally, if the collection has more than one object with the same ID (never a good idea except when necessary), a second numeric parameter enables you to select which identically named group you want (using zero-based index values within that subgroup). This obviously does not apply to collections, such as attributes and rules, which have no ID associated with them.

The method returns a reference to the object specified by the parameters.

Example

Use The Evaluator (Chapter 13) to experiment with the `item()` method. Type the following statements into the topmost text box and view the results for each.

W3C and IE5:

```
document.getElementById("myP").childNodes.length
document.getElementById("myP").childNodes.item(0).data
document.getElementById("myP").childNodes.item(1).nodeName
```

W3C, IE4, and IE5:

```
document.forms[1].elements.item(0).type
```

IE4 and IE5:

```
document.all.item("myP").outerHTML
myP.outerHTML
```

In the last two examples, both statements return the same string. The first example is helpful when your script is working with a string version of an object's name. If your script already knows the object reference, the second approach is more efficient and compact.

Related Items: All object element properties that return collections (arrays) of other objects.

mergeAttributes("*sourceObject*")

Returns: Nothing.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `mergeAttributes()` method is a convenient way to propagate attributes in newly created elements without painstakingly adding attributes one at a time. Once you have an object whose attributes can function as a prototype for other elements, those attributes (except for the `id` attribute) can be applied to a newly created element instantaneously.

Example

Listing 15-29 demonstrates the usage of `mergeAttributes()` in the process of replicating the same form input field while assigning a unique ID to each new field. So you can see the results as you go, I display the HTML for each input field in the field.

The `doMerge()` function begins by generating two new elements: a `p` element and an `input` element. Because these newly created elements have no properties associated with them, a unique ID is assigned to the `input` element via the `uniqueID` property. Attributes from the field in the source code (`field1`) are merged into the new `input` element. Thus, all attributes except `name` and `id` are copied to the new element. The `input` element is inserted into the `p` element, and the `p` element is appended to the document's form element. Finally, the `outerHTML` of the new element is displayed in its field. Notice that except for the `name` and `id` attributes, all others are copied. This includes stylesheet attributes and event handlers. To prove that the event handler works in the new elements, you can add a space to any one of them and press Tab to trigger the `onchange` event handler that changes the content to all uppercase characters.

Listing 15-29: **Using the mergeAttributes() Method**

```
<html>
   <head>
      <title>mergeAttributes() Method</title>
      <script type="text/javascript">
      function doMerge(form) {
         var newPElem = document.createElement("p");
         var newInputElem = document.createElement("input");
         newInputElem.id = newInputElem.uniqueID;
         newInputElem.mergeAttributes(form.field1);
         newPElem.appendChild(newInputElem);
         form.appendChild(newPElem);
         newInputElem.value = newInputElem.outerHTML;
      }

      // called by onChange event handler of fields
      function upperMe(field) {
         field.value = field.value.toUpperCase();
      }
      </script>
   </head>
   <body
   onload="document.expandable.field1.value =
document.expandable.field1.outerHTML">
      <h1>mergeAttributes() Method</h1>
      <hr />
      <form name="expandable" onsubmit="return false">
         <p><input type="button" value="Append Field 'Clone'"
            onclick="doMerge(this.form)" /></p>
         <p><input type="text" name="field1" id="FIELD1" size="120" value=""
            style="font-size:9pt" onchange="upperMe(this)" /></p>
      </form>
   </body>
</html>
```

Related Items: `clearAttributes()`, `cloneNode()`, `removeAttributes()` methods.

normalize()

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

In the course of appending, inserting, removing, and replacing child nodes of an element, it is conceivable that two text nodes can end up adjacent to each other. While this typically has no effect on the rendering of the content, some XML-centric applications that rely heavily on the document node hierarchy to interpret content properly may not like having two text nodes sitting next to each other. The "proper" form of a node hierarchy is for a single text node to be bounded by other node types. The `normalize()` method sweeps through the child nodes of the current node object and combines adjacent text nodes into a single text node. The effect obviously impacts the number of child nodes of an element, but it also cleanses the nested node hierarchy.

Example

Use The Evaluator (Chapter 13) to experiment with the `normalize()` method in Mozilla-based browsers. The following sequence adds a text node adjacent to one in the `myP` element. A subsequent invocation of the `normalize()` method removes the division between the adjacent text nodes.

Begin by confirming the number of child nodes of the `myP` element:

```
document.getElementById("myP").childNodes.length
```

Three nodes initially inhabit the element. Next, create a text node and append it as the last child of the `myP` element:

```
a = document.createTextNode("This means you!")
document.getElementById("myP").appendChild(a)
```

With the new text now rendered on the page, the number of child nodes increases to four:

```
document.getElementById("myP").childNodes.length
```

You can see that the last child node of `myP` is the text node you just created:

```
document.getElementById("myP").lastChild.nodeValue
```

But by invoking `normalize()` on `myP`, all adjacent text nodes are accumulated into single nodes:

```
document.getElementById("myP").normalize()
```

You can now see that the `myP` element is back to three child nodes, and the last child is a combination of the two previously distinct, but adjacent, text nodes:

```
document.getElementById("myP").childNodes.length
document.getElementById("myP").lastChild.nodeValue
```

Related Items: `document.createTextNode()`, `appendChild()`, `insertBefore()`, `removeChild()`, `replaceChild()` methods.

releaseCapture()
setCapture(*containerBoolean*)

Returns: Nothing.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

You can instruct a single object on a page to capture all mouse events (`onmousedown`, `onmouseup`, `onmousemove`, `onmouseout`, `onmouseover`, `onclick`, and `ondblclick`) via the

WinIE-specific `setCapture()` method. A primary scenario for mouse event capture is when some content appears on the page that you wish to leave as the center of user focus — items such as pull-down menus, context menus, or simulated modal window areas. When such items appear on the screen, you want the effect of blocking all mouse events except those that apply to the menu or currently visible pseudowindow. When the region disappears, mouse events can be released so that individual elements (such as buttons and links elsewhere on the page) respond to mouse events.

Event capture does not block the events. Instead, the events are redirected to the object set to capture all mouse events. Events bubble up from that point unless explicitly cancelled (see Chapter 25). For example, consider a document that has a `<body>` tag containing an `onclick` event handler that governs the entire document at all times. If you turn on event capture for a `div` somewhere in the document, the click event first goes to the `div`. That `div` might have an `onclick` event handler that looks to process click events when they occur in some of its child elements. If the event handler for the `div` does not also cancel the bubbling of that click event, the `body` element's `onclick` event handler eventually receives and processes the event, even though the `div` initially captured the event.

Deciding which object should capture events is an important design issue to confront. With event capture engaged, all mouse events (no matter where they occur) get funneled to the object set to capture the events. Therefore, if you design an application whose entire interface consists of clicking and dragging positionable elements, you can set one of those elements (or even the `document` object) to perform the capturing. For pop-up regions, however, it is generally more logical and convenient for your coding to assign the capture mechanism to the primary container of the pop-up content (usually a positioned `div`).

The `setCapture()` method has one optional Boolean parameter. The parameter controls whether mouse events on child elements within the capturing object are under control of the event capture mechanism. The default value (`true`) means that all mouse events targeted at elements within the current object go to the current object rather than to the original target — the most likely way you will use `setCapture()` for things such as pop-up and context menus. But if you specify `false` as the parameter, then mouse events occurring in child elements of the capturing container receive their events directly. From there, regular event bubbling upward from the target ensues (see Chapter 25).

You may encounter odd behavior when the region you set up to capture mouse events contains form elements such as text input fields and `select` lists. Because these elements require mouse events to gain focus for interaction, the event capture mechanism inhibits access to these items. To work around this behavior, you can examine the click event's `srcElement` property to see if the click was on one of these elements and script the focus of that element (or instruct the user to press the Tab key until the element gets focus manually).

Once an object is set to capture events, your other code must define which events actually do something; and decide whether events should bubble up beyond the capturing element. You need to worry about bubbling only if your design includes mouse event handlers in elements higher up the element containment hierarchy. You may not wish for those event handlers to fire while event capture is on; in this case, you need to cancel the bubbling of those events in the capturing object.

If your application design requires that the pop-up area be hidden and event handling be returned to normal (such as after the user makes a pop-up menu selection), use the `releaseCapture()` method in conjunction with hiding the container. Because event capture can be engaged for only one element at a time, you can release capture by invoking the `releaseCapture()` method from the container or from the `document` object.

Event capture is automatically disengaged when the user performs any of the following actions:

- ✦ Gives focus to any other window
- ✦ Displays any system modal dialog window (for example, alert window)
- ✦ Scrolls the page
- ✦ Opens a browser context menu (by right-clicking)
- ✦ Tabs to give focus to the Address field in the browser window

Therefore, you may want to set the `document` object's `onlosecapture` event handler to hide any container that your script displays in concert with event capture.

Also be aware that even though mouse events may be captured to prevent mouse access to the rest of the page, keyboard events are not captured. Thus, using the event capture mechanism to simulate modal windows is not foolproof: a user can tab to any form element or link in the page and press the spacebar or Enter key to activate that element.

Event capture, as defined in the W3C DOM, operates differently from WinIE event capture. In the W3C DOM, you can instruct the browser to substitute event capture of any kind of event for the normal event bubbling behavior. For example, you can attach an event listener to the `body` element in such a way that it sees all click events aimed at elements contained by the `body` element before the events reach their target elements. (See Chapters 14 and 25 for more on the W3C DOM event model and how to integrate it into cross-browser applications.)

Example

Listing 15-30 demonstrates the usage of `setCapture()` and `releaseCapture()` in a "quick-and-dirty" context menu for WinIE5+. The job of the context menu is to present a list of numbering styles for the ordered list of items on the page. Whenever the user brings up the context menu atop the `ol` element, the custom context menu appears. Event capture is turned on in the process to prevent mouse actions elsewhere on the page from interrupting the context menu choice. Even a click on the link set up as the title of the list is inhibited while the context menu is visible. A click anywhere outside of the context menu hides the menu. Clicking a choice in the menu changes the `listStyleType` property of the `ol` object and hides the menu. Whenever the context menu is hidden, event capture is turned off so that clicking on the page (such as the link) works as normal.

For this design, `onclick`, `onmouseover`, and `onmouseout` event handlers are assigned to the `div` element that contains the context menu. To trigger the display of the context menu, the `ol` element has an `oncontextmenu` event handler. This handler invokes the `showContextMenu()` function. In this function, event capture is assigned to the context menu `div` object. The `div` is also positioned at the location of the click before it is set to be visible. To prevent the system's regular context menu from also appearing, the `event` object's `returnValue` property is set to `false`.

Now that all mouse events on the page go through the `contextMenu div` object, let's examine what happens with different kinds of events triggered by user action. As the user rolls the mouse, a flood of `mouseover` and `mouseout` events fire. The event handlers assigned to the `div` manage these events. But notice that the two event handlers, `highlight()` and `unhighlight()`, perform action only when the `srcElement` property of the event is one of the menu items in the `div`. Because the page has no other `onmouseover` or `onmouseout` event handlers defined for elements up the containment hierarchy, you do not have to cancel event bubbling for these events.

When a user clicks the mouse button, different things happen depending on whether event capture is enabled. Without event capture, the `click` event bubbles up from wherever it occurred to the `onclick` event handler in the `body` element. (An alert dialog box displays to let you know when the event reaches the `body`.) But with event capture turned on (the context menu is showing), the `handleClick()` event handler takes over to apply the desired choice whenever the click is atop one of the context menu items. For all `click` events handled by this function, the context menu is hidden and the `click` event is cancelled from bubbling up any higher (no alert dialog box appears). This takes place whether the user makes a choice in the context menu or clicks anywhere else on the page. In the latter case, all you need is for the context menu to go away like the real context menu does. For added insurance, the `onlosecapture` event handler hides the context menu when a user performs any of the actions just listed that cancel capture.

Listing 15-30: **Using setCapture() and releaseCapture()**

```
<html>
   <head>
      <title></title>
      <style type="text/css">
      #contextMenu {position:absolute; background-color:#cfcfcf;
         border-style:solid; border-width:1px;
         border-color:#EFEFEF #505050 #505050 #EFEFEF;
         padding:3px 10px; font-size:8pt; font-family:Arial, Helvetica;
         line-height:150%; visibility:hidden}
      .menuItem {color:black}
      .menuItemOn {color:white}
      ol {list-style-position:inside; font-weight:bold; cursor:nw-resize}
      li {font-weight:normal}
      </style>
      <script type="text/javascript">
      function showContextMenu() {
         contextMenu.setCapture();
         contextMenu.style.pixelTop = event.clientY + document.body.scrollTop;
         contextMenu.style.pixelLeft = event.clientX +
            document.body.scrollLeft;
         contextMenu.style.visibility = "visible";
         event.returnValue = false;
      }

      function revert() {
         document.releaseCapture();
         hideMenu();
      }

      function hideMenu() {
         contextMenu.style.visibility = "hidden";
      }

      function handleClick() {
         var elem = window.event.srcElement;
         if (elem.id.indexOf("menuItem") == 0) {
            document.getElementById("shapesList").style.listStyleType =
               elem.listtype;
         }
```

Continued

Listing 15-30 *(continued)*

```
            revert();
            event.cancelBubble = true;
         }

         function highlight() {
            var elem = event.srcElement;
            if (elem.className == "menuItem") {
               elem.className = "menuItemOn";
            }
         }

         function unhighlight() {
            var elem = event.srcElement
            if (elem.className == "menuItemOn") {
               elem.className = "menuItem";
            }
         }
         </script>
      </head>
      <body onclick="alert('You reached the document object.')">
         <ol id="shapesList" oncontextmenu="showContextMenu()">
            <li style="list-style: none"><a href=
               "javascript:alert('A%20sample%20link.')">Three-Dimensional
               Shapes</a></li>
            <li value="1">Circular Cylinder</li>
            <li>Cube</li>
            <li>Rectangular Prism</li>
            <li>Regular Right Pyramid</li>
            <li>Right Circular Cone</li>
            <li>Sphere</li>
         </ol>
         <div id="contextMenu" onlosecapture="hideMenu()" onclick="handleClick()"
         onmouseover="highlight()" onmouseout="unhighlight()">
            <span id="menuItem1" class="menuItem"
            listtype="upper-alpha">A,B,C,...</span><br />
            <span id="menuItem2" class="menuItem"
            listtype="lower-alpha">a,b,c,...</span><br />
            <span id="menuItem3" class="menuItem"
            listtype="upper-roman">I,II,III,...</span><br />
            <span id="menuItem4" class="menuItem"
            listtype="lower-roman">i,ii,iii,...</span><br />
            <span id="menuItem5" class="menuItem"
            listtype="decimal">1,2,3,...</span>
         </div>
      </body>
   </html>
```

Related Items: `addEventListener()`, `dispatchEvent()`, `fireEvent()`, `removeEventListener()` methods; `onlosecapture` event; `Event` object (Chapter 25).

`removeAttribute("`*attributeName*`"[, `*caseSensitivity*`])`

Returns: Boolean (IE); Nothing (NN/DOM).
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

If you create an attribute with the `setAttribute()` method, you can eliminate that attribute from the element object via the `removeAttribute()` method. The required parameter is the name of the attribute. IE4+ permits you to set and remove attributes such that the attribute names are case-sensitive. The default behavior of `removeAttribute()` in IE (the second parameter is a Boolean value) is `false`. Therefore, if you supply a value of `true` for the case-sensitivity parameter in `setAttribute()`, you should set the parameter to `true` in `removeAttribute()` to ensure a proper balance between created and removed attributes.

The W3C DOM (NN/Moz/Safari) version of the `removeAttribute()` method has a single parameter (a case-insensitive attribute name) and returns no value. The returned value in IE is `true` if the removal succeeds and `false` if it doesn't succeed (or the attribute is one that you set in some other manner).

Example

Use The Evaluator (Chapter 13) to experiment with the `removeAttribute()` method for the elements in the page. See the examples for the `setAttribute()` method later in this chapter, and enter the corresponding `removeAttribute()` statements in the top text box. Interlace statements using `getAttribute()` to verify the presence or absence of each attribute.

Related Items: `attributes` property; `document.createAttribute()`, `getAttribute()`, `setAttribute()` methods.

`removeAttributeNode(`*`attributeNode`*`)`
`setAttributeNode(`*`attributeNode`*`)`

Returns: `Attribute` object.
Compatibility: WinIE6+, MacIE-, NN6+, Moz1+, Safari1+

As discussed in the coverage of the `getAttributeNode()` method earlier in this chapter, the W3C DOM treats a name–value attribute pair as an attribute object. An attribute object is a distinct node within a named node map — a collection of attribute objects belonging to an element. Understanding named node maps and attribute objects is more useful in an XML environment where attributes can not only contain valuable data, but are not exposed to the document object model as properties you can access via script. Instead of accessing an object's properties, you work with the actual attributes.

If you want to insert an attribute in the formal W3C methodology, you can use `document.createAttribute()` to generate a new attribute object. Subsequent script statements assign values to the `nodeName` and `nodeValue` properties to give the attribute its traditional name–value pair. You can then insert that new attribute object into the attribute list of an object via the `setAttributeNode()` method. The sole parameter is an attribute object, and the return value is a reference to the newly inserted attribute object.

To remove an attribute node from an element using this syntax, employ the `removeAttributeNode()` method. Again, the sole parameter is an attribute object. If your script knows only the attribute's name, you can use `getAttributeNode()` to obtain a valid reference to the attribute object. The `removeAttributeNode()` method returns a reference to the removed attribute object. That object remains in the browser's memory, but it is not part of the document hierarchy. By capturing this removed attribute object in a variable, you have the flexibility to modify and assign it to another object elsewhere in the document.

In practice, you may rarely, if ever, need to address attributes as nodes. Other methods — notably `getAttribute()`, `removeAttribute()`, and `setAttribute()` — do the job when your scripts have only the name (as a string) of an attribute belonging to an element.

Example

Use The Evaluator (Chapter 13) to experiment with the `setAttributeNode()` and `removeAttributeNode()` methods for the `p` element in the page. The task is to create and add a `style` attribute to the `p` element. Begin by creating a new attribute and storing it temporarily in the global variable `a`:

```
a = document.createAttribute("style")
```

Assign a value to the `attribute` object:

```
a.nodeValue = "color:red"
```

Now insert the new attribute into the `p` element:

```
document.getElementById("myP").setAttributeNode(a)
```

The paragraph changes color in response to the newly added attribute.

Due to the NN6 bug that won't allow the method to return a reference to the newly inserted attribute node, you can artificially obtain such a reference:

```
b = document.getElementById("myP").getAttributeNode("style")
```

Finally, use the reference to the newly added attribute to remove it from the element:

```
document.getElementById("myP").removeAttribute(b)
```

Upon removing the attribute, the paragraph resumes its initial color. See the example for the `setAttribute()` method later in this chapter to discover how you can perform this same kind of operation with `setAttribute()`.

Related Items: `attributes` property; `document.createAttribute()`, `getAttribute()`, `getAttributeNode()`, `setAttribute()` methods.

removeAttributeNS("*namespaceURI*", "*localName*")

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

This method removes the attribute specified in the two parameters. The first parameter of the method is a URI string matching a URI assigned to a label in the document. The second parameter is the local name portion of the attribute whose value you are removing.

Related Items: `attributes`, `namespaceURI`, `localName` properties; `removeAttribute()`, `getAttributeNS()`, and `setAttributeNS()` methods.

removeBehavior(*ID*)

Returns: Boolean.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `removeBehavior()` method detaches a behavior from an object. It assumes that the behavior was added to the object via the `addBehavior()` method. The return value of the `addBehavior()` method is a unique identifier for that particular behavior. This identifier is the required parameter for the `removeBehavior()` method. Thus, you can add two behaviors to an object and remove just one of them if so desired. If the removal succeeds, the `removeBehavior()` method returns `true`; otherwise, it returns `false`.

Example

See Listings 15-19a and 15-19b earlier in this chapter for examples of how to use `addBehavior()` and `removeBehavior()`.

Related Item: `addBehavior()` method.

`removeChild(nodeObject)`

Returns: `Node` object reference.
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The `removeChild()` method erases a child element from the current element. Content associated with the child element is no longer visible on the page, and the object is no longer part of the document object hierarchy.

As destructive as that sounds, the specifications for the deleted object are not lost to the ether necessarily. The `removeChild()` method returns a reference to the removed node. By assigning this value to a variable, you can hold onto that object specification for insertion later in the session. You are free to use this value as a parameter to such methods as `appendChild()`, `replaceChild()`, `swapNode()`, and `insertBefore()`.

Remember that `removeChild()` is invoked from the point of view of a parent element. If you simply want to remove an element, you can do so more directly (in WinIE5+ only) with the `removeNode()` method.

Example

You can see an example of `removeChild()` as part of Listing 15-21 earlier in this chapter.

Related Items: `appendChild()`, `replaceChild()`, `removeNode()` methods.

`removeEventListener()`

(See `addEventListener()`)

`removeExpression("propertyName")`

Returns: Boolean.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

If you assign an expression to an object property (including an object's `style` object) via the `setExpression()` method, you can remove it under script control with the `removeExpression()` method. The sole parameter is the name of the property in string form. Property names are case-sensitive.

The method returns `true` if the removal succeeds; otherwise, `false` is returned. Be aware that removing an expression does not alter the value that is currently assigned to the property. In other words, you can use `setExpression()` to set a property's value and then remove the expression so that no further changes are made when the document recalculates expressions. If this is your goal, however, you are probably better served by simply setting the property directly via scripting.

Example

You can experiment with all three expression methods in The Evaluator (Chapter 13). The following sequence adds an expression to a stylesheet property of the `myP` element on the page and then removes it.

To begin, enter the number `24` in the bottom one-line text box in The Evaluator (but don't press Enter or click the List Properties button). This is the value used in the expression to govern the `fontSize` property of the `myP` object. Next, assign an expression to the `myP` object's `style` object by entering the following statement into the topmost text box:

```
myP.style.setExpression("fontSize","document.forms[0].inspector.value","JScript")
```

You can now enter different font sizes into the lower text box and have the values immediately applied to the `fontSize` property. (Keyboard events in the text box automatically trigger the recalculation.) The default unit is `px`, but you can also append other units (such as `pt`) to the value in the text field to see how different measurement units influence the same numeric value.

Before proceeding to the next step, enter a value other than 16 (the default `fontSize` value). Finally, enter the following statement in the topmost text box to disconnect the expression from the property:

```
myP.style.removeExpression("fontSize")
```

Notice that although you can no longer adjust the font size from the lower text box, the most recent value assigned to it still sticks to the element. To prove it, enter the following statement in the topmost text box to see the current value:

```
myP.style.fontSize
```

Related Items: `document.recalc()`, `getExpression()`, `setExpression()` methods.

removeNode(*removeChildrenFlag*)

Returns: `Node` object reference.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

You can use the `removeNode()` method to delete the current node from an element hierarchy in WinIE5+. The sole parameter is a Boolean value that directs the method to remove only itself (without its child nodes) or the node and all of its children (value of `true`). Exercise care with this method when you use a default parameter value of `false`: If the node has child nodes (for example, you attempt to remove a `table` but not its child nodes), IE5 can crash on you. However, you can safely remove the node and all of its children by passing `true` as the sole parameter.

The method returns a reference to the node object removed. This removed object is no longer accessible to the document object model. But the returned value contains all properties of the object as it existed before you removed it (including properties such as `outerHTML` and explicitly set stylesheet rules). Thus, you can use this value as a parameter to insert the node elsewhere in the document.

While the W3C DOM does not have a `removeNode()` method, the cross-browser method whose behavior most closely resembles `removeNode()` is the `removeChild()` method. The scope of the `removeChild()` method is one level up the object hierarchy from the object you use for the `removeNode()` method.

Example

Examine Listing 15-21 for the `appendChild()` method to understand the difference between `removeChild()` and `removeNode()`. In the `restore()` function, you can replace this statement

```
mainObj.removeChild(oneChild);
```

in IE5+ with

```
oneChild.removeNode(true);
```

The difference is subtle, but it is important to understand. See Listing 15-31 later in this chapter for another example of the `removeNode()` method.

Related Items: `Node` object; `appendChild()`, `cloneChild()`, `removeChild()`, `replaceChild()`, `replaceNode()` methods.

replaceAdjacentText("*location*", "*text*")

Returns: String.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `replaceAdjacentText()` method enables you to replace one chunk of document text with another in a specific position relative to the current object. Be aware that this method works only for plain text and not HTML tags. The returned value is the string of the text that you replace.

Both parameters are required. The first must be one of four possible case-insensitive locations for the insertion, shown in the following table:

Location	*Description*
`beforeBegin`	Before the current element's start tag
`afterBegin`	After the start tag, but before any nested content
`beforeEnd`	Before the end tag, but after all other nested content
`afterEnd`	After the end tag

This method is best used with inline (rather than block) elements when specifying the `beforeBegin` and `afterEnd` parameters. For example, if you attempt to use `replaceAdjacentText()` with `beforeBegin` on the second of two consecutive paragraph elements, the replacement text is inserted into the end of the first paragraph. You can think of the `replaceAdjacentText()` method in terms of text fragment nodes. The method replaces the text fragment node (given any one of the four position parameters) with new text. Replacing the text of a simple element with either the `afterBegin` or `beforeEnd` locations is the same as assigning that text to the object's `innerText` property.

Example

Use The Evaluator (Chapter 13) to experiment with the `replaceAdjacentText()` method. Enter each of the following statements into the top text box and watch the results in the `myP` element (and its nested `myEM` element) below the solid rule:

```
document.getElementById("myEM").replaceAdjacentText("afterBegin", "twenty")
```

Notice that the `myEM` element's new text picks up the behavior of the element. In the meantime, the replaced text (`all`) is returned by the method and displayed in the Results box:

```
document.getElementById("myEM").replaceAdjacentText("beforeBegin", "We need ")
```

All characters of the text fragment, including spaces, are replaced. Therefore, you may need to supply a trailing space, as shown here, if the fragment you replace has a space:

```
document.getElementById("myP").replaceAdjacentText("beforeEnd", " good people.")
```

This is another way to replace the text fragment following the myEM element, but it is also relative to the surrounding myP element. If you now attempt to replace text after the end of the myP block-level element,

```
document.getElementById("myP").replaceAdjacentText("afterEnd", "Hooray!")
```

the text fragment is inserted after the end of the myP element's tag set. The fragment is just kind of floating in the document object model as an unlabeled text node.

Related Items: innerText, outerText properties; getAdjacentText(), insertAdjacentHTML(), insertAdjacentText() methods.

replaceChild(*newNodeObject, oldNodeObject*)

Returns: Node object reference.
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The replaceChild() method enables you to swap an existing child node object for a new node object. Parameters for the replaceChild() method are node object references, and they must be in the order of the new object followed by the object you want to replace. The old object must be an immediate child node of the parent used to invoke the method, and the new object must also be a "legal" child element within the document containment hierarchy.

The method returns a reference to the child object that you replaced with the new object. This reference can be used as a parameter to any of the node-oriented insertion or replacement methods.

Remember that replaceChild() is invoked from the point of view of a parent element. If you simply want to change an element, you can do so more directly in WinIE5+ with the swapNode() or replaceNode() method.

Example

You can see an example of replaceChild() as part of Listing 15-21 (for the appendChild property) earlier in this chapter.

Related Items: appendChild(), removeChild(), replaceNode(), swapNode() methods.

replaceNode("*newNodeObject*")

Returns: Node object reference.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The replaceNode() method is related to the replaceChild() method, but you invoke this method on the actual node you want to replace (instead of the object's parent). The sole parameter is a reference to a valid node object, which you can generate via the document.createElement() method or copy from an existing node. The value returned from the method is a reference to the object that you replace. Thus, you can preserve a copy of the replaced node by storing the results in a variable for use later.

If the node you replace contains other nodes, the replaceNode() method removes all contained nodes of the original from the document. Therefore, if you want to change a wrapper node but want to maintain the original children, your script must capture the children and put them back into the new node as shown in the following example.

Example

Listing 15-31 demonstrates three node-related methods: removeNode(), replaceNode(), and swapNode(). These methods work in WinIE5+ only.

The page rendered from Listing 15-31 begins with a `ul` type list of four items. Four buttons control various aspects of the node structure of this list element. The first button invokes the `replace()` function, which changes the `ul` type to `ol`. To do this, the function must temporarily tuck away all child nodes of the original `ul` element so that they can be added back into the new `ol` element. At the same time, the old `ul` node is stored in a global variable (`oldNode`) for restoration in another function.

To replace the `ul` node with an `ol`, the `replace()` function creates a new, empty `ol` element and assigns the `myOL` ID to it. Next, the children (`li` elements) are stored en masse as an array in the variable innards. The child nodes are then inserted into the empty `ol` element, using the `insertBefore()` method. Notice that as each child element from the `innards` array is inserted into the `ol` element, the child element is removed from the `innards` array. That's why the loop to insert the child nodes is a `while` loop that constantly inserts the first item of the `innards` array to the new element. Finally, the `replaceNode()` method puts the new node in the old node's place, while the old node (just the `ul` element) is stored in `oldNode`.

The `restore()` function operates in the inverse direction of the `replace()` function. The same juggling of nested child nodes is required.

The third button invokes the `swap()` function, whose script exchanges the first and last nodes. The `swapNode()` method, like the others in this discussion, operates from the point of view of the node. Therefore, the method is attached to one of the swapped nodes, while the other node is specified as a parameter. Because of the nature of the `ol` element, the number sequence remains fixed but the text of the `li` node swaps.

To demonstrate the `removeNode()` method, the fourth function removes the last child node of the list. Each call to `removeNode()` passes the `true` parameter to guarantee that the text nodes nested inside each `li` node are also removed. Experiment with this method by setting the parameter to `false` (the default). Notice how the parent–child relationship changes when you remove the `li` node.

Listing 15-31: **Using Node-Related Methods**

```
<html>
   <head>
      <title>removeNode(), replaceNode(), and swapNode() Methods</title>
      <script type="text/javascript">
      // store original node between changes
      var oldNode;

      // replace UL node with OL
      function replace() {
         if (document.getElementById("myUL")) {
            var newNode = document.createElement("OL");
            newNode.id = "myOL";
            var innards = document.getElementById("myUL").children;
            while (innards.length > 0) {
               newNode.insertBefore(innards[0]);
            }
            oldNode = document.getElementById("myUL").replaceNode(newNode);
         }
      }

      // restore OL to UL
      function restore() {
```

Continued

Listing 15-31 *(continued)*

```
        if (document.getElementById("myOL") && oldNode) {
           var innards = document.getElementById("myOL").children;
           while (innards.length > 0) {
              oldNode.insertBefore(innards[0]);
           }
           document.getElementById("myOL").replaceNode(oldNode);
        }
     }

     // swap first and last nodes
     function swap() {
        if (document.getElementById("myUL")) {
           document.getElementById("myUL").firstChild.swapNode(
              document.getElementById("myUL").lastChild);
        }
        if (document.getElementById("myOL")) {
           document.getElementById("myOL").firstChild.swapNode(
              document.getElementById("myOL").lastChild);
        }
     }

     // remove last node
     function remove() {
        if (document.getElementById("myUL")) {
           document.getElementById("myUL").lastChild.removeNode(true);
        }
        if (document.getElementById("myOL")) {
           document.getElementById("myOL").lastChild.removeNode(true);
        }
     }
     </script>
  </head>
  <body>
     <h1>Node Methods</h1>
     <hr />
     Here is a list of items:
     <ul id="myUL">
        <li>First Item</li>
        <li>Second Item</li>
        <li>Third Item</li>
        <li>Fourth Item</li>
     </ul>
     <form>
        <input type="button" value="Change to OL List"
        onclick="replace()" />   <input type="button"
        value="Restore LI List" onclick="restore()" />   <input
        type="button" value="Swap First/Last" onclick="swap()" />  
        <input type="button" value="Remove Last" onclick="remove()" />
     </form>
  </body>
</html>
```

You can accomplish the same functionality shown in Listing 15-31 in a cross-browser fashion using the W3C DOM. In place of the `removeNode()` and `replaceNode()` methods, use `removeChild()` and `replaceChild()` methods to shift the point of view (and object references)

to the parent of the ul and ol objects: the document.body. Also, you need to change the document.all references to document.getElementById().

Related Items: removeChild(), removeNode(), replaceChild(), swapNode() methods.

scrollIntoView(*topAlignFlag*)

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN7+, Moz1+, Safari-

The scrollIntoView() method scrolls the page (vertically and/or horizontally as needed) such that the current object is visible within the window or frame that contains it. A single parameter, a Boolean value, controls the location of the element within the viewable space. A value of true (the default) causes the element to be displayed so that its top is aligned with the top of the window or frame (provided the document beneath it is long enough to allow this amount of scrolling). But a value of false causes the bottom of the element to align with the bottom of the viewable area. In most cases, you want the former so that the beginning of a page section is at the top of the viewable area. But if you don't want a user to see content below a certain element when you jump to the new view, use the false parameter.

For form elements, you must use the typical form element reference (document.*formName*.*elementName*.scrollIntoView()) unless you also specify an ID attribute for the element (document.getElementById("*elementID*").scrollIntoView()).

Example

Use The Evaluator (Chapter 13) to experiment with the scrollIntoView() method. Resize the browser window height so that you can see only the topmost text box and the Results text area. Enter each of the following statements into the top text box and see where the myP element comes into view:

```
myP.scrollIntoView()
myP.scrollIntoView(false)
```

Expand the height of the browser window until you can see part of the table lower on the page. If you enter

```
myTable.scrollIntoView(false)
```

into the top text box, the page scrolls to bring the bottom of the table to the bottom of the window. But if you use the default parameter (true or empty),

```
myTable.scrollIntoView()
```

the page scrolls as far as it can in an effort to align the top of the element as closely as possible to the top of the window. The page cannot scroll beyond its normal scrolling maximum (although if the element is a positioned element, you can use dynamic positioning to place it wherever you want — including "off the page"). Also, if you shrink the window and try to scroll the top of the table to the top of the window, be aware that the table element contains a caption element so the caption is flush with the top of the window.

Related Items: window.scroll(), window.scrollBy(), window.scrollTo() methods.

setActive()

Returns: Nothing.
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The setActive() method lets a script designate an element object as the active element. However, unlike the focus() method, the window does not scroll the active element into

view. Any `onFocus` event handler defined for the element fires when `setActive()` is invoked, without the browser giving the element focus.

Example

Use The Evaluator (Chapter 13) to compare the `setActive()` and `focus()` methods. With the page scrolled to the top and the window sized so that you cannot see the sample checkbox near the bottom of the page, enter the following statement into the top text box:

```
document.forms[1].myCheckbox.setActive()
```

Scroll down to see that the checkbox has operational focus (press the spacebar to see). Now, scroll back to the top and enter the following:

```
document.forms[1].myCheckbox.focus()
```

This time, the checkbox gets focus and the page automatically scrolls the object into view.

Related Item: `focus()` method.

`setAttribute("attributeName", value[, caseSensitivity])`

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `setAttribute()` method assigns a new value to an existing attribute of the current object or inserts an entirely new attribute name–value pair among the attributes of the current object. This method represents an alternative syntax to setting a property of the object directly.

The W3C DOM Level 2 standard recommends `getAttribute()` and `setAttribute()` for reading and writing element object attribute values, rather than reading and writing those values by way of their corresponding properties. While using these methods is certainly advisable for XML elements, the same DOM standard sends conflicting signals by defining all kinds of properties for HTML element objects. Browsers, of course, will support access via properties well into the future, so don't feel obligated to change your ways just yet.

The first two parameters of `setAttribute()` are required. The first is the name of the attribute. The default behavior of this method respects the case of the attribute name. Therefore, if you use `setAttribute()` to adjust the value of an existing attribute in default mode, the first parameter must match the case of the attribute as known by the object model for the current document. Remember that all names of all attributes assigned as inline source code attributes are automatically converted to lowercase letters.

A value you assign to the attribute is the second parameter. For cross-browser compatibility, the value should be either a string or Boolean data type.

IE provides an optional third parameter to control the case-sensitivity issue for the attribute name. The default value (`true`) has a different impact on your object depending on whether you use `setAttribute()` to assign a new attribute or reassign an existing one. In the former case, the third parameter as `true` means that the attribute name assigned to the object observes the case of the first parameter. In the latter case, the third parameter as `true` means that the attribute isn't reassigned unless the first parameter matches the case of the attribute currently associated with the object. Instead, a new attribute with a different case sequence is created.

Attempting to manage the case-sensitivity of newly created attributes is fraught with peril, especially if you try to reuse names but with different case sequences. I strongly recommend using default case-sensitivity controls for `setAttribute()` and `getAttribute()`.

IE4+ imposes some limitations on the action resulting from the `setAttribute()` method. Any attribute you add via `setAttribute()` does not become part of the `attributes` collection associated with the element. While you can extract the value of such a newly added attribute via `getAttribute()`, you cannot access the new attribute from the `attributes` collection. Thus, after creating a new attribute as follows:

```
document.getElementById("myTable").setAttribute("currYear",
   (new Date()).getFullYear());
```

you can access that attribute value through either of the following two statements:

```
var tableYear = document.getElementById("myTable").getAttribute("curryear");
var tableYear = document.getElementById("myTable").currYear;
```

However, you cannot access the attribute value with the following statement:

```
var tableYear = document.getElementById("myTable").attributes["currYear"];
```

See also the W3C DOM facilities for treating attributes as node objects in the discussions of the `getAttributeNode()` and `removeAttributeNode()` methods earlier in this chapter.

Example

Use The Evaluator (Chapter 13) to experiment with the `setAttribute()` method for the elements in the page. For IE4, use the `document.all` notation; IE5+ and W3C browsers understand the W3C standard `getElementById()` method of addressing an element.

Setting attributes can have immediate impact on the layout of the page (just as setting an object's properties can). Enter the following sample statements into the top text box to view attribute values.

IE4+:

```
document.all.myTable.setAttribute("width", "80%")
document.all.myTable.setAttribute("border", "5")
```

IE5+/W3C:

```
document.getElementById("myTable").setAttribute("width", "80%")
document.getElementById("myTable").setAttribute("border", "5")
```

Related Items: `attributes` property; `document.createAttribute()`, `getAttribute()`, `getAttributeNode()`, `removeAttribute()`, `removeAttributeNode()`, `setAttributeNode()` methods.

setAttributeNode()

(See `removeAttributeNode()`)

setAttributeNodeNS("*attributeNode*")

Returns: `Attribute` object.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

This method inserts or replaces an attribute in the current element. The sole parameter is an attribute object, and the return value is a reference to the newly inserted attribute object. When the method is invoked, the browser looks for a pairing of local name and namespace URI between the nodes. If there is a match, the node replaces the matched node; otherwise, the node is inserted.

Related Items: `attributes`, `namespaceURI`, `localName` properties; `removeAttributeNS()`, `getAttributeNS()`, and `setAttributeNS()` methods.

***elementObject*.setAttributeNS()**

setAttributeNS("*namespaceURI*", "*qualifiedName*", "*value*")

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

This method inserts or replaces an attribute in the current element, as specified in the three parameters. The first parameter of the method is a URI string matching a URI assigned to a label in the document. The second parameter is the local name portion of the attribute whose value you are getting. If a match is found among these parameters, the value in the third parameter is assigned to the existing attribute; otherwise, the value is inserted as a new attribute.

Related Items: `attributes`, `namespaceURI`, `localName` properties; `removeAttributeNS()`, `getAttributeNS()`, and `setAttributeNodeNS()` methods.

setCapture(*containerBoolean*)

(See `releaseCapture()`)

setExpression("*propertyName*", "*expression*",["*language*"])

Returns: Nothing.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

Use the `setExpression()` method to assign the result of an executable expression to the value of an element object property. This method can assign values to both HTML element objects and style objects that belong to them.

The `setExpression()` method is a scripted way of assigning expressions to attributes. But you can also assign expressions directly to stylesheet definitions in the HTML tag of an element using the `expression()` syntax, as in the following example:

```
<p style="width:expression(document.body.style.width * 0.75)">
```

The `setExpression()` method requires three parameters. The first parameter is the name of the property (in string form) to which you assign the expression. Property names are case-sensitive. The second parameter is a string form of the expression to be evaluated to supply a value for the property. Expressions can refer to global variables or properties of other objects in the same document (provided the property is anything other than an array). An expression may also contain math operators.

Pay close attention to the data type of the evaluated value of the expression. The value must be a valid data type for the property. For example, the URL of the body background image must be a string. But for numeric values, you can generally use number and string types interchangeably because the values are converted to the proper type for the property. Even for expressions that evaluate to numbers, encase the expression inside quotes. It may not be necessary in all cases, but if you get into the habit of using quotes, you'll have fewer problems for strings or complex expressions that require them.

You are not limited to using JavaScript as the language for the expression because you can also specify the scripting language of the expression in the optional third parameter. Acceptable parameter values for the language are

```
JScript
JavaScript
VBScript
```

For all intents and purposes, JScript and JavaScript are the same. Both languages are ECMA-262 compatible. `JScript` is the default value for the `language` parameter.

One reason to use `setExpression()` for dynamic properties is to let the property always respond to the current conditions on the page. For example, if you set a property that is dependent on the current width of the body, then you want a recalculation that is applied to the property if the user resizes the window. The browser automatically responds to many events and updates any dynamic properties. In essence, the browser recalculates the expressions and applies the new values to the property. Keyboard events, in particular, trigger this kind of automatic recalculation for you. But if your scripts perform actions on their own (in other words, not triggered by events), then your scripts need to force the recalculation of the expressions. The `document.recalc()` method takes care of this, but you must invoke it to force the recalculation of dynamic properties in these cases.

Example

Listing 15-32 shows the `setExpression()`, `recalc()`, and `getExpression()` methods at work in a DHTML-based clock. Figure 15-1 shows the clock. As time clicks by, the bars for hours, minutes, and seconds adjust their widths to reflect the current time. At the same time, the `innerHTML` of `span` elements to the right of each bar display the current numeric value for the bar.

The dynamically calculated values in this example are based on the creation of a new date object over and over again to get the current time from the client computer clock. It is from the date object (stored in the variable called `now`) that the hour, minute, and second values are retrieved. Some other calculations are involved so that a value for one of these time components is converted into a pixel value for the width of the bars. The bars are divided into 24 (for the hours) and 60 (for the minutes and seconds) parts, so the scale for the two types differs. For the 60-increment bars in this application, each increment is set to 5 pixels (stored in `shortWidth`); the 24-increment bars are 2.5 times the `shortWidth`.

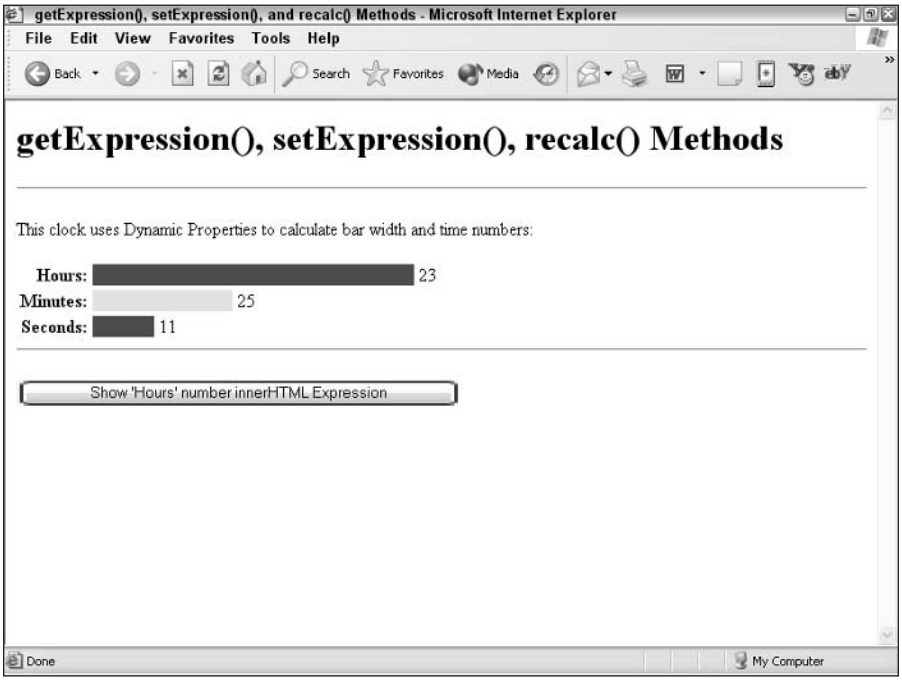


Figure 15-1: A bar graph clock created with dynamic expressions.

As the document loads, the three `span` elements for the colored bars are given no width, which means that they assume the default width of zero. But after the page loads, the `onload` event handler invokes the `init()` function, which sets the initial values for each bar's width and the text (`innerHTML`) of the three labeled spans. Once these initial values are set, the `init()` function invokes the `updateClock()` function.

In the `updateClock()` function, a new date object is created for the current instant. The `document.recalc()` method is called, instructing the browser to recalculate the expressions that were set in the `init()` function and assign the new values to the properties. To keep the clock "ticking," the `setTimeout()` method is set to invoke this same `updateClock()` function in one second.

To see what the `getExpression()` method does, you can click the button on the page. It simply displays the returned value for one of the attributes that you assign using `setExpression()`.

Listing 15-32: **Dynamic Properties**

```
<html>
   <head>
      <title>getExpression(), setExpression(), and recalc() Methods</title>
      <style type="text/css">
      th {text-align:right}
      span {vertical-align:bottom}
      </style>
      <script type="text/javascript">
      var now = new Date();
      var shortWidth = 5;
      var multiple = 2.5;

      function init() {
         with (document.all) {
            hoursBlock.style.setExpression("width","now.getHours() *
               shortWidth * multiple","jscript");
            hoursLabel.setExpression("innerHTML","now.getHours()","jscript");
            minutesBlock.style.setExpression("width","now.getMinutes() *
               shortWidth","jscript");
            minutesLabel.setExpression("innerHTML","now.getMinutes()",
               "jscript");
            secondsBlock.style.setExpression("width","now.getSeconds() *
               shortWidth","jscript");
            secondsLabel.setExpression("innerHTML","now.getSeconds()",
               "jscript");
         }

         updateClock();
      }

      function updateClock() {
         now = new Date();
         document.recalc();
         setTimeout("updateClock()",1000);
      }

      function showExpr() {
         alert("Expression for the \'Hours\' innerHTML property is:\r\n" +
            document.getElementById("hoursLabel").getExpression("innerHTML") +
            ".");      }
```

```
      </script>
   </head>
   <body onload="init()">
      <h1>getExpression(), setExpression(), recalc() Methods</h1>
      <hr />
      <p>This clock uses Dynamic Properties to calculate bar width and time
         numbers:</p>
      <table border="0">
         <tr>
            <th>Hours:</th>
            <td><span id="hoursBlock" style="background-color:red"></span>
                <span id="hoursLabel"></span></td>
         </tr>
         <tr>
            <th>Minutes:</th>
            <td><span id="minutesBlock" style="background-color:yellow"></span>
                <span id="minutesLabel"></span></td>
         </tr>
         <tr>
            <th>Seconds:</th>
            <td><span id="secondsBlock" style="background-color:green"></span>
                <span id="secondsLabel"></span></td>
         </tr>
      </table>
      <hr />
      <form>
         <input type="button" value="Show 'Hours' number innerHTML Expression"
         onclick="showExpr()" />
      </form>
   </body>
</html>
```

Related Items: `document.recalc()`, `removeExpression()`, `setExpression()` methods.

`swapNode(`*`otherNodeObject`*`)`

Returns: `Node` object reference.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `swapNode()` method exchanges the positions of two nodes within an element hierarchy. Contents of both nodes are preserved in their entirety during the exchange. The single parameter must be a valid node object (perhaps created with `document.createElement()` or copied from an existing node). A return value is a reference to the object whose `swapNode()` method was invoked.

Example

See Listing 15-31 (the `replaceNode()` method) for an example of the `swapNode()` method in action.

Related Items: `removeChild()`, `removeNode()`, `replaceChild()`, `replaceNode()` methods.

`tags("`*`tagName`*`")`

Returns: Array of element objects.
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

***elementObjectCollection*.tags()**

The `tags()` method does not belong to every element, but it is a method of every collection of objects (such as `all`, `forms`, and `elements`). The method is best thought of as a kind of filter for the elements that belong to the current collection. For example, to get an array of all `p` elements inside a document, use this expression:

```
document.all.tags("P")
```

You must pass a parameter string consisting of the tag name you wish to extract from the collection. The tag name is case-insensitive.

The return value is an array of references to the objects within the current collection whose tags match the parameter. If there are no matches, the returned array has a length of zero. If you need cross-browser compatibility, use the `getElementsByTagName()` method described earlier in this chapter, and pass a wildcard value of `"*"`.

Example

Use The Evaluator (Chapter 13) to experiment with the `tags()` method. Enter the following statements one at a time into the upper text box and study the results:

```
document.all.tags("div")
document.all.tags("div").length
myTable.all.tags("td").length
```

Because the `tags()` method returns an array of objects, you can use one of those returned values as a valid element reference:

```
document.all.tags("form")[1].elements.tags("input").length
```

Related Item: `getElementsByTagName()` method.

urns("*behaviorURN*")

Returns: Array of element objects.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `urns()` method does not belong to every element, but it is a method of every collection of objects. You must pass a parameter string consisting of the URN (Uniform Resource Name) of a behavior resource (most typically `.htc`) assigned to one or more elements of the collection. The parameter does not include the extension of the filename. If there is no matching behavior URN for the specified parameter, the `urns()` method returns an array of zero length. This method is related to the `behaviorUrns` property, which contains an array of behavior URNs assigned to a single element object.

Example

In case the `urns()` method is reconnected in the future, you can add a button and function to Listing 15-19b that reveals whether the `makeHot.htc` behavior is attached to the `myP` element. Such a function looks like this:

```
function behaviorAttached() {
   if (document.all.urns("makeHot")) {
      alert("There is at least one element set to \'makeHot\'.");
   }
}
```

Related Item: `behaviorUrns` property.

Event handlers

onactivate
onbeforedeactivate
ondeactivate

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The onactivate and ondeactivate event handlers are very similar to the onfocus and onblur event handlers, respectively. If an element receives focus, the onactivate event fires for that element just before the onfocus event fires; conversely, just prior to the element losing focus, events fire in the sequence onbeforedeactivate, ondeactivate, onblur. Only elements that, by their nature, can accept focus (for example, links and form input controls) or that have a tabindex attribute set can become the active element (and therefore fire these events).

WinIE5.5+ maintains the original onfocus and onblur event handlers. But because the behaviors are so close to those of the onactivate and ondeactivate events, I don't recommend mixing the old and new event handler names in your coding style. If you script exclusively for WinIE5.5+, you can use the new terminology throughout.

Example

You can modify Listing 15-34 later in this chapter by substituting onactivate for onfocus and ondeactivate for onblur.

Use The Evaluator (Chapter 13) to experiment with the onbeforedeactivate event handler. To begin, set the myP element so it can accept focus:

```
myP.tabIndex = 1
```

If you repeatedly press the Tab key, the myP paragraph will eventually receive focus — indicated by the dotted rectangle around it. To see how you can prevent the element from losing focus, assign an anonymous function to the onbeforedeactivate event handler, as shown in the following statement:

```
myP.onbeforedeactivate = new Function("event.returnValue=false")
```

Now you can press Tab all you like or click other focusable elements all you like, and the myP element will not lose focus until you reload the page (which clears away the event handler). Please do not do this on your pages unless you want to infuriate and alienate your site visitors.

Related Items: onblur, onfocus event handlers.

onbeforecopy

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The onbeforecopy event handler fires before the actual copy action takes place whenever the user initiates a content copy action via the Edit menu (including the Ctrl+C keyboard shortcut) or the right-click context menu. If the user accesses the Copy command via the Edit or context menu, the onbeforecopy event fires before either menu displays. In practice, the event may fire twice even though you expect it only once. Just because the onbeforecopy event fires, it does not guarantee that a user will complete the copy operation (for example, the context menu may close before the user makes a selection).

Unlike paste-related events, the onbeforecopy event handler does not work with form input elements. Just about any other HTML element is fair game, however.

Example

You can use the onbeforecopy event handler to preprocess information prior to an actual copy action. In Listing 15-33, the function invoked by the second paragraph element's onbeforecopy event handler selects the entire paragraph so that the user can select any character(s) in the paragraph to copy the entire paragraph into the clipboard. You can paste the results into the text area to verify the operation. By assigning the paragraph selection to the onbeforecopy event handler, the page notifies the user about what the copy operation will entail prior to making the menu choice. Had the operation been deferred to the oncopy event handler, the selection would have been made after the user chose Copy from the menu.

Listing 15-33: **The onbeforecopy Event Handler**

```
<html>
   <head>
      <title>onbeforecopy Event Handler</title>
      <script type="text/javascript">
      function selectWhole() {
         var obj = window.event.srcElement;
         var range = document.body.createTextRange();
         range.moveToElementText(obj);
         range.select();
         event.returnValue = false;
      }
      </script>
   </head>
   <body>
      <h1>onbeforecopy Event Handler</h1>
      <hr />
      <p>Select one or more characters in the following paragraph. Then execute
         a Copy command via Edit or context menu.</p>
      <p id="myP" onbeforecopy="selectWhole()">Lorem ipsum dolor sit amet,
         consectetaur adipisicing elit, sed do eiusmod tempor incididunt ut
         labore et dolore magna aliqua. Ut enim adminim veniam, quis nostrud
         exercitation ullamco laboris nisi ut aliquip ex ea commodo
         consequat.</p>
      <form>
         <p>Paste results here:<br />
             <textarea name="output" cols="60" rows="5">
            </textarea></p>
      </form>
   </body>
</html>
```

Related Items: onbeforecut, oncopy event handlers.

onbeforecut

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The onbeforecut event handler fires before the actual cut action takes place whenever the user initiates a content cut via the Edit menu (including the Ctrl+X keyboard shortcut) or the right-click context menu. If the user accesses the Cut command via the Edit or context menu, the onbeforecut event fires before either menu displays. In practice, the event may fire twice even though you expect it only once. Just because the onbeforecut event fires, it does not

guarantee that a user will complete the cut operation (for example, the context menu may close before the user makes a selection). If you add the onbeforecut event handler to an HTML element, the context menu usually disables the Cut menu item. But assigning a JavaScript call to this event handler brings the Cut menu item to life.

Example

You can use the onbeforecut event handler to preprocess information prior to an actual cut action. You can try this by editing a copy of Listing 15-33, changing the onbeforecopy event handler to onbeforecut. Notice that in its original form, the example does not activate the Cut item in either the context or Edit menu when you select some text in the second paragraph. But by assigning a function to the onbeforecut event handler, the menu item is active, and the entire paragraph is selected from the function that is invoked.

Related Items: onbeforecopy, oncut event handlers.

onbeforedeactivate

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

(See onactivate event handler)

onbeforeeditfocus

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The onbeforeeditfocus event handler is triggered whenever you edit an element on a page in an environment such as Microsoft's DHTML Editing ActiveX control or with the editable page content feature of IE5.5+. This discussion focuses on the latter scenario because it is entirely within the scope of client-side JavaScript. The onbeforeeditfocus event fires just before the element receives its focus. (There may be no onscreen feedback that editing is turned on unless you script it yourself.) The event fires each time a user clicks the element, even if the element just received edit focus elsewhere in the same element.

Example

Use The Evaluator (Chapter 13) to explore the onbeforeeditfocus in WinIE5.5+. In the following sequence, you assign an anonymous function to the onbeforeeditfocus event handler of the myP element. The function turns the text color of the element to red when the event handler fires:

```
myP.onbeforeeditfocus = new Function("myP.style.color='red'")
```

Now turn on content editing for the myP element:

```
myP.contentEditable = true
```

If you now click inside the myP element on the page to edit its content, the text turns to red before you begin editing. In a page scripted for this kind of user interface, you would include some control that turns off editing and changes the color to normal.

Related Items: document.designMode, contentEditable, isContentEditable properties.

onbeforepaste

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

Like onbeforecopy and onbeforecut, the onbeforepaste event occurs just prior to the display of either the context or menu bar Edit menu when the current object is selected (or has a selection within it). The primary value of this event comes when you use scripts to control the

copy-and-paste process of a complex object. Such an object may have multiple kinds of data associated with it, but your script captures only one of the data types. Or, you may want to put some related data about the copied item (for example, the `id` property of the element) into the clipboard. By using the `onbeforepaste` event handler to set the `event.returnValue` property to `false`, you guarantee that the pasted item is enabled in the context or Edit menu (provided the clipboard is holding some content). A handler invoked by `onpaste` should then apply the specific data subset from the clipboard to the currently selected item.

Example

See Listing 15-45 for the `onpaste` event handler (later in this chapter) to see how the `onbeforepaste` and `onpaste` event handlers work together.

Related Items: `oncopy`, `oncut`, `onpaste` event handlers.

onblur

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `onblur` event fires when an element that has focus is about to lose focus because some other element is about to receive focus. For example, a text input element fires the `onblur` event when a user tabs from that element to the next one inside a form. The `onblur` event of the first element fires before the `onfocus` event of the next element.

The availability of the `onblur` event has expanded with succeeding generations of script-capable browsers. In the earlier versions, blur and focus were largely confined to text-oriented input elements (including the `select` element). These are safe to use with all scriptable browser versions. The `window` object received the `onblur` event handler starting with NN3 and IE4. IE4 also extended the event handler to more form elements, predominantly on the Windows operating system because that OS has a user interface clue (the dotted rectangle) when items such as buttons and links receive focus (so that you may act upon them by pressing the keyboard's spacebar). For IE5+, the `onblur` event handler is available to virtually every HTML element. For most of those elements, however, blur and focus are not possible unless you assign a value to the `tabindex` attribute of the element's tag. For example, if you assign `tabindex="1"` inside a `<p>` tag, the user can bring focus to that paragraph (highlighted with the dotted rectangle in Windows) by clicking the paragraph or pressing the Tab key until that item receives focus in sequence.

If you plan to use the `onblur` event handler on window or text-oriented input elements, be aware that there might be some unexpected and undesirable consequences of scripting for the event. For example, in IE, a `window` object that has focus loses focus (and triggers the `onblur` event) if the user brings focus to any element on the page (or even clicks a blank area on the page). Similarly, the interaction between `onblur`, `onfocus`, and the `alert()` dialog box can be problematic with text input elements. This is why I generally recommend using the `onchange` event handler to trigger form validation routines. If you should employ both the `onblur` and `onchange` event handler for the same element, the `onchange` event fires before `onblur`. For more details about using this event handler for data validation, see Chapter 43 on the CD-ROM.

WinIE5.5+ adds the `ondeactivate` event handler, which fires immediately before the `onblur` event handler. Both the `onblur` and `ondeactivate` events can be blocked if the `onbeforedeactivate` event handler function sets `event.returnValue` to `false`.

Example

More often than not, a page author uses the `onblur` event handler to exert extreme control over the user, such as preventing a user from exiting out of a text box unless that user types

something into the box. This is not a Web-friendly practice, and it is one that I discourage because there are intelligent ways to ensure a field has something typed into it before a form is submitted (see Chapter 43 on the CD-ROM). Listing 15-34 simply demonstrates the impact of the `tabindex` attribute in a WinIE5 element with respect to the `onblur` and `onfocus` events. Notice that as you press the Tab key, only the second paragraph issues the events even though all three paragraphs have event handlers assigned to them.

Listing 15-34: **onblur and onfocus Event Handlers**

```
<html>
   <head>
      <title>onblur and onblur Event Handlers</title>
      <script type="text/javascript">
      function showBlur() {
         var id = event.srcElement.id;
         alert("Element \"" + id + "\" has blurred.");
      }

      function showFocus() {
         var id = event.srcElement.id;
         alert("Element \"" + id + "\" has received focus.");
      }
      </script>
   </head>
   <body>
      <h1 id="H1" tabindex="2">onblur and onblur Event Handlers</h1>
      <hr />
      <p id="P1" onblur="showBlur()" onfocus="showFocus()">Lorem ipsum dolor
         sit amet, consectetaur adipisicing elit, sed do eiusmod tempor
         incididunt ut labore et dolore magna aliqua. Ut enim adminim veniam,
         quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
         commodo consequat.</p>
      <p id="P2" tabindex="1" onblur="showBlur()" onfocus="showFocus()">Bis
         nostrud exercitation ullam mmodo consequet. Duis aute involuptate
         velit esse cillum dolore eu fugiat nulla pariatur. At vver eos et
         accusam dignissum qui blandit est praesent luptatum delenit
         aigueexcepteur sint occae.</p>
      <p id="P3" onblur="showBlur()" onfocus="showFocus()">Unte af phen
         neigepheings atoot Prexs eis phat eit sakem eit vory gast te Plok
         peish ba useing phen roxas. Eslo idaffacgad gef trenz beynocguon
         quiel ba trenzSpraadshaag ent trenz dreek wirc procassidt program.</p>
   </body>
</html>
```

Related Items: `blur()`, `focus()` methods; `ondeactivate`, `onbeforedeactivate`, `onfocus`, `onactivate` event handlers.

onclick

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `onclick` event fires when a user presses down (with the primary mouse button) and releases the button with the pointer atop the element (both the down and up strokes must be within the rectangle of the same element). The event also fires with non-mouse click equivalents in operating systems such as Windows. For example, you can use the keyboard to give

focus to a clickable object and then press the spacebar or Enter key to perform the same action as clicking the element. In IE, if the element object supports the `click()` method, the `onclick` event fires with the invocation of that method (notice that this does not apply to Navigator or other browsers).

The `onclick` event is closely related to other mouse events. The other related events are `onmousedown`, `onmouseup`, and `ondoubleclick`. The `onmousedown` event fires when the user makes contact with the mouse switch on the downstroke of a click action. Next comes the `onmouseup` event (when the contact breaks). Only then does the `onclick` event fire — provided that the `onmousedown` and `onmouseup` events have fired in the same object. See the discussions on the `onmousedown` and `onmouseup` events later in this chapter for examples of their usage.

Interaction with the `ondblclick` event is simple: the `onclick` event fires first (after the first click), followed by the `ondblclick` event (after the second click). See the discussion of the `ondblclick` event handler later in this chapter for more about the interaction of these two event handlers.

When used with objects that have intrinsic actions when users click them (namely links and areas), the `onclick` event handler can perform all of the action — including navigating to the destination normally assigned to the `href` attribute of the element. For example, to be compatible with all scriptable browsers, you can make an image clickable if you surround its tag with an `<a>` link tag. This lets the `onclick` event of that tag substitute for the missing `onclick` event handler of earlier `<img>` tags. If you assign an `onclick` event handler without special protection, the event handler will execute and the intrinsic action of the element will be carried out. Therefore, you need to block the intrinsic action. To accomplish this, the event handler must evaluate to the statement `return false`. You can do this in two ways. The first is to append a `return false` statement to the script statement assigned to the event handler:

```
<a href="#" onclick="yourFunction(); return false"><img...></a>
```

As an alternative, you can let the function invoked by the event handler supply the `false` part of the `return false` statement, as shown in the following sequence:

```
function yourFunction() {
   [statements that do something here]
   return false;
}
...
<a href="#" onclick="return yourFunction()"><img...></a>
```

Either methdology is acceptable. A third option is to not use the `onclick` event handler at all, but assign a `javascript:` pseudo-URL to the `href` attribute (see the `Link` object in Chapter 19).

The event model in IE4+ provides one more way to prevent the intrinsic action of an object from firing when a user clicks it. If the `onclick` event handler function sets the `returnValue` property of the event object to `false`, the intrinsic action is cancelled. Simply include the following statement in the function invoked by the event handler:

```
event.returnValue = false;
```

The event model of the W3C DOM has a different approach to cancelling the default action. In the event handler function for an event, invoke the *`eventObj`*`.cancelDefault()` method.

A common mistake made by scripting beginners is to use a submit type input button as a button intended to perform some script action rather than submitting a form. The typical scenario is an `input` element of type `submit` assigned an `onclick` event handler to perform some local action. The submit input button has an intrinsic behavior, just like links and areas. While you can block the intrinsic behavior, as just described, you should use an `input` element of type `button`.

If you are experiencing difficulty with an implementation of the `onclick` event handler (such as trying to find out which mouse button was used for the click), it may be that the operating system or default browser behavior is getting in the way of your scripting. But you can usually get what you need via the `onmousedown` event handler. (The `onmouseup` event may not fire when you use the secondary mouse button to click an object.) Use the `onclick` event handler whenever possible to capture user clicks because this event behaves most like users are accustomed to in their daily computing work. But fall back on `onmousedown` in an emergency.

Example

The `onclick` event handler is one of the simplest to grasp and use. Listing 15-35 demonstrates its interaction with the `ondblclick` event handler and shows you how to prevent a link's intrinsic action from activating when combined with `click` events. As you click and/or double-click the link, the status bar displays a message associated with each event. Notice that if you double-click, the `click` event fires first with the first message immediately replaced by the second. For demonstration purposes, I show both backward-compatible ways of cancelling the link's intrinsic action. In practice, decide on one style and stick with it.

Listing 15-35: **Using onclick and ondblclick Event Handlers**

```
<html>
   <head>
      <title>onclick and ondblclick Event Handlers</title>
      <script type="text/javascript">
      var timeout;
      function clearOutput() {
         document.getElementById("clickType").innerHTML = "";
      }
      function showClick() {
         document.getElementById("clickType").innerHTML = "single";
         clearTimeout(timeout);
         timeout = setTimeout("clearOutput()", 3000);
      }

      function showDblClick() {
         document.getElementById("clickType").innerHTML = "double";
         clearTimeout(timeout);
         timeout = setTimeout("clearOutput()", 3000);
      }
      </script>
   </head>
   <body>
      <h1>onclick and ondblclick Event Handlers</h1>
      <hr />
      <a href="#" onclick="showClick();return false"
      ondblclick="return showDblClick()">A sample link.</a>
      (Click type: <span id="clickType"></span>)
   </body>
</html>
```

Related Items: `click()` method; `oncontextmenu`, `ondblclick`, `onmousedown`, `onmouseup` event handlers.

oncontextmenu

Compatibility: WinIE5+, MacIE-, NN7+, Moz1+, Safari-

The oncontextmenu event fires when the user clicks an object with the secondary (usually the right-hand) mouse button. The only click-related events that fire with the secondary button are onmousedown and oncontextmenu.

To block the intrinsic application menu display of the oncontextmenu event, use any of the three event cancellation methodologies available in WinIE5+ (as just described in the onclick event handler description: two variations of evaluating the event handler to return false; assigning false to the event.returnValue property). It is not uncommon to wish to block the context menu from appearing so that users are somewhat inhibited from downloading copies of images or viewing the source code of a frame. Be aware, however, that if a user turns Active Scripting off in WinIE5+, the event handler cannot prevent the context menu from appearing.

Another possibility for this event is to trigger the display of a custom context menu constructed with other DHTML facilities. In this case, you must also disable the intrinsic context menu so that both menus do not display at the same time.

Example

See Listing 15-30 earlier in this chapter for an example of using the oncontextmenu event handler with a custom context menu.

Related Items: releaseCapture(), setCapture() methods.

oncontrolselect

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The oncontrolselect event fires just before a user makes a selection on an editable element while the page is in edit mode. It's important to note that it is the element itself that is selected in order to trigger this event, not the content within the element.

Related Items: onresizeend, onresizestart event handlers.

oncopy
oncut

Compatibility: WinIE5+, MacIE4+, NN-, Moz-, Safari-

The oncopy and oncut events fire immediately after the user or script initiates a copy or cut edit action on the current object. Each event is preceded by its associated "before" event, which fires before any Edit or context menu appears (or before the copy or cut action, if initiated by keyboard shortcut).

Use these event handlers to provide edit functionality to elements that don't normally allow copying or cutting. In such circumstances, you need to enable the Copy or Cut menu items in the context or Edit menu by setting the event.returnValue for the onbeforecopy or onbeforecut event handlers to false. Then your oncopy or oncut event handlers must manually stuff a value into the clipboard by way of the setdata() method of the clipboardData object. If you use the setdata() method in your oncopy or oncut event handler, you must also set the event.returnValue property to false in the handler function to avoid the default copy or cut action from wiping out your clipboard contents.

Because you are in charge of what data is stored in the clipboard, you are not limited to a direct copy of the data. For example, you might wish to store the value of the src property of an image object so that the user can paste it elsewhere on the page.

In the case of the oncut event handler, your script is also responsible for cutting the element or selected content from the page. To eliminate all of the content of an element, you can set the element's innerHTML or innerText property to an empty string. For a selection, use the selection.createRange() method to generate a TextRange object whose contents you can manipulate through the TextRange object's methods.

Example

Listing 15-36 shows both the onbeforecut and oncut event handlers in action (as well as onbeforepaste and onpaste). Notice how the handleCut() function not only stuffs the selected word into the clipboardData object, but it also erases the selected text from the table cell element from where it came. If you replace the onbeforecut and oncut event handlers with onbeforecopy and oncopy (and change handleCut() to not eliminate the inner text of the event source element), the operation works with copy and paste instead of cut and paste. I demonstrate this later in the chapter in Listing 15-45.

Listing 15-36: **Cutting and Pasting under Script Control**

```
<html>
   <head>
      <title>onbeforecut and oncut Event Handlers</title>
      <style type="text/css">
      td {text-align:center}
      th {text-decoration:underline}
      .blanks {text-decoration:underline}
      </style>
      <script type="text/javascript">
      function selectWhole() {
         var obj = window.event.srcElement;
         var range = document.body.createTextRange();
         range.moveToElementText(obj);
         range.select();
         event.returnValue = false;
      }

      function handleCut() {
         var rng = document.selection.createRange();
         clipboardData.setData("Text",rng.text);
         var elem = event.srcElement;
         elem.innerText = "";
         event.returnValue = false;
      }

      function handlePaste() {
         var elem = window.event.srcElement;
         if (elem.className == "blanks") {
            elem.innerHTML = clipboardData.getData("Text");
         }
         event.returnValue = false;
      }

      function handleBeforePaste() {
         var elem = window.event.srcElement;
         if (elem.className == "blanks") {
```

Continued

Listing 15-36 *(continued)*

```
            event.returnValue = false;
         }
      }
      </script>
   </head>
   <body>
      <h1>onbeforecut and oncut Event Handlers</h1>
      <hr />
      <p>Your goal is to cut and paste one noun and one adjective from the
         following table into the blanks of the sentence. Select a word from
         the table and use the Edit or context menu to cut it from the table.
         Select one or more spaces of the blanks in the sentence and choose
         Paste to replace the blank with the clipboard contents.</p>
      <table cellpadding="5" onbeforecut="selectWhole()" oncut="handleCut()">
         <tr>
            <th>Nouns</th>
            <th>Adjectives</th>
         </tr>
         <tr>
            <td>truck</td>
            <td>round</td>
         </tr>
         <tr>
            <td>doll</td>
            <td>red</td>
         </tr>
         <tr>
            <td>ball</td>
            <td>pretty</td>
         </tr>
      </table>
      <p id="myP" onbeforepaste="handleBeforePaste()" onpaste="handlePaste()">
         Pat said, "Oh my, the <span id="blank1"
         class="blanks">     </span> is so <span
         id="blank2" class="blanks">     </span>!"</p>
      <button onclick="location.reload()">Reset</button>
   </body>
</html>
```

Related Items: `onbeforecopy`, `onbeforecut`, `onbeforepaste`, and `onpaste` event handlers.

ondblclick

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The `ondblclick` event fires after the second click of a double-click sequence. The timing between clicks depends on the client's mouse control panel settings. The `onclick` event also fires, but only after the first of the two clicks.

In general, it is rarely a good design to have an element perform one task when the mouse is single-clicked and a different task if double-clicked. With the event sequence employed in modern browsers, this isn't practical anyway (the `onclick` event always fires, even when the user double-clicks). But it is not uncommon to have the mouse down action perform some helper action. You see this in most icon-based file systems: if you click a file icon, it is highlighted at

mouse down to select the item; you can double-click the item to launch it. In either case, one event's action does not impede the other nor confuse the user.

Example

See Listing 15-35 (for the `onclick` event handler) to see the `ondblclick` event in action.

Related Items: `onclick`, `onmousedown`, `onmouseup` event handlers.

ondrag, ondragend, ondragstart

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `ondrag` event fires after the `ondragstart` event and continues firing repeatedly while the user drags a selection or object on the screen. Unlike the `onmousemove` event, which fires only as the cursor moves on the screen, the `ondrag` event continues to fire even when the cursor is stationary. In the WinIE5+ environment, users can drag objects to other browser windows or other applications. The event fires while the dragging extends beyond the browser window.

Because the event fires regardless of what is underneath the dragged object, you can use it in a game or training environment in which the user has only a fixed amount of time to complete a dragging operation (for example, matching similar pairs of objects). If the browser accommodates downloadable cursors, the `ondrag` event could cycle the cursor through a series of cursor versions to resemble an animated cursor.

Understanding the sequence of drag-related events during a user drag operation can be helpful if your scripts need to micromanage the actions (usually not necessary for basic drag-and-drop operations). Consider the drag-and-drop operation shown in Figure 15-2.

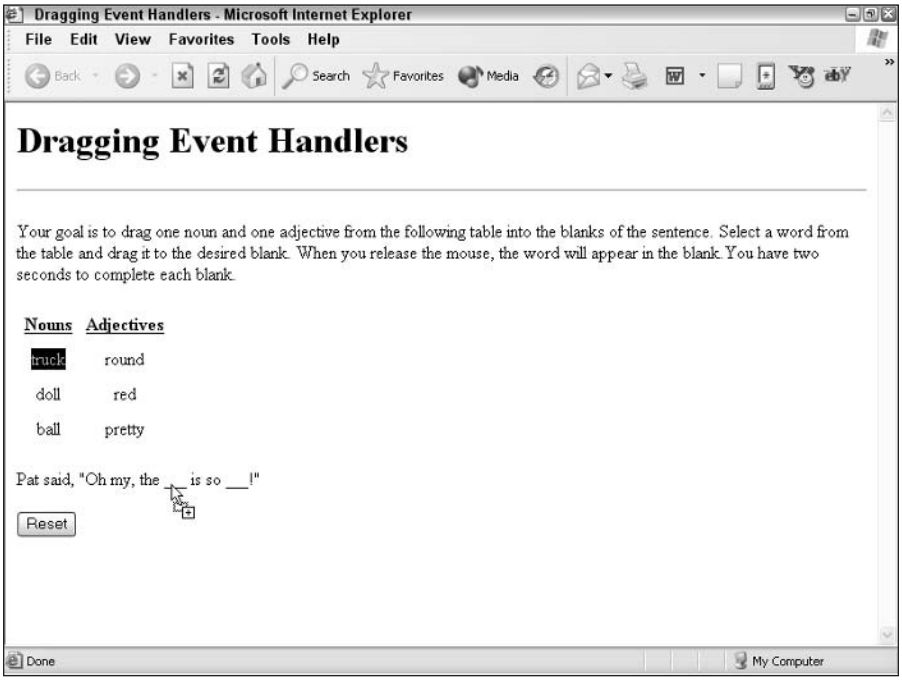


Figure 15-2: A typical drag-and-drop operation.

It helps to imagine that the cells of the table with draggable content are named like spreadsheet cells: "truck" is cell A1; "round" is B1; "doll" is A2; and so on. During the drag operation, many objects are the targets of a variety of drag-related events. Table 15-10 lists the event sequence and the event targets.

Table 15-10: Events and Their Targets During a Typical Drag-and-Drop Operation

Event	*Target*	*Discussion*
ondragstart	cell A1	The very first event that fires during a drag-and-drop operation.
ondrag	cell A1	Fires continually on this target throughout the entire operation. Other events get interspersed, however.
ondragenter	cell A1	Even though the cursor hasn't moved from cell A1 yet, the ondragenter event fires upon first movement within the source element.
ondragover	cell A1	Fires continually on whatever element the cursor rests on at that instant. If the user simply holds the mouse button down and does not move the cursor during a drag, the ondrag and ondragover events fire continually, alternating between the two.
(repetition)	cell A1	ondrag and ondragover events fire alternately while the cursor remains atop cell A1.
ondragenter	table	The table element, represented by the border and/or cell padding, receives the ondragenter event when the cursor touches its space.
ondragleave	cell A1	Notice that the ondragleave event fires after the ondragenter event fires on another element.
ondrag	cell A1	Still firing away.
ondragover	table	The source element for this event shifts to the table because that's what the cursor is "over" at this instant. If the cursor doesn't move from this spot, the ondrag (cell A1) and ondragover (table) events continue to fire in turn.
ondragenter	cell B1	The drag is progressing from the table border space to cell B1.
ondragleave	table	
ondrag	cell A1	The ondrag event continues to fire on the cell A1 object.
ondragover	cell B1	The cursor is atop cell B1 now, so the ondragover event fires for that object. Fires multiple times (depending on the speed of the computer and the user's drag action), alternating with the previous ondrag event. [More of the same as the cursor progresses from cell B1 through the table border again to cell B2, the table again, cell B3, and the outermost edge of the table.]
ondragenter	body	Dragging is free of the table and is floating free on the bare body element.
ondragleave	table	Yes, you just left the table.
ondrag	cell A1	Still alive and receiving this event.
ondragover	body	That's where the cursor is now. Fires multiple times (depending on the speed of the computer and the user's drag action), alternating with the previous ondrag event.
ondragenter	blank1	The cursor reaches the span element whose ID is blank1, where the empty underline is.
ondragleave	body	Just left the body for the blank.

Event	*Target*	*Discussion*
ondrag	cell A1	Still kicking.
ondragover	blank1	That's where the cursor is now. Fires multiple times (depending on the speed of the computer and the user's drag action), alternating with the previous ondrag event.
ondrop	blank1	The span element gets the notification of a recent drop.
ondragend	cell A1	The original source element gets the final word that dragging is complete. This event fires even if the drag does not succeed because the drag does not end on a drop target.

In practice, some of the events shown in Table 15-10 may not fire. Much has to do with how many event handlers you trap that need to execute scripts along the way. The other major factor is the physical speed at which the user performs the drag-and-drop operation (which interacts with the CPU processing speed). The kinds of events that are most likely to be skipped are the ondragenter and ondragleave events, and perhaps some ondragover events if the user flies over an object before its ondragover event has a chance to fire.

Despite this uncertainty about drag-related event reliability, you can count on several important ones to fire all the time. The ondragstart, ondrop (if over a drop target), and ondragend events — as well some interstitial ondrag events — will definitely fire in the course of dragging on the screen. All but ondrop direct their events to the source element, while ondrop fires on the target.

Example

Listing 15-37 shows several drag-related event handlers in action. The page resembles the example in Listing 15-36, but the scripting behind the page is quite different. In this example, the user is encouraged to select individual words from the Nouns and Adjectives columns and drag them to the blanks of the sentence. To beef up the demonstration, Listing 15-37 shows you how to pass the equivalent of array data from a drag source to a drag target. At the same time, the user has a fixed amount of time (two seconds) to complete each drag operation.

The ondragstart and ondrag event handlers are placed in the <body> tag because those events bubble up from any element that the user tries to drag. The scripts invoked by these event handlers filter the events so that the desired action is triggered only by the "hot" elements inside the table. This approach to event handlers prevents you from having to duplicate event handlers (or IE <script for=> tags) for each table cell.

The ondragstart event handler invokes setupDrag(). This function cancels the ondragstart event except when the target element (the one about to be dragged) is one of the td elements inside the table. To make this application smarter about what kind of word is dragged to which blank, it passes not only the word's text, but also some extra information about the word. This lets another event handler verify that a noun has been dragged to the first blank, while an adjective has been dragged to the second blank. To help with this effort, class names are assigned to the td elements to distinguish the words from the Nouns column from the words of the Adjectives column. The setupDrag() function generates an array consisting of the innerText of the event's source element plus the element's class name. But the event.dataTransfer object cannot store array data types, so the Array.join() method converts the array to a string with a colon separating the entries. This string, then, is stuffed into the event.dataTransfer object. The object is instructed to render the cursor display during the drag-and-drop operation so that when the cursor is atop a drop target, the cursor

is the "copy" style. Finally, the `setupDrag()` function is the first to execute in the drag operation, so a timer is set to the current clock time to time the drag operation.

The `ondrag` event handler (in the `body`) captures the `ondrag` events that are generated by whichever table cell element is the source element for the action. Each time the event fires (which is a lot during the action), the `timeIt()` function is invoked to compare the current time against the reference time (global `timer`) set when the drag starts. If the time exceeds two seconds (2,000 milliseconds), an alert dialog box notifies the user. To close the alert dialog box, the user must unclick the mouse button to end the drag operation.

To turn the blank `span` elements into drop targets, their `ondragenter`, `ondragover`, and `ondrop` event handlers must set `event.returnValue` to `false`; also, the `event.dataTransfer.dropEffect` property should be set to the desired effect (`copy` in this case). These event handlers are placed in the `p` element that contains the two `span` elements, again for simplicity. Notice, however, that the `cancelDefault()` functions do their work only if the target element is one of the `span` elements whose ID begins with "blank."

As the user releases the mouse button, the `ondrop` event handler invokes the `handleDrop()` function. This function retrieves the string data from `event.dataTransfer` and restores it to an array data type (using the `String.split()` method). A little bit of testing makes sure that the word type ("noun" or "adjective") is associated with the desired blank. If so, the source element's text is set to the drop target's `innerText` property; otherwise, an error message is assembled to help the user know what went wrong.

Listing 15-37: **Using Drag-Related Event Handlers**

```
<html>
   <head>
      <title>Dragging Event Handlers</title>
      <style type="text/css">
      td {text-align:center}
      th {text-decoration:underline}
      .blanks {text-decoration:underline}
      </style>
      <script type="text/javascript">
      var timer;

      function setupDrag() {
         if (event.srcElement.tagName != "TD") {
            // don't allow dragging for any other elements
            event.returnValue = false;
         } else {
            // setup array of data to be passed to drop target
            var passedData = [event.srcElement.innerText,
               event.srcElement.className];
            // store it as a string
            event.dataTransfer.setData("Text", passedData.join(":"));
            event.dataTransfer.effectAllowed = "copy";
            timer = new Date();
         }
      }

      function timeIt() {
         if (event.srcElement.tagName == "TD" && timer) {
            if ((new Date()) - timer > 2000) {
               alert("Sorry, time is up. Try again.");
```

```
            timer = 0;
         }
      }
   }

   function handleDrop() {
      var elem = event.srcElement;
      var passedData = event.dataTransfer.getData("Text");
      var errMsg = "";
      if (passedData) {
         // reconvert passed string to an array
         passedData = passedData.split(":");
         if (elem.id == "blank1") {
            if (passedData[1] == "noun") {
               event.dataTransfer.dropEffect = "copy";
               event.srcElement.innerText = passedData[0];
            } else {
               errMsg = "You can't put an adjective into the noun
                  placeholder.";
            }
         } else if (elem.id == "blank2") {
            if (passedData[1] == "adjective") {
               event.dataTransfer.dropEffect = "copy";
               event.srcElement.innerText = passedData[0];
            } else {
               errMsg = "You can't put a noun into the adjective
                  placeholder.";
            }
         }
         if (errMsg) {
            alert(errMsg);
         }
      }
   }

   function cancelDefault() {
      if (event.srcElement.id.indexOf("blank") == 0) {
         event.dataTransfer.dropEffect = "copy";
         event.returnValue = false;
      }
   }
   </script>
</head>
<body ondragstart="setupDrag()" ondrag="timeIt()">
   <h1>Dragging Event Handlers</h1>
   <hr />
   <p>Your goal is to drag one noun and one adjective from the following
      table into the blanks of the sentence. Select a word from the table
      and drag it to the desired blank. When you release the mouse, the word
      will appear in the blank.You have two seconds to complete each
      blank.</p>
   <table cellpadding="5">
      <tr>
         <th>Nouns</th>
         <th>Adjectives</th>
      </tr>
      <tr>
         <td class="noun">truck</td>
```

Continued

Listing 15-37 *(continued)*

```
            <td class="adjective">round</td>
         </tr>
         <tr>
            <td class="noun">doll</td>
            <td class="adjective">red</td>
         </tr>
         <tr>
            <td class="noun">ball</td>
            <td class="adjective">pretty</td>
         </tr>
      </table>
      <p id="myP" ondragenter="cancelDefault()" ondragover="cancelDefault()"
         ondrop="handleDrop()">Pat said, "Oh my, the <span id="blank1"
         class="blanks">     </span> is so <span
         id="blank2" class="blanks">     </span>!"</p>
       <button onclick="location.reload()">Reset</button>
   </body>
</html>
```

One event handler not shown in Listing 15-37 is `ondragend`. You can use this event to display the elapsed time for each successful drag operation. Because the event fires on the drag source element, you can implement it in the `<body>` tag and filter events similar to the way the `ondragstart` or `ondrag` event handlers filter events for the `td` element.

Related Items: `event.dataTransfer` object; `ondragenter`, `ondragleave`, `ondragover`, `ondrop` event handlers.

ondragenter
ondragleave
ondragover

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

These events fire during a drag operation. When the cursor enters the rectangular space of an element on the page, the `ondragenter` event fires on that element. Immediately thereafter, the `ondragleave` event fires on the element from which the cursor came. While this may seem to occur out of sequence from the physical action, the events always fire in this order. Depending on the speed of the client computer's CPU and the speed of the user's dragging action, one or the other of these events may not fire — especially if the physical action outstrips the computer's capability to fire the events in time.

The `ondragover` event fires continually while a dragged cursor is atop an element. In the course of dragging from one point on the page to another, the `ondragover` event target changes with the element beneath the cursor. If no other drag-related events are firing (the mouse button is still down in the drag operation, but the cursor is not moving), the `ondrag` and `ondragover` events fire continually, alternating between the two.

You should have the `ondragover` event handler of a drop target element set the `event.returnValue` property to `false`. See the discussion of the `ondrag` event handler earlier in this chapter for more details on the sequence of drag-related events.

Example

Listing 15-38 shows the ondragenter and ondragleave event handlers in use. The simple page displays (via the status bar) the time of entry to one element of the page. When the dragged cursor leaves the element, the ondragleave event handler hides the status bar message. No drop target is defined for this page, so when you drag the item, the cursor remains as the "no drop" cursor.

Listing 15-38: Using ondragenter and ondragleave Event Handlers

```
<html>
   <head>
      <title>ondragenter and ondragleave Event Handlers</title>
      <script type="text/javascript">
      function showEnter() {
         status = "Entered at: " + new Date();
         event.returnValue = false;
      }
      function clearMsg() {
         status = "";
         event.returnValue = false;
      }
      </script>
   </head>
   <body>
      <h1 ondragenter="showEnter()" ondragleave="clearMsg()">
      ondragenter and ondragleave Event Handlers</h1>
      <hr />
      <p>Select any character(s) from this paragraph, and slowly drag it around
         the page. When the dragging action enters the large header above, the
         status bar displays when the onDragEnter event handler fires. When you
         leave the header, the message is cleared via the onDragLeave event
         handler.</p>
   </body>
</html>
```

Related Items: ondrag, ondragend, ondragstart, ondrop event handlers.

ondragstart

(See ondrag)

ondrop

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The ondrop event fires on the drop target element as soon as the user releases the mouse button at the end of a drag-and-drop operation. Microsoft recommends that you denote a drop target by applying the ondragenter, ondragover, and ondrop event handlers to the target element. In each of those event handlers, you should set the dataTransfer.dropEffect to the transfer effect you wish to portray in the drag-and-drop operation (signified by a different cursor for each type). These settings should match the dataTransfer.effectAllowed property that is usually set in the ondragstart event handler. Each of the three drop-related handlers should also override the default event behavior by setting the event.returnValue property to

`false`. See the discussion of the `ondrag` event handler earlier in this chapter for more details on the sequence of drag-related events.

Example

See Listing 15-37 of the `ondrag` event handler to see how to apply the `ondrop` event handler in a typical drag-and-drop scenario.

Related Items: `event.dataTransfer` object; `ondrag`, `ondragend`, `ondragenter`, `ondragleave`, `ondragover`, `ondragstart` event handlers.

onfilterchange

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The `onfilterchange` event fires whenever an object's visual filter switches to a new state or a transition completes (a transition may be extended over time). Only objects that accommodate filters and transitions in IE (primarily block elements and form controls) receive the event.

A common usage of the `onfilterchange` event is to trigger the next transition within a sequence of transition activities. This may include an infinite loop transition, for which the object receiving the event toggles between two transition states. If you don't want to get into a loop of that kind, place the different sets of content into their own positionable elements and use the `onfilterchange` event handler in one to trigger the transition in the other.

Example

Listing 15-39 demonstrates how the `onfilterchange` event handler can trigger a second transition effect after another one completes. The `onload` event handler triggers the first effect. Although the `onfilterchange` event handler works with most of the same objects in IE4 as IE5, the filter object transition properties are not reflected in a convenient form. The syntax shown in Listing 15-39 uses the new ActiveX filter control found in IE5.5+ (described in Chapter 30).

Listing 15-39: Using the onFilterChange Event Handler

```
<html>
   <head>
      <title>onfilterchange Event Handler</title>
      <script type="text/javascript">
      function init() {
         image1.filters[0].apply();
         image2.filters[0].apply();
         start();
      }

      function start() {
         image1.style.visibility = "hidden";
         image1.filters[0].play();
      }

      function finish() {
         // verify that first transition is done (optional)
         if (image1.filters[0].status == 0) {
            image2.style.visibility = "visible";
            image2.filters[0].play();
         }
```

```
      }
      </script>
   </head>
   <body onload="init()">
      <h1>onfilterchange Event Handler</h1>
      <hr />
      <p>The completion of the first transition ("circle-in") triggers the
         second ("circle-out"). <button onclick="location.reload()">Play It
         Again</button></p>
      <div id="image1"
      style="visibility:visible; position:absolute; top:150px; left:150px;
      filter:progID:DXImageTransform.Microsoft.Iris(irisstyle='CIRCLE',
      motion='in')" onfilterchange="finish()">
         <img alt="image" src="desk1.gif" height="90" width="120" />
      </div>
      <div id="image2"
      style="visibility:hidden; position:absolute; top:150px; left:150px;
      filter:progID:DXImageTransform.Microsoft.Iris(irisstyle='CIRCLE',
      motion='out')">
         <img alt="image" src="desk3.gif" height="90" width="120" />
      </div>
   </body>
</html>
```

Related Item: `filter` object.

onfocus

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `onfocus` event fires when an element receives focus, usually following some other object losing focus. (The element losing focus receives the `onblur` event before the current object receives the `onfocus` event.) For example, a text input element fires the `onfocus` event when a user tabs to that element while navigating through a form via the keyboard. Clicking an element also gives that element focus, as does making the browser the frontmost application on the client desktop.

The availability of the `onfocus` event has expanded with succeeding generations of script-capable browsers. In earlier versions, blur and focus were largely confined to text-oriented input elements such as the `select` element. The `window` object received the `onfocus` event handler starting with NN3 and IE4. IE4 also extended the event handler to more form elements, predominantly on the Windows operating system because that OS has a user interface clue (the dotted rectangle) when items such as buttons and links receive focus (so that users may act upon them by pressing the keyboard's spacebar). For IE5+, the `onfocus` event handler is available to virtually every HTML element. For most of those elements, however, you cannot use blur and focus unless you assign a value to the `tabindex` attribute of the element's tag. For example, if you assign `tabindex="1"` inside a `<p>` tag, the user can bring focus to that paragraph (highlighted with the dotted rectangle in Windows) by clicking the paragraph or pressing the Tab key until that item receives focus in sequence.

WinIE5.5 adds the `onactivate` event handler, which fires immediately before the `onfocus` event handler. You can use one or the other, but there is little need to include both event handlers for the same object unless you temporarily wish to block an item from receiving focus. To prevent an object from receiving focus in IE5.5+, include an `event.returnValue=false` statement in the `onactivate` event handler for the same object. In other browsers, you can usually get away with assigning `onfocus="this.blur()"` as an event handler for elements such as

form controls. However, this is not a foolproof way to prevent a user from changing a control's setting. Unfortunately, there are few reliable alternatives, short of disabling the control.

Example

See Listing 15-34 earlier in this chapter for an example of the `onfocus` and `onblur` event handlers.

Related Items: `onactivate`, `onblur`, `ondeactivate` event handlers.

onhelp

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `onhelp` event handler fires in Windows whenever an element of the document has focus and the user presses the F1 function key on a Windows PC. As of MacIE5, the event fires only on the window (in other words, event handler specified in the `<body>` tag) and does so via the dedicated Help key on a Mac keyboard. Browser Help menu choices do not activate this event. To prevent the browser's Help window from appearing, the event handler must evaluate to `return false` (for IE4+) or set the `event.returnValue` property to `false` (IE5+). Because the event handler can be associated with individual elements of a document in the Windows version, you can create a context-sensitive help system. However, if the focus is in the Address field of the browser window, you cannot intercept the event. Instead, the browser's Help window appears.

Example

Listing 15-40 is a rudimentary example of a context-sensitive help system that displays help messages tailored to the kind of text input required by different text fields. When the user gives focus to either of the text fields, a small legend appears to remind the user that help is available by a press of the F1 help key. MacIE5 provides only generic help.

Listing 15-40: Creating Context-Sensitive Help

```
<html>
   <head>
      <title>onhelp Event Handler</title>
      <script type="text/javascript">
      function showNameHelp() {
         alert("Enter your first and last names.");
         event.cancelBubble = true;
         return false;
      }
      function showYOBHelp() {
         alert("Enter the four-digit year of your birth. For example: 1972");
         event.cancelBubble = true;
         return false;
      }
      function showGenericHelp() {
         alert("All fields are required.");
         event.cancelBubble = true;
         return false;
      }
      function showLegend() {
         document.getElementById("legend").style.visibility = "visible";
      }
      function hideLegend() {
```

```
            document.getElementById("legend").style.visibility = "hidden";
         }
         function init() {
            var msg = "";
            if (navigator.userAgent.indexOf("Mac") != -1) {
               msg = "Press \'help\' key for help.";
            } else if (navigator.userAgent.indexOf("Win") != -1) {
               msg = "Press F1 for help.";
            }
            document.getElementById("legend").style.visibility = "hidden";
            document.getElementById("legend").innerHTML = msg;
         }
         </script>
      </head>
      <body onload="init()" onhelp="return showGenericHelp()">
         <h1>onhelp Event Handler</h1>
         <hr />
         <p id="legend" style="visibility:hidden; font-size:10px"> </p>
         <form>
            Name: <input type="text" name="name" size="30" onfocus="showLegend()"
            onblur="hideLegend()" onhelp="return showNameHelp()" /><br />
             Year of Birth: <input type="text" name="YOB" size="30"
            onfocus="showLegend()" onblur="hideLegend()"
            onhelp="return showYOBHelp()" />
         </form>
      </body>
   </html>
```

Related Items: `window.showHelp()`, `window.showModalDialog()` methods.

onkeydown
onkeypress
onkeyup

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

When someone presses and releases a keyboard key, a sequence of three events fires in quick succession. The `onkeydown` event fires when the key makes its first contact. This is followed immediately by the `onkeypress` event. When contact is broken by the key release, the `onkeyup` event fires. If you hold a character key down until it begins auto-repeating, the `onkeydown` and `onkeypress` events fire with each repetition of the character.

The sequence of events can be crucial in some keyboard event handling. Consider the scenario that wants the focus of a series of text fields to advance automatically after the user enters a fixed number of characters (for example, date, month, and two-digit year). By the time the `onkeyup` event fires, the character associated with the key press action is already added to the field and you can accurately determine the length of text in the field, as shown in this simple example:

```
<html>
<head>
<script type="text/javascript">
function jumpNext(fromFld, toFld) {
   if (fromFld.value.length == 2) {
      document.forms[0].elements[toFld].focus();
      document.forms[0].elements[toFld].select();
   }
```

```
}
</script>
</head>
<body>
<form>
Month: <input name="month" type="text" size="3" value=""
   onkeyup="jumpNext(this, day)" maxlength="2" />
Day: <input name ="day" type="text" size="3" value=""
   onkeyup ="jumpNext(this, year)" maxlength="2" />
Year: <input name="year" type="text" size="3" value=""
   onkeyup ="jumpNext(this, month)" maxlength="2" />
</form>
</body>
</html>
```

These three events do not fire for all keys of the typical PC keyboard on all browser versions that support keyboard events. The only keys that you can rely on supporting the events in all browsers shown in the preceding compatibility chart are the alphanumeric keys represented by ASCII values. This includes keys such as the spacebar and Enter (Return on the Mac), but it excludes all function keys, arrow keys, and other navigation keys. Modifier keys, such as Shift, Ctrl (PC), Alt (PC), Command (Mac), and Option (Mac), generate some events on their own (depending on browser and version). However, functions invoked by other key events can always inspect the pressed states of these modifier keys.

The `onkeydown` event handler works in Mozilla-based browsers only starting with Mozilla 1.4 (and Netscape 7.1).

Scripting keyboard events almost always entails examining which key is pressed so that some processing or validation can be performed on that key press. This is where the situation gets very complex if you are writing for cross-browser implementation. In some cases, even writing just for Internet Explorer gets tricky because non-alphanumeric keys generate only the `onkeydown` and `onkeyup` events.

In fact, to fully comprehend keyboard events, you need to make a distinction between *key codes* and *character codes*. Every PC keyboard key has a key code associated with it. This key code is always the same regardless of what other keys you press at the same time. Only the alphanumeric keys (letters, numbers, spacebar, and so on), however, generate character codes. The code represents the typed character produced by that key. The value might change if you press a modifier key. For example, if you type the "A" key by itself, it generates a lowercase "a" character (character code 97); if you also hold down the Shift key, that same key produces an uppercase "A" character (character code 65). The key code for that key (65 for Western language keyboards) remains the same no matter what.

That brings us, then, to where these different codes are made available to scripts. In all cases, the code information is conveyed as one or two properties of the browser's `event` object. IE's `event` object has only one such property — `keyCode`. It contains key codes for `onkeydown` and `onkeyup` events, but character codes for `onkeypress` events. The NN6/Moz1 event object, on the other hand, contains two separate properties: `charCode` and `keyCode`. You can find more details and examples about these `event` object properties in Chapter 25.

The bottom-line script consideration is to use either `onkeydown` or `onkeyup` event handlers when you want to look for non-alphanumeric key events (for example, function keys, arrow and page navigation keys, and so on). To process characters as they appear in text boxes, use the `onkeypress` event handler. You can experiment with these events and codes in Listing 15-41 as well as in examples from Chapter 25.

Common keyboard event tasks

WinIE4+ enables you to modify the character that a user who is editing a text box enters. The `onkeypress` event handler can modify the `event.keyCode` property and allow the event to continue (in other words, don't evaluate to `return false` or set the `event.returnValue` property to `false`). The following IE function (invoked by an `onkeypress` event handler) makes sure text entered into a text field is all uppercase, even if you type it as lowercase:

```
function assureUpper() {
   if (event.keyCode >= 97 && event.keyCode <= 122) {
      event.keyCode = event.keyCode - 32;
   }
}
```

Doing this might confuse (or frustrate) users, so think carefully before implementing such a plan.

To prevent a keyboard key press from becoming a typed character in a text field, the `onkeypress` event handler prevents the default action of the event. For example, the following HTML page shows how to inspect a text field's entry for numbers only:

```
<html>
<head>
<title>Keyboard Capture</title>
<script type="text/javascript">
function checkIt(evt) {
   var charCode = (evt.charCode) ? evt.charCode : ((
      evt.which) ? evt.which : evt.keyCode);
   if (charCode > 31 && (charCode < 48 || charCode > 57)) {
      alert("Please make sure entries are numbers only.");
      return false;
   }
   return true;
}
</script>
</head>

<body>
<form>
Enter any positive integer: <input type="text" name="numeric"
   onkeypress="return checkIt(event)">
</form>
</body>
</html>
```

Whenever a user enters a non-number, the user receives a warning and the character is not appended to the text box's text.

Keyboard events also enable you to script the submission of a form when a user presses the Enter (Return on the Mac) key within a text box. The ASCII value of the Enter/Return key is 13. Therefore, you can examine each key press in a text box and submit the form whenever value 13 arrives, as shown in the following function:

```
function checkForEnter(evt) {
   evt = (evt) ? evt : event;
   var charCode = (evt.charCode) ? evt.charCode : ((
      evt.which) ? evt.which : evt.keyCode);
   if (charCode == 13) {
      document.forms[0].submit();
      return false;
```

```
    }
    return true;
}
```

By assigning the `checkForEnter()` function to each field's `onkeypress` event handler, you suddenly add some extra power to a typical HTML form.

You can intercept Ctrl+keyboard combinations (letters only) in HTML pages most effectively in Internet Explorer, but only if the browser itself does not use the combination. In other words, you cannot redirect Ctrl+key combinations that the browser uses for its own control. The `onkeypress` `keyCode` value for Ctrl+combinations ranges from 1 through 26 for letters A through Z (except for those used by the browser, in which case no keyboard event fires).

Example

Listing 15-41 is a working laboratory that you can use to better understand the way keyboard event codes and modifier keys work in IE5+ and W3C browsers. The actual code of the listing is less important than watching the page while you use it. For every key or key combination that you press, the page shows the `keyCode` value for the `onkeydown`, `onkeypress`, and `onkeyup` events. If you hold down one or more modifier keys while performing the key press, the modifier key name is highlighted for each of the three events. Note that when run in NN6+, the `keyCode` value is not the character code (which doesn't show up in this example for NN6+). Also, you may need to click the NN6+ page for the `document` object to recognize the keyboard events.

The best way to watch what goes on during keyboard events is to press and hold a key to see the key codes for the `onkeydown` and `onkeypress` events. Then release the key to see the code for the `onkeyup` event. Notice, for instance, that if you press the A key without any modifier key, the `onkeydown` event key code is 65 (A) but the `onkeypress` key code in IE (and the `charCode` property in NN6+) is 97 (a). If you then repeat the exercise but hold the Shift key down, all three events generate the 65 (A) key code (and the Shift modifier labels are highlighted). Releasing the Shift key causes the `onkeyup` event to show the key code for the Shift key.

In another experiment, press any of the four arrow keys. No key code is passed for the `onkeypress` event because those keys don't generate those events. They do, however, generate `onkeydown` and `onkeyup` events.

Listing 15-41: **Keyboard Event Handler Laboratory**

```
<html>
   <head>
      <title>Keyboard Event Handler Lab</title>
      <style type="text/css">
      td {text-align:center}
      </style>
      <script type="text/javascript">
      function init() {
         document.onkeydown = showKeyDown;
         document.onkeyup = showKeyUp;
         document.onkeypress = showKeyPress;
      }

      function showKeyDown(evt) {
         evt = (evt) ? evt : window.event;
         document.getElementById("pressKeyCode").innerHTML = 0;
```

```
   document.getElementById("upKeyCode").innerHTML = 0;
   document.getElementById("pressCharCode").innerHTML = 0;
   document.getElementById("upCharCode").innerHTML = 0;
   restoreModifiers("");
   restoreModifiers("Down");
   restoreModifiers("Up");
   document.getElementById("downKeyCode").innerHTML = evt.keyCode;
   if (evt.charCode) {
      document.getElementById("downCharCode").innerHTML = evt.charCode;
   }
   showModifiers("Down", evt);
}

function showKeyUp(evt) {
   evt = (evt) ? evt : window.event;
   document.getElementById("upKeyCode").innerHTML = evt.keyCode;
   if (evt.charCode) {
      document.getElementById("upCharCode").innerHTML = evt.charCode;
   }
   showModifiers("Up", evt);
   return false;
}

function showKeyPress(evt) {
   evt = (evt) ? evt : window.event;
   document.getElementById("pressKeyCode").innerHTML = evt.keyCode;
   if (evt.charCode) {
      document.getElementById("pressCharCode").innerHTML = evt.charCode;
   }
   showModifiers("", evt);
   return false;
}

function showModifiers(ext, evt) {
   restoreModifiers(ext);
   if (evt.shiftKey) {
      document.getElementById("shift" + ext).style.backgroundColor =
         "#ff0000";
   }
   if (evt.ctrlKey) {
      document.getElementById("ctrl" + ext).style.backgroundColor =
         "#00ff00";
   }
   if (evt.altKey) {
      document.getElementById("alt" + ext).style.backgroundColor =
         "#0000ff";
   }
}

function restoreModifiers(ext) {
   document.getElementById("shift" + ext).style.backgroundColor =
      "#ffffff";
   document.getElementById("ctrl" + ext).style.backgroundColor =
      "#ffffff";
   document.getElementById("alt" + ext).style.backgroundColor =
      "#ffffff";
}
```

Continued

Listing 15-41 *(continued)*

```
      </script>
   </head>
   <body onload="init()">
      <h1>Keyboard Event Handler Lab</h1>
      <hr />
      <form>
         <table border="2" cellpadding="2">
            <tr>
               <th></th>
               <th>onKeyDown</th>
               <th>onKeyPress</th>
               <th>onKeyUp</th>
            </tr>
            <tr>
               <th>Key Codes</th>
               <td id="downKeyCode">0</td>
               <td id="pressKeyCode">0</td>
               <td id="upKeyCode">0</td>
            </tr>
            <tr>
               <th>Char Codes (IE5/Mac; NN6)</th>
               <td id="downCharCode">0</td>
               <td id="pressCharCode">0</td>
               <td id="upCharCode">0</td>
            </tr>
            <tr>
               <th rowspan="3">Modifier Keys</th>
               <td><span id="shiftDown">Shift</span></td>
               <td><span id="shift">Shift</span></td>
               <td><span id="shiftUp">Shift</span></td>
            </tr>
            <tr>
               <td><span id="ctrlDown">Ctrl</span></td>
               <td><span id="ctrl">Ctrl</span></td>
               <td><span id="ctrlUp">Ctrl</span></td>
            </tr>
            <tr>
               <td><span id="altDown">Alt</span></td>
               <td><span id="alt">Alt</span></td>
               <td><span id="altUp">Alt</span></td>
            </tr>
         </table>
      </form>
   </body>
</html>
```

Spend some time with this lab, and try all kinds of keys and key combinations until you understand the way the events and key codes work.

Related Item: `String.fromCharCode()` method.

onlosecapture

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The onlosecapture event handler fires whenever an object that has event capture turned on no longer has that capture. Event capture is automatically disengaged when the user performs any of the following actions:

- ✦ Gives focus to any other window
- ✦ Displays any system modal dialog box (for example, alert window)
- ✦ Scrolls the page
- ✦ Opens a browser context menu (right-clicking)
- ✦ Tabs to give focus to the Address field in the browser window

A function associated with the onlosecapture event handler should perform any cleanup of the environment due to an object no longer capturing mouse events.

Example

See Listing 15-30 earlier in this chapter for an example of how to use onlosecapture with an event-capturing scenario for displaying a context menu. The onlosecapture event handler hides the context menu when the user performs any action that causes the menu to lose mouse capture.

Related Items: releaseCapture(), setCapture() methods.

onmousedown
onmouseup

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The onmousedown event handler fires when the user presses any button of a mouse. The onmouseup event handler fires when the user releases the mouse button, provided the object receiving the event also received an onmousedown event. When a user performs a typical click of the mouse button atop an object, mouse events occur in the following sequence: onmousedown, onmouseup, onclick. But if the user presses the mouse atop an object and then slides the cursor away from the object, only the onmousedown event fires.

These events enable authors and designers to add more application-like behavior to images that act as action or icon buttons. If you notice the way most buttons work, the appearance of the button changes while you press the mouse button and reverts to its original style when you release the mouse button (or you drag the cursor out of the button). These events enable you to emulate that behavior.

The event object created with every mouse button action has a property that reveals which mouse button the user pressed. NN4's event model calls that property the which property. IE4+ and NN6+/Moz1+ call it the button property (but with different values for the buttons). It is most reliable to test for the mouse button number on either the onmousedown or onmouseup event, rather than on onclick. The onclick event object does not always contain the button information.

Example

To demonstrate a likely scenario of changing button images in response to rolling atop an image, pressing down on it, releasing the mouse button, and rolling away from the image, Listing 15-42 presents a pair of small navigation buttons (left- and right-arrow buttons). Because the image object is not part of the document object model for NN2 or IE3 (which reports itself as Navigator version 2), the page is designed to accept all browsers. Only those browsers that support precached images and image swapping (and thus pass the test for the presence of the

`document.images` array) can execute those statements. For a browser with an image object, images are preloaded into the browser cache as the page loads so that response to the user is instantaneous the first time the user calls upon new versions of the images.

Listing 15-42: **Using onmousedown and onmouseup Event Handlers**

```
<html>
   <head>
      <title>onmousedown and onmouseup Event Handlers</title>
      <script type="text/javascript">
      if (document.images) {
         var RightNormImg = new Image(16,16);
         var RightUpImg = new Image(16,16);
         var RightDownImg = new Image(16,16);
         var LeftNormImg = new Image(16,16);
         var LeftUpImg = new Image(16,16);
         var LeftDownImg = new Image(16,16);

         RightNormImg.src = "RightNorm.gif";
         RightUpImg.src = "RightUp.gif";
         RightDownImg.src = "RightDown.gif";
         LeftNormImg.src = "LeftNorm.gif";
         LeftUpImg.src = "LeftUp.gif";
         LeftDownImg.src = "LeftDown.gif";
      }

      function setImage(imgName, type) {
         if (document.images) {
            var imgFile = eval(imgName + type + "Img.src");
            document.images[imgName].src = imgFile;
            return false;
         }
      }
      </script>
   </head>
   <body>
      <h1>onmousedown and onmouseup Event Handlers</h1>
      <hr />
      <p>Roll atop and click on the buttons to see how the link event handlers
         swap images:</p>
      <center>
         <a href="javascript:void(0)"
         onmouseover="return setImage('Left','Up')"
         onmousedown="return setImage('Left','Down')"
         onmouseup="return setImage('Left','Up')"
         onmouseout="return setImage('Left','Norm')"><img alt="image"
         name="Left" src="LeftNorm.gif" height="16" width="16"
         border="0" /></a>    <a href="javascript:void(0)"
         onmouseover="return setImage('Right','Up')"
         onmousedown="return setImage('Right','Down')"
         onmouseup="return setImage('Right','Up')"
         onmouseout="return setImage('Right','Norm')"><img alt="image"
         name="Right" src="RightNorm.gif" height="16" width="16"
         border="0" /></a>
      </center>
   </body>
</html>
```

IE4+ and W3C browsers simplify the implementation of this kind of three-state image button by allowing you to assign the event handlers directly to `img` element objects. Wrapping images inside links is a backward-compatibility approach that allows older browsers to respond to clicks on images for navigation or other scripting tasks.

Related Item: `onclick` event handler.

onmouseenter
onmouseleave

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

WinIE5.5 introduced the `onmouseenter` and `onmouseleave` event handlers. Both event handlers operate just like the `onmouseover` and `onmouseout` event handlers, respectively. Microsoft simply offers an alternate terminology. The old and new events continue to fire in IE5.5+. The old ones fire just before the new ones for each act of moving the cursor atop, and exiting from atop, the object. If you are scripting exclusively for IE5.5+, you should use the new terminology; otherwise, stay with the older versions.

Example

You can modify Listing 15-43 with the IE5.5 syntax by substituting `onmouseenter` for `onmouseover` and `onmouseleave` for `onmouseout`. The effect is the same.

Related Items: `onmouseover`, `onmouseout` event handlers.

onmousemove

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The `onmousemove` event handler fires whenever the cursor is atop the current object and the mouse is moved, even by a single pixel. You do not have to press the mouse button for the event to fire, although the event is most commonly used in element dragging — especially in NN/Mozilla, where no `ondrag` event handler is available.

Even though the granularity of this event can be at the pixel level, you should not use the number of event firings as a measurement device. Depending on the speed of cursor motion and the performance of the client computer, the event may not fire at every pixel location.

In NN4, you cannot assign the `onmousemove` event handler to any object by way of tag attributes. But you can use the NN4 event capturing mechanism to instruct (via scripting) a `window`, `document`, or `layer` object to capture `mouseMove` events. This allows for NN4 scripts to produce positioned element (layer) dragging. In IE4+ and W3C DOM-compatible browsers, however, you can assign the `onmousemove` event handler to any element (although you can drag only with positioned elements). When designing a page that encourages users to drag multiple items on a page, it is most common to assign the `onmousemove` event handler to the `document` object and let all such events bubble up to the document for processing.

Example

See Chapters 39 and 56 on the CD-ROM for examples of using mouse events to control element dragging on a page.

Related Items: `ondrag`, `onmousedown`, `onmouseup` event handlers.

***elementObject*.onmouseout**

onmouseout
onmouseover

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The onmouseover event fires for an object whenever the cursor rolls into the rectangular space of the object on the screen. The onmouseout event handler fires when you move the cursor outside the object's rectangle. These events most commonly display explanatory text about an object in the window's status bar and effect image swapping (so-called mouse rollovers). Use the onmouseover event handler to change the state to a highlighted version; use the onmouseout event handler to restore the image or status bar to its normal setting.

While these two events have been in object models of scriptable browsers since the beginning, they were not available to most objects in earlier browsers. IE4+ and W3C DOM-compatible browsers provide support for these events on every element that occupies space on the screen. IE5.5+ includes an additional pair of event handlers — onmouseenter and onmouseleave — that duplicate the onmouseover and onmouseout events but with different terminology. The old event handlers fire just before the new versions.

The onmouseout event handler commonly fails to fire if the event is associated with an element that is near a frame or window edge and the user moves the cursor quickly outside of the current frame.

Example

Listing 15-43 uses the U.S. Pledge of Allegiance with four links to demonstrate how to use the onmouseover and onmouseout event handlers. Notice that for each link, the handler runs a general-purpose function that sets the window's status message. The function returns a true value, which the event handler call evaluates to replicate the required return true statement needed for setting the status bar. In one status message, I supply a URL in parentheses to let you evaluate how helpful you think it is for users.

Listing 15-43: **Using onmouseover and onmouseout Event Handlers**

```
<html>
   <head>
      <title>onmouseover and onmouseout Event Handlers</title>
      <script type="text/javascript">
      function setStatus(msg) {
         status = msg;
         return true;
      }

      // destination of all link HREFs
      function emulate() {
         alert("Not going there in this demo.");
      }
      </script>
   </head>
   <body>
      <h1>onmouseover and onmouseout Event Handlers</h1>
      <hr />
      <h1>Pledge of Allegiance</h1>
      <hr />
      I pledge <a href="javascript:emulate()"
```

```
      onmouseover="return setStatus('View dictionary definition')"
      onmouseout="return setStatus('')">allegiance</a> to the <a
      href="javascript:emulate()"
      onmouseover="return setStatus('Learn about the U.S. flag
      (http://lcweb.loc.gov)')"
       onmouseout="return setStatus('')">flag</a> of the <a
      href="javascript:emulate()"
      onmouseover="return setStatus('View info about the U.S. government')"
      onmouseout="return setStatus('')">United States of America</a>, and to
      the Republic for which it stands, one nation <a
      href="javascript:emulate()"
      onmouseover="return setStatus('Read about the history of this phrase in
      the Pledge')"
       onmouseout="return setStatus('')">under God</a>, indivisible, with
      liberty and justice for all.
   </body>
</html>
```

Related Items: `onmouseenter`, `onmouseleave`, `onmousemove` event handlers.

onpaste

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `onpaste` event fires immediately after the user or script initiates a paste edit action on the current object. The event is preceded by the `onbeforepaste` event, which fires prior to any edit or context menu that appears (or before the paste action if initiated by keyboard shortcut).

Use this event handler to provide edit functionality to elements that don't normally allow pasting. In such circumstances, you need to enable the Paste menu item in the context or Edit menu by setting the `event.returnValue` for the `onbeforepaste` event handler to `false`. Then your `onpaste` event handler must manually retrieve data from the clipboard (by way of the `getData()` method of the `clipboardData` object) and handle the insertion into the current object.

Because you are in charge of what data is stored in the clipboard, you are not limited to a direct copy of the data. For example, you might wish to store the value of the `src` property of an image object so that you can paste it elsewhere on the page.

Example

Listing 15-44 demonstrates how to use the `onbeforepaste` and `onpaste` event handlers (in conjunction with `onbeforecopy` and `oncopy`) to let scripts control the data-transfer process during a copy-and-paste user operation. A table contains words to be copied (one column of nouns, one column of adjectives) and then pasted into blanks in a paragraph. The `onbeforecopy` and `oncopy` event handlers are assigned to the `table` element because the events from the `td` elements bubble up to the `table` container and there is less HTML code to contend with.

Inside the paragraph, two `span` elements contain underscored blanks. To paste text into the blanks, the user must first select at least one character of the blanks. (See Listing 15-37, which gives a drag-and-drop version of this application.) The `onbeforepaste` event handler in the paragraph (which gets the event as it bubbles up from either `span`) sets the `event.returnValue` property to `false`, thus allowing the Paste item to appear in the context and Edit menus (not a normal occurrence in HTML body content).

At paste time, the `innerHTML` property of the target `span` is set to the text data stored in the clipboard. The `event.returnValue` property is set to `false` here, as well, to prevent normal system pasting from interfering with the controlled version.

Listing 15-44: **Using onbeforepaste and onpaste Event Handlers**

```
<html>
   <head>
      <title>onbeforepaste and onpaste Event Handlers</title>
      <style type="text/css">
      td {text-align:center}
      th {text-decoration:underline}
      .blanks {text-decoration:underline}
      </style>
      <script type="text/javascript">
      function selectWhole() {
         var obj = window.event.srcElement;
         var range = document.body.createTextRange();
         range.moveToElementText(obj);
         range.select();
         event.returnValue = false;
      }
      function handleCopy() {
         var rng = document.selection.createRange();
         clipboardData.setData("Text",rng.text);
         event.returnValue = false;
      }

      function handlePaste() {
         var elem = window.event.srcElement;
         if (elem.className == "blanks") {
            elem.innerHTML = clipboardData.getData("Text");
         }
         event.returnValue = false;
      }
      function handleBeforePaste() {
         var elem = window.event.srcElement;
         if (elem.className == "blanks") {
            event.returnValue = false;
         }
      }
      </script>
   </head>
   <body>
      <h1>onbeforepaste and onpaste Event Handlers</h1>
      <hr />
      <p>Your goal is to copy and paste one noun and one adjective from the
         following table into the blanks of the sentence. Select a word from
         the table and copy it to the clipboard. Select one or more spaces of
         the blanks in the sentence and choose Paste to replace the blank with
         the clipboard contents.</p>
      <table cellpadding="5" onbeforecopy="selectWhole()"
      oncopy="handleCopy()">
         <tr>
            <th>Nouns</th>
            <th>Adjectives</th>
         </tr>
         <tr>
```

```
            <td>truck</td>
            <td>round</td>
         </tr>
         <tr>
            <td>doll</td>
            <td>red</td>
         </tr>
         <tr>
            <td>ball</td>
            <td>pretty</td>
         </tr>
      </table>
      <p id="myP" onbeforepaste="handleBeforePaste()" onpaste="handlePaste()">
         Pat said, "Oh my, the <span id="blank1"
         class="blanks">     </span> is so <span
         id="blank2" class="blanks">     </span>!"</p>
       <button onclick="location.reload()">Reset</button>
   </body>
</html>
```

Related Items: `oncopy`, `oncut`, `onbeforepaste` event handlers.

onpropertychange

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `onpropertychange` event fires in WinIE5+ whenever a script modifies an object's property. This includes changes to the properties of an object's style. Changing properties by way of the `setAttribute()` method also triggers this event.

A script can inspect the nature of the property change because the `event.propertyName` property contains the name (as a string) of the property that was just changed. In the case of a change to an object's `style` object, the `event.propertyName` value begins with `"style."` as in `style.backgroundcolor`.

You can use this event handler to localize any object-specific post-processing of changes to an object's properties. Rather than include the post-processing statements inside the function that makes the changes, you can make that function generalized (perhaps to modify properties of multiple objects).

Example

Listing 15-45 shows how you can respond programmatically to an object's properties being changed. The page generated by the listing contains four radio buttons that alter the `innerHTML` and `style.color` properties of a paragraph. The paragraph's `onpropertychange` event handler invokes the `showChange()` function, which extracts information about the event and displays the data in the status bar of the window. Notice how the property name includes `style.` when you modify the stylesheet property.

Listing 15-45: Using the onPropertyChange Property

```
<html>
   <head>
      <title>onpropertychange Event Handler</title>
```

Continued

Listing 15-45 *(continued)*

```
      <script type="text/javascript">
      function normalText() {
         myP.innerText = "This is a sample paragraph.";
      }
      function shortText() {
         myP.innerText = "Short stuff.";
      }
      function normalColor() {
         myP.style.color = "black";
      }
      function hotColor() {
         myP.style.color = "red";
      }
      function showChange() {
         var objID = event.srcElement.id;
         var propName = event.propertyName;
         var newValue = eval(objID + "." + propName);
         status = "The " + propName + " property of the " + objID;
         status += " object has changed to \"" + newValue + "\".";
      }
      </script>
   </head>
   <body>
      <h1>onpropertychange Event Handler</h1>
      <hr />
      <p id="myP" onpropertychange="showChange()">This is a sample
      paragraph.</p>
      <form>
         Text: <input type="radio" name="btn1" checked="checked"
         onclick="normalText()" />Normal <input type="radio" name="btn1"
         onclick="shortText()" />Short<br />
          Color: <input type="radio" name="btn2" checked="checked"
         onclick="normalColor()" />Black <input type="radio" name="btn2"
         onclick="hotColor()" />Red
      </form>
   </body>
</html>
```

Related Items: `style` property; `setAttribute()` method.

onreadystatechange

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `onreadystatechange` event handler fires whenever the ready state of an object changes. See details about these states in the discussion of the `readyState` property earlier in this chapter (and notice the limits for IE4). The change of state does not guarantee that an object is, in fact, ready for script statements to access its properties. Always check the `readyState` property of the object in any script that the `onreadystatechange` event handler invokes.

This event fires for objects that are capable of loading data: `applet`, `document`, `frame`, `frameset`, `iframe`, `img`, `link`, `object`, `script`, and XML objects. The event doesn't fire for other types of objects unless a Microsoft DHTML behavior is associated with the object. The `onreadystatechange` event does not bubble, nor can you cancel it.

Related Item: readyState property.

onresize

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The onresize event handler fires whenever an object is resized in response to a variety of user or scripted actions. Most elements include this event handler, provided the object has dimensional style attributes (for example, height, width, or position) assigned to it.

In IE4+ and NN6+/Moz1+, the onresize event does not bubble. Resizing the browser window or frame does not cause the window's onload event handler to fire.

Example

If you want to capture the user's resizing of the browser window (or frame), you can assign a function to the onresize event handler either via script

```
window.onresize = handleResize;
```

or by an HTML attribute of the body element:

```
<body onresize="handleResize()">
```

Related Item: window.resize() method.

onresizeend
onresizestart

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The onresizeend and onresizestart event handlers fire only on a resizable object in Windows edit mode.

Related Item: oncontrolselect event handler.

onselectstart

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The onselectstart event handler fires when a user begins to select content on the page. Selected content can be inline text, images, or text within an editable text field. If the user selects more than one object, the event fires in the first object affected by the selection.

Example

Use the page from Listing 15-46 to see how the onselectstart event handler works when a user selects across multiple elements on a page. As the user begins a selection anywhere on the page, the ID of the object receiving the event appears in the status bar. Notice that the event doesn't fire until you actually make a selection. When no other element is under the cursor, the body element fires the event.

Listing 15-46: Using the onselectstart Event Handler

```
<html>
   <head>
      <title>onselectstart Event Handler</title>
```

Continued

Listing 15-46 *(continued)*

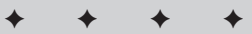
```
      <style type="text/css">
      td {text-align:center}
      </style>
      <script type="text/javascript">
      function showObj() {
         var objID = event.srcElement.id;
         status = "Selection started with object: " + objID;
      }
      </script>
   </head>
   <body id="myBody" onselectstart="showObj()">
      <h1 id="myH1">
         onselectstart Event Handler
      </h1>
      <hr id="myHR" />
      <p id="myP">This is a sample paragraph.</p>
      <table border="1">
         <tr id="row1">
            <th id="header1">Column A</th>
            <th id="header2">Column B</th>
            <th id="header3">Column C</th>
         </tr>
         <tr id="row2">
            <td id="cellA2">text</td>
            <td id="cellB2">text</td>
            <td id="cellC2">text</td>
         </tr>
         <tr id="row3">
            <td id="cellA3">text</td>
            <td id="cellB3">text</td>
            <td id="cellC3">text</td>
         </tr>
      </table>
   </body>
</html>
```

Related Item: `onselect` event handler for a variety of objects.

✦ ✦ ✦

CHAPTER 16

Window and Frame Objects



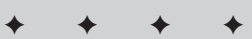
In This Chapter

Scripting communication among multiple frames

Creating and managing new windows

Controlling the size, position, and appearance of the browser window

Details of `window`, `frame`, `frameset`, and `iframe` objects



A quick look at the basic document object model diagram in Chapter 14 (see Figure 14-1) reveals that the `window` object is the outermost, most global container of all document-related objects that you script with JavaScript. All HTML and JavaScript activity takes place inside a window. That window may be a standard Windows, Mac, or XWindows application-style window, complete with scrollbars, toolbars, and other "chrome;" you can also generate windows that have only some of a typical window's chrome. A frame is also a window, even though a frame doesn't have many accoutrements beyond scroll bars. The `window` object is where everything begins in JavaScript references to objects. IE4+, NN6+, and W3C browsers treat the frameset as a special kind of `window` object, so that it is also covered in this chapter.

Of all the objects associated with browser scripting, the `window` and window-related objects have by far the most object-specific terminology associated with them. This necessitates a rather long chapter to keep the discussion in one place. Use the running footers as a navigational aid through this substantial collection of information.

Window Terminology

The `window` object is often a source of confusion when you first learn about the document object model. A number of synonyms for `window` objects muck up the works: `top`, `self`, `parent`, and `frame`. Aggravating the situation is that these terms are also properties of a `window` object. Under some conditions, a window is its own parent, but if you define a frameset with two frames, you have only one parent among a total of three `window` objects. It doesn't take long before the whole subject can make your head hurt.

If you do not use frames in your Web applications, all of these headaches never appear. But if frames are part of your design plan, you should get to know how frames affect the object model.

Frames

The application of frames has become a religious issue among Web designers: some swear by them; others swear at them. I believe there can be compelling reasons to use frames at times. For example, if you

have a document that requires considerable scrolling to get through, you may want to maintain a static set of navigation controls visible at all times. By placing those controls — be they links or image maps — in a separate frame, you have made the controls available for immediate access, regardless of the scrolled condition of the main document.

Creating frames

The task of defining frames in a document remains the same whether or not you're using JavaScript. The simplest framesetting document consists of tags that are devoted to setting up the frameset, as follows:

```
<html>
<head>
<title>My Frameset</title>
</head>
<frameset>
    <frame name="Frame1" src="document1.html">
    <frame name="Frame2" src="document2.html">
</frameset>
</html>
```

The preceding HTML document, which the user never sees, defines the frameset for the entire browser window. Each frame must have a URL reference (specified by the `src` attribute) for a document to load into that frame. For scripting purposes, assigning a name to each frame with the `name` attribute greatly simplifies scripting frame content.

The frame object model

Perhaps the key to successful frame scripting is understanding that the object model in the browser's memory at any given instant is determined by the HTML tags in the currently loaded documents. All canned object model graphics in this book, such as Figure 16-1, do not reflect the precise object model for your document or document set.

For a single, frameless document, the object model starts with just one `window` object, which contains one document, as shown in Figure 16-1. In this simple structure, the `window` object is the starting point for all references to any loaded object. Because the window is always there — it must be there for a document to load into — a reference to any object in the document can omit a reference to the current window.

In a simple two-framed frameset model (see Figure 16-2), the browser treats the container of the initial, framesetting document as the parent window. The only visible evidence that the document exists is that the framesetting document's title appears in the browser window title bar.

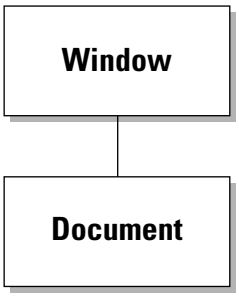


Figure 16-1: The simplest window–document relationship.

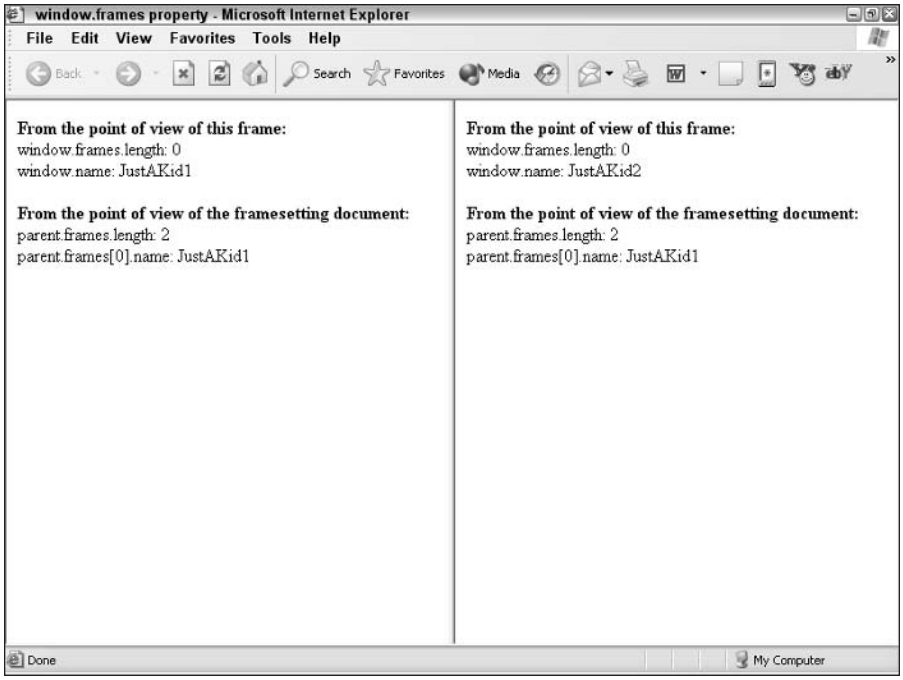


Figure 16-4: Property readouts from both frames loaded from Listing 16-7.

history

Value: Object. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

(See the discussion of the `history` object in Chapter 17.)

innerHeight
innerWidth
outerHeight
outerWidth

Value: Integer. Read/Write (see text)
Compatibility: WinIE-, MacIE-, NN4+, Moz1+, Safari1+

NN4+ lets scripts adjust the height and width of any window, including the main browser window by setting properties. This adjustment can be helpful when your page shows itself best with the browser window sized to a particular height and width. Rather than relying on the user to size the browser window for optimum viewing of your page, you can dictate the size of the window (although the user can always manually resize the main window). And because you can examine the operating system of the visitor via the `navigator` object (see Chapter 38 on the CD-ROM), you can size a window to adjust for the differences in font and form element rendering on different platforms.

Supporting browsers provide two different points of reference for measuring the height and width of a window: inner and outer. Both are measured in pixels. The inner measurements are that of the active document area of a window (sometimes known as a window's content region). If the optimum display of your document depends on the document display area being a certain number of pixels high and/or wide, the `innerHeight` and `innerWidth` properties are the ones to set.

In contrast, the outer measurements are of the outside boundary of the entire window, including whatever chrome is showing in the window: scroll bars, status bar, and so on. Setting the `outerHeight` and `outerWidth` is generally done in concert with a reading of `screen` object properties (see Chapter 38 on the CD-ROM). Perhaps the most common usage of the outer properties is to set the browser window to fill the available screen area of the visitor's monitor.

A more efficient way of modifying both outer dimensions of a window is with the `window.resizeTo()` method, which is also available in IE4+. The method takes pixel width and height (as integer values) as parameters, thus accomplishing a window resizing in one statement. Be aware that resizing a window does not adjust the location of a window. Therefore, just because you set the outer dimensions of a window to the available space returned by the `screen` object doesn't mean that the window will suddenly fill the available space on the monitor. Application of the `window.moveTo()` method is necessary to ensure the top-left corner of the window is at screen coordinates 0,0.

Despite the freedom that these properties afford the page author, Netscape and Mozilla-based browsers have built in a minimum size limitation for scripts that are not cryptographically signed. You cannot set these properties such that the outer height and width of the window is smaller than 100 pixels on a side. This limitation is to prevent an unsigned script from setting up a small or nearly invisible window that monitors activity in other windows. With signed scripts, however, windows can be made smaller than 100 × 100 pixels with the user's permission. IE4+ maintains a smaller minimum size to prevent resizing a window to zero size.

Users may dislike your scripts messing with their browser window sizes and positions. NN7+/Moz1+/Safari do not allow scripts to resize windows.

Example

In Listing 16-9, a number of buttons let you see the results of setting the `innerHeight`, `innerWidth`, `outerHeight`, and `outerWidth` properties in NN4 and NN6. Later browsers and Safari ignore scripted adjustments to these properties.

Listing 16-9: Setting Window Height and Width

```
<html>
   <head>
      <title>Window Sizer</title>
      <script type="text/javascript">
      // store original outer dimensions as page loads
      var originalWidth = window.outerWidth;
      var originalHeight = window.outerHeight;
      // generic function to set inner dimensions
```

Continued

Listing 16-9 *(continued)*

```
      function setInner(width, height) {
         window.innerWidth = width;
         window.innerHeight = height;
      }
      // generic function to set outer dimensions
      function setOuter(width, height) {
         window.outerWidth = width;
         window.outerHeight = height;
      }
      // restore window to original dimensions
      function restore() {
         window.outerWidth = originalWidth;
         window.outerHeight = originalHeight;
      }
      </script>
   </head>
   <body>
      <form>
         <b>Setting Inner Sizes</b><br />
         <input type="button" value="600 Pixels Square"
         onclick="setInner(600,600)" /><br />
         <input type="button" value="300 Pixels Square"
         onclick="setInner(300,300)" /><br />
         <input type="button" value="Available Screen Space"
         onclick="setInner(screen.availWidth, screen.availHeight)" /><br />
         <hr />
         <b>Setting Outer Sizes</b><br />
         <input type="button" value="600 Pixels Square"
         onclick="setOuter(600,600)" /><br />
         <input type="button" value="300 Pixels Square"
         onclick="setOuter(300,300)" /><br />
         <input type="button" value="Available Screen Space"
         onclick="setOuter(screen.availWidth, screen.availHeight)" /><br />
         <hr />
         <input type="button" value="Cinch up for Win95"
         onclick="setInner(273,304)" /><br />
         <input type="button" value="Cinch up for Mac"
         onclick="setInner(273,304)" /><br />
         <input type="button" value="Restore Original"
         onclick="restore()" /><br />
      </form>
   </body>
</html>
```

Related Items: `window.resizeTo()`, `window.moveTo()` methods; `screen` object; `navigator` object.

`loading`

Value: Boolean. Read-Only
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

This NN4-specific property allows you to query whether the window is still loading content. The property returns `true` if the page is still loading and `false` if the page has completed loading all of its content.

location

Value: Object. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

(See the discussion of the `location` object in Chapter 17.)

locationbar

(See `directories`)

name

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

All `window` objects can have names assigned to them. Names are particularly useful for working with frames, because a good naming scheme for a multiframe environment can help you determine precisely which frame you're working with in references coming from other frames.

The main browser window, however, has no name attached to it by default. Its value is an empty string. There aren't many reasons to assign a name to the window, because JavaScript and HTML provide plenty of other ways to refer to the `window` object (the `top` property, the `_top` constant for `target` attributes, and the `opener` property from subwindows).

If you want to attach a name to the main window, you can do so by setting the `window.name` property at any time. But be aware that because this is one window property whose life extends beyond the loading and unloading of any given document, chances are that your scripts would use the reference in only one document or frameset. Unless you restore the default empty string, your programmed window name will be present for any other document that loads later. My suggestion in this regard is to assign a name in a window's or frameset's `onload` event handler, and then reset it to empty in a corresponding `onunload` event handler:

```
<body onload="self.name = 'Main'" onunload="self.name = ''">
```

You can see an example of this application in Listing 16-16, where setting a parent window name is helpful for learning the relationships among parent and child windows.

Related Items: `top` property; `window.open()`, `window.sizeToContent()` methods.

navigator

Value: Object. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Although the `navigator` object appears as a property of the `window` object only in the most recent browsers, the `navigator` object has been around since the very beginning (see Chapter 38 on the CD-ROM). In previous browsers, the `navigator` object was referenced as a standalone object. And because you can omit any reference to the `window` object for a `window` object's properties, you can use the same window-less reference syntax for compatibility across all scriptable browsers (at least for the `navigator` object properties that exist across all browsers). That's the way I recommend referring to the `navigator` object.

Example

This book is littered with examples of using the `navigator` object, primarily for performing browser detection. You can find examples of specific `navigator` object properties in Chapter 38 on the CD-ROM.

Related Items: `navigator` object.

offscreenBuffering

Value: Boolean or string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

IE4+ by default initially renders a page in a buffer (a chunk of memory) before it is blasted to the video screen. You can control this behavior explicitly by modifying the `window.offscreenBuffering` property.

The default value of the property is the string `auto`. You can also assign Boolean `true` or `false` to the property to override IE's normal automatic handling of this behavior.

Example

If you want to turn off buffering for an entire page, include the following statement at the beginning of your script statements:

```
window.offscreenBuffering = false;
```

onerror

Value: Function. Read/Write
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari-

The `onerror` property is an exception to the rule of this book to not describe event handlers as properties within object reference sections. The reason is that the `onerror` event brings along some special properties that are useful to control by setting the event handler property in scripts.

Recent browsers (IE5+, NN4+, and W3C) are designed to prevent script errors from being intrusive if a user encounters a script error while loading or interacting with a page. Even so, even the subtle hints about problems (messages or icons in the status bar) can be confusing for users who have no idea what JavaScript is. JavaScript lets you turn off the display of script error windows or messages as someone executes a script on your page. The question is: When should you turn off these messages?

Script errors generally mean that something is wrong with your script. The error may be the result of a coding mistake or, conceivably, a bug in JavaScript (perhaps on a platform version of the browser that you haven't been able to test). If such errors occur, often the script won't continue to do what you intended. Hiding the script error from yourself during development would be foolhardy, because you'd never know whether unseen errors are lurking in your code. It can be equally dangerous to turn off error dialog boxes for users who may believe that the page is operating normally, when, in fact, it's not. Some data values may not be calculated or displayed correctly.

That said, I can see some limited instances of when you may want to keep such dialog box windows from appearing. For example, if you know for a fact that a platform-specific bug trips the error message without harming the execution of the script, you may want to prevent that error alert dialog box from appearing in the files posted to your Web site. You should do this only after extensive testing to ensure that the script ultimately behaves correctly, even with the bug or error.

IE fires the `onerror` event handler only for runtime errors. This means that if you have a syntactical error in your script that trips the browser as the page loads, the `onerror` event doesn't fire, and you cannot trap that error message. Moreover, if the user has the IE script debugger installed, any code you use to prevent browser error messages from appearing will not work.

When the browser starts, the `window.onerror` property is `<undefined>`. In this state, all errors are reported via the normal JavaScript error window or message. To turn off error alerts, set the `window.onerror` property to invoke a function that does absolutely nothing:

```
function doNothing() { return true; }
window.onerror = doNothing;
```

To restore the error messages, reload the page.

You can, however, also assign a custom function to the `window.onerror` property. This function then handles errors in a more friendly way under your script control. Whenever error messages are turned on (the default behavior), a script error (or Java applet or class exception) invokes the function assigned to the `onerror` property, passing three parameters:

- Error message
- URL of document causing the error
- Line number of the error

You can essentially trap for all errors and handle them with your own interface (or no user notification at all). The last statement of this function must be `return true` if you do not want the JavaScript script error message to appear.

If you are using LiveConnect to communicate with a Java applet directly from your scripts, you can use the same scheme to handle any exception that Java may throw. A Java exception is not necessarily a mistake kind of error: Some methods assume that the Java code will trap for exceptions to handle special cases (for example, reacting to a user's denial of access when prompted by a signed script dialog box). See Chapter 44 on the CD-ROM for an example of trapping for a specific Java exception. Also, see Chapter 31 for JavaScript exception handling introduced for W3C DOM–compatible browsers.

Example

In Listing 16-10, one button triggers a script that contains an error. I've added an error handling function to process the error so that it opens a separate window and fills in a `textarea` form element (see Figure 16-5). A Submit button is also provided to mail the bug information to a support center e-mail address — an example of how to handle the occurrence of a bug in your scripts.

Listing 16-10: **Controlling Script Errors**

```
<html>
   <head>
      <title>Error Dialog Control</title>
      <script type="text/javascript">
      // function with invalid variable value
      function goWrong() {
         var x = fred;
      }
      // turn off error dialogs
      function errOff() {
         window.onerror = doNothing;
      }
```

Continued

Listing 16-10 *(continued)*

```
      // turn on error dialogs with hard reload
      function errOn() {
         window.onerror = handleError;
      }

      // assign default error handler
      window.onerror = handleError;

      // error handler when errors are turned off...prevents error dialog
      function doNothing() { return true; }
      function handleError(msg, URL, lineNum) {
         var errWind = window.open("","errors","height=270,width=400");
         var wintxt = "<html><body bgcolor=red>";
         wintxt += "<b>An error has occurred on this page.  Please report it to
            Tech Support.</b>";
         wintxt += "<form method=POST enctype='text/plain'
            action=mailTo:support4@dannyg.com >";
         wintxt += "<textarea name='errMsg' cols=45 rows=8 wrap=VIRTUAL>";
         wintxt += "Error: " + msg + "\n";
         wintxt += "URL: " + URL + "\n";
         wintxt += "Line: " + lineNum + "\n";
         wintxt += "Client: " + navigator.userAgent + "\n";
         wintxt += "-----------------------------------------\n";
         wintxt += "Please describe what you were doing when the error
            occurred:";
         wintxt += "</textarea><br />";
         wintxt += "<input type=SUBMIT value='Send Error Report'>";
         wintxt += "<input type=button value='Close' onclick='self.close()'>";
         wintxt += "</form></body></html>";
         errWind.document.write(wintxt);
         errWind.document.close();
         return true;
      }
      </script>
   </head>
   <body>
      <form name="myform">
         <input type="button" value="Cause an Error" onclick="goWrong()" />
         <p><input type="button" value="Turn Off Error Dialogs"
            onclick="errOff()" /> <input type="button"
            value="Turn On Error Dialogs" onclick="errOn()" /></p>
      </form>
   </body>
</html>
```

I provide a button that performs a hard reload, which, in turn, resets the `window.onerror` property to its default value. With error dialog boxes turned off, the error handling function does not run.

Related Items: `location.reload()` method; JavaScript exception handling (Chapter 31); debugging scripts (Chapter 45 on the CD-ROM).

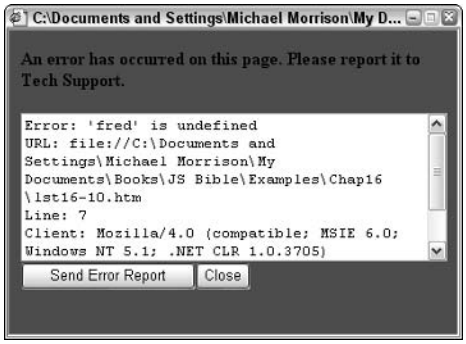


Figure 16-5: An example of a self-reporting error window.

opener

Value: Window object reference. Read/Write
Compatibility: WinIE3+, MacIE3+, NN3+, Moz1+, Safari1+

Many scripters make the mistake of thinking that a new browser window created with the `window.open()` method has a child–parent relationship similar to the one that frames have with their parents. That's not the case at all. New browser windows, once created, have a very slim link to the window from whence they came: via the `opener` property. The purpose of the `opener` property is to provide scripts in the new window with a valid reference back to the original window. For example, the original window may contain some variable values or general-purpose functions that a new window at this Web site wants to use. The original window may also have form elements whose settings are either of value to the new window or get set by user interaction in the new window.

Because the value of the `opener` property is a reference to a genuine `window` object, you can begin references with the property name. Or, you may use the more complete `window.opener` or `self.opener` reference. But the reference must then include some object or property of that original window, such as a window method or a reference to something contained by that window's document.

If a subwindow opens yet another subwindow, the chain is still valid, albeit one step longer. The third window can reach the main window with a reference that begins

```
opener.opener....
```

It's a good idea for the third window to store in a global variable the value of `opener.opener` while the page loads. Thus, if the user closes the second window, the variable can be used to start a reference to the main window.

When a script that generates a new window is within a frame, the `opener` property of the subwindow points to that frame. Therefore, if the subwindow needs to communicate with the main window's parent or another frame in the main window, you have to very carefully build a reference to that distant object. For example, if the subwindow needs to get the `checked` property of a checkbox in a sister frame of the one that created the subwindow, the reference is

```
opener.parent.sisterFrameName.document.formName.checkboxName.checked
```

It is a long way to go, indeed, but building such a reference is always a case of mapping out the path from where the script is to where the destination is, step-by-step.

Example

To demonstrate the importance of the `opener` property, take a look at how a new window can define itself from settings in the main window (see Listing 16-11). The `doNew()` function generates a small subwindow and loads the file in Listing 16-12 into the window. Notice the initial conditional statements in `doNew()` to make sure that if the new window already exists, it comes to the front by invoking the new window's `focus()` method. You can see the results in Figure 16-6.

Listing 16-11: Contents of a Main Window Document That Generates a Second Window

```
<html>
   <head>
      <title>Master of all Windows</title>
      <script type="text/javascript">
      var myWind;
      function doNew() {
         if (!myWind || myWind.closed) {
            myWind = window.open("lst16-12.htm", "subWindow",
               "height=200,width=350,resizable");
         } else {
            // bring existing subwindow to the front
            myWind.focus();
         }
      }
      </script>
   </head>
   <body>
      <form name="input">
         Select a color for a new window: <input type="radio" name="color"
         value="red" checked="checked" />Red <input type="radio" name="color"
         value="yellow" />Yellow <input type="radio" name="color"
         value="blue" />Blue <input type="button" name="storage"
         value="Make a Window" onclick="doNew()" />
         <hr />
         This field will be filled from an entry in another window: <input
         type="text" name="entry" size="25" />
      </form>
   </body>
</html>
```

Listing 16-12: References to the opener Property

```
<html>
   <head>
      <title>New Window on the Block</title>
      <script type="text/javascript">
      function getColor() {
         // shorten the reference
         colorButtons = self.opener.document.forms[0].color;
         // see which radio button is checked
         for (var i = 0; i < colorButtons.length; i++) {
            if (colorButtons[i].checked) {
               return colorButtons[i].value;
            }
         }
         return "white";
      }
      </script>
      <script type="text/javascript">
      document.write("<body bgcolor='" + getColor() + "'>")
      </script>
   </head>
   <body>
      <h1>This is a new window.</h1>
      <form>
         <input type="button" value="Who's in the Main window?"
         onclick="alert(self.opener.document.title)" />
         <p>Type text here for the main window: <input type="text" size="25"
            onchange="self.opener.document.forms[0].entry.value = this.value"
/></p>
      </form>
   </body>
</html>
```

In the `getColor()` function, the multiple references to the radio button array can be very long. To simplify the references, the `getColor()` function starts out by assigning the radio button array to a variable I arbitrarily call `colorButtons`. That shorthand now stands in for lengthy references as I loop through the radio buttons to determine which button is checked and retrieve its value property.

A button in the second window simply fetches the title of the opener window's document. Even if another document loads in the main window in the meantime, the `opener` reference still points to the main window: Its `document` object, however, will change.

Finally, the second window contains a text input object. Enter any text there that you like and either tab or click out of the field. The `onchange` event handler updates the field in the opener's document (provided that document is still loaded).

Related Items: `window.open()`, `window.focus()` methods.

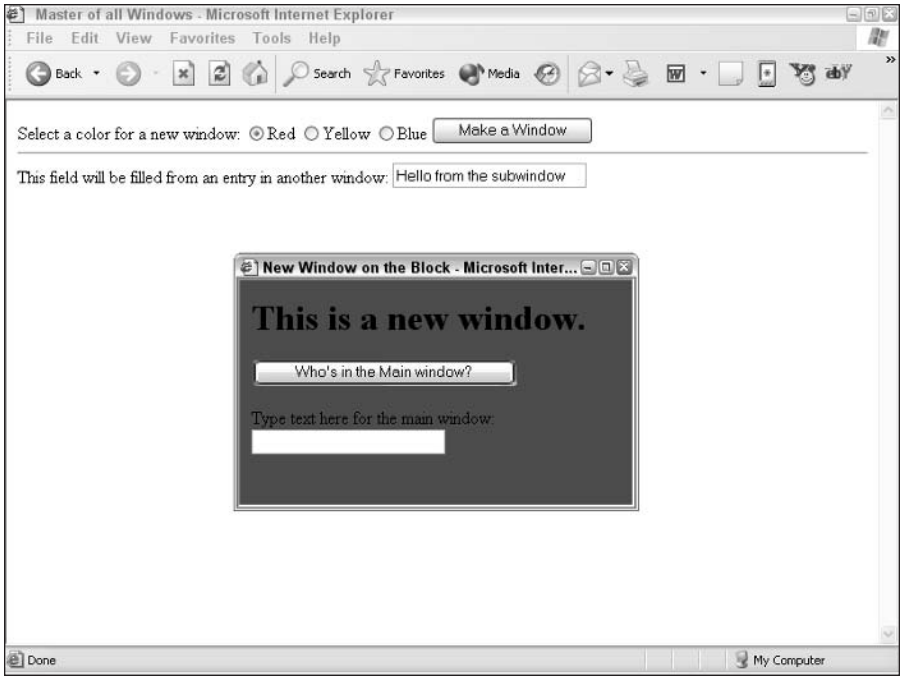


Figure 16-6: The main and subwindows, inextricably linked via the window.opener property.

outerHeight
outerWidth

(See `innerHeight` and `innerWidth`, earlier)

pageXOffset
pageYOffset

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN4+, Moz1+, Safari1+

The top-left corner of the content (inner) region of the browser window is an important geographical point for scrolling documents. When a document is scrolled all the way to the top and flush left in the window (or when a document is small enough to fill the browser window without displaying scrollbars), the document's location is said to be 0,0, meaning zero pixels from the top and zero pixels from the left. If you were to scroll the document, some other coordinate point of the document would be under that top-left corner. That measure is called the page offset, and the `pageXOffset` and `pageYOffset` properties let you read the pixel value of the document at the inner window's top-left corner: `pageXOffset` is the horizontal offset, and `pageYOffset` is the vertical offset.

The value of these measures becomes clear if you design navigation buttons in your pages to carefully control paging of content being displayed in the window. For example, you might have a two-frame page in which one of the frames features navigation controls, while the other displays the primary content. The navigation controls take the place of scrollbars,

which, for aesthetic reasons, are turned off in the display frame. Scripts connected to the simulated scrolling buttons can determine the `pageYOffset` value of the document and then use the `window.scrollTo()` method to position the document precisely to the next logical division in the document for viewing.

IE4+ has corresponding values as `body` object properties: `body.scrollLeft` and `body.scrollTop` (see Chapter 18).

Example

The script in Listing 16-13 is an unusual construction that creates a frameset and creates the content for each of the two frames all within a single HTML document (see "frame Element Object" later in the chapter for more details). The purpose of this example (not available in Safari 1.0) is to provide you with a playground to become familiar with the page offset concept and how the values of these properties correspond to physical activity in a scrollable document.

In the left frame of the frameset are two fields that are ready to show the pixel values of the right frame's `pageXOffset` and `pageYOffset` properties. The content of the right frame is a 30-row table of fixed width (800 pixels). Mouse click events are captured by the document level (see Chapter 18), allowing you to click any table or cell border or outside the table to trigger the `showOffsets()` function in the right frame. That function is a simple script that displays the page offset values in their respective fields in the left frame.

Listing 16-13: **Viewing the pageXOffset and pageYOffset Properties**

```
<html>
   <head>
      <title>Master of all Windows</title>
      <script type="text/javascript">
      function leftFrame() {
         var output = "<html><body<h3>Page Offset Values<\/h3><hr \/>\n";
         output += "<form>PageXOffset:<input type='text' name='xOffset'
            size='4' \/><br \/>\n";
         output += "PageYOffset:<input type='text' name='yOffset' size='4'
            \/><br \/>\n";
         output += "<\/form><\/body><\/html>";
         return output;
      }

      function rightFrame() {
         var output = "<html><head><script type='text/javascript'>\n";
         output += "function showOffsets() {\n";
         output += "parent.readout.document.forms[0].xOffset.value =
            self.pageXOffset;\n";
         output += "parent.readout.document.forms[0].yOffset.value =
            self.pageYOffset;\n}\n";
         output += "document.captureEvents(Event.CLICK);\n";
         output += "document.onclick = showOffsets;\n";
         output += "<\/script><\/head><body><h3>Content Page<\/h3>\n";
         output += "Scroll this frame and click on a table border to view " +
            "page offset values.<br \/><hr \/>\n";
         output += "<table border='5' width='800'>";
```

Continued

Listing 16-13 *(continued)*

```
        var oneRow = "<td>Cell 1<\/td><td>Cell 2<\/td><td>Cell 3<\/td>
           <td>Cell 4<\/td>" + "<td>Cell 5<\/td>";
        for (var i = 1; i <= 30; i++) {
           output += "<tr><td><b>Row " + i + "<\/b><\/td>" + oneRow +
              "<\/tr>";
        }
        output += "<\/table><\/body><\/html>";
        return output;
     }
     </script>
  </head>
  <frameset cols="30%,70%">
     <frame name="readout" src="javascript:parent.leftFrame()" />
     <frame name="display" src="javascript:parent.rightFrame()" />
  </frameset>
</html>
```

To gain an understanding of how the offset values work, scroll the window slightly in the horizontal direction and notice that the `pageXOffset` value increases; the same goes for the `pageYOffset` value as you scroll down. Remember that these values reflect the coordinate in the document that is currently under the top-left corner of the window (frame) holding the document. You can see an IE4+ version of this example in Listing 18-20. A cross-browser version would require very little browser branching.

Related Items: `window.innerHeight`, `window.innerWidth`, `body.scrollLeft`, `body.scrollTop` properties; `window.scrollBy()`, `window.scrollTo()` methods.

parent

Value: Window object reference. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `parent` property (and the `top` property that follows later in this section) comes into play primarily when a document is to be displayed as part of a multiframe window. The HTML documents that users see in the frames of a multiframe browser window are distinct from the document that specifies the frameset for the entire window. That document, though still in the browser's memory (and appearing as the URL in the location field of the browser), is not otherwise visible to the user (except in the Source View).

If scripts in your visible documents need to reference objects or properties of the frameset window, you can reference those frameset window items with the `parent` property (do not, however, expand the reference by preceding it with the `window` object, as in `window.parent.`*`propertyName`*, because this causes problems in early browsers). In a way, the `parent` property seems to violate the object hierarchy because, from a single frame's document, the property points to a level seemingly higher in precedence. If you didn't specify the `parent` property or instead specified the `self` property from one of these framed documents, the object reference is to the frame only, rather than to the outermost framesetting `window` object.

A nontraditional but perfectly legal way to use the `parent` object is as a means of storing temporary variables. Thus, you could set up a holding area for individual variable values or even an array of data. These values can then be shared among all documents loaded into the frames, including when documents change inside the frames. You have to be careful,

however, when storing data in the parent on the fly (that is, in response to user action in the frames). Variables can revert to their default values (that is, the values set by the parent's own script) if the user resizes the window in early browsers.

A child window can also call a function defined in the parent window. The reference for such a function is

```
parent.functionName([parameters])
```

At first glance, it may seem as though the `parent` and `top` properties point to the same framesetting `window` object. In an environment consisting of one frameset window and its immediate children, that's true. But if one of the child windows was, itself, another framesetting window, then you wind up with three generations of windows. From the point of view of the "youngest" child (for example, a window defined by the second frameset), the `parent` property points to its immediate parent, whereas the `top` property points to the first framesetting window in this chain.

On the other hand, a new window created via the `window.open()` method has no parent–child relationship to the original window. The new window's `top` and `parent` point to that new window. You can read more about these relationships in the "Frames" section earlier in this chapter.

Example

To demonstrate how various `window` object properties refer to window levels in a multiframe environment, use your browser to load the Listing 16-14 document. It, in turn, sets each of two equal-size frames to the same document: Listing 16-15. This document extracts the values of several window properties, plus the `document.title` properties of two different window references.

Listing 16-14: **Framesetting Document for Listing 16-15**

```
<html>
   <head>
      <title>The Parent Property Example</title>
      <script type="text/javascript">
      self.name = "Framesetter";
      </script>
   </head>
   <frameset cols="50%,50%" onunload="self.name = ''">
      <frame name="JustAKid1" src="lst16-15.htm" />
      <frame name="JustAKid2" src="lst16-15.htm" />
   </frameset>
</html>
```

Listing 16-15: **Revealing Various Window-Related Properties**

```
<html>
   <head>
      <title>Window Revealer II</title>
      <script type="text/javascript">
      function gatherWindowData() {
```

Continued

Listing 16-15 *(continued)*

```
        var msg = "";
        msg = msg + "top name: " + top.name + "<br />";
        msg = msg + "parent name: " + parent.name + "<br />";
        msg = msg + "parent.document.title: " + parent.document.title +
           "<br />";
        msg = msg + "window name: " + window.name + "<br />";
        msg = msg + "self name: " + self.name + "<br />";
        msg = msg + "self.document.title: " + self.document.title;
        return msg;
     }
     </script>
  </head>
  <body>
     <script type="text/javascript">
     document.write(gatherWindowData());
     </script>
  </body>
</html>
```

In the two frames (see Figure 16-7), the references to the `window` and `self` object names return the name assigned to the frame by the frameset definition (`JustAKid1` for the left frame, `JustAKid2` for the right frame). In other words, from each frame's point of view, the `window` object is its own frame. References to `self.document.title` refer only to the document loaded into that window frame. But references to the top and parent windows (which are one and the same in this example) show that those object properties are shared between both frames.

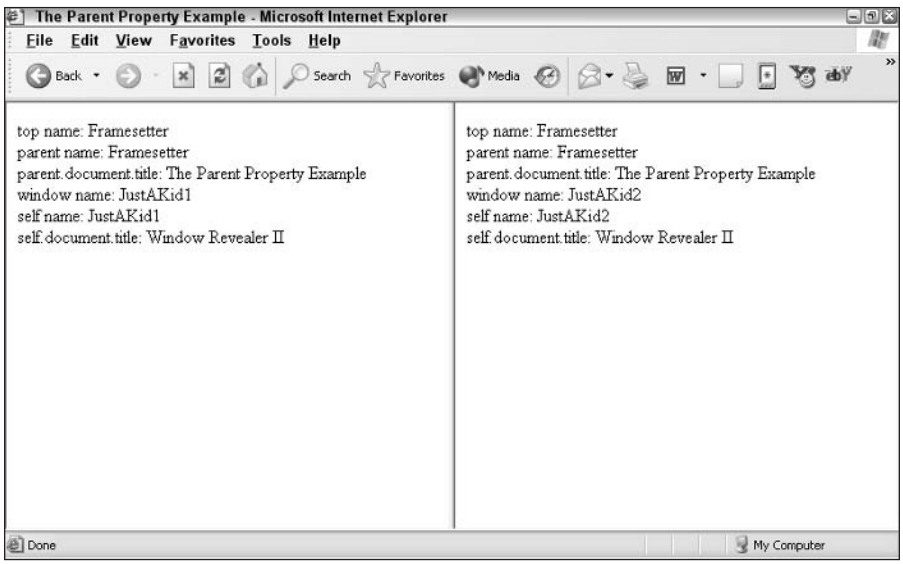


Figure 16-7: Parent and top properties being shared by both frames.

A couple other fine points are worth highlighting. First, the name of the framesetting window is set as Listing 16-14 loads, rather than in response to an `onload` event handler in the `<frameset>` tag. The reason for this is that the name must be set in time for the documents loading in the frames to get that value. If I had waited until the frameset's `onload` event handler, the name wouldn't be set until after the frame documents had loaded. Second, I restore the parent window's name to an empty string when the framesetting document unloads. This is to prevent future pages from getting confused about the window name.

Related Items: `window.frames`, `window.self`, `window.top` properties.

personalbar

(See `directories`)

returnValue

Value: Any data type. Read/Write
Compatibility: WinIE4+, MacIE5+, NN-, Moz-, Safari-

Scripts use the `returnValue` property in a document that loads into the IE-specific modal dialog box. A modal dialog box is generated via the `showModalDialog()` method, which returns whatever data has been assigned to the `returnValue` property of the dialog box window before it closes. This is possible because script processing in the main window freezes while the modal dialog box is visible. As the dialog box closes, a value can be returned to the main window's script right where the modal dialog box was invoked, and the main window's script resumes executing statements.

Example

See Listing 16-39 for the `showModalDialog()` method for an example of how to get data back from a dialog box in IE4+.

Related Items: `showModalDialog()` method.

screen

Value: `screen` object. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Although the `screen` object appears as a property of the `window` object only in the most recent browsers, the `screen` object is also available in NN4 (see Chapter 38 on the CD-ROM), but as a standalone object. Because you can omit any reference to the `window` object for a `window` object's properties, the same window-less reference syntax can be used for compatibility across all browsers that support the `screen` object. That's the way I recommend referring to the `screen` object.

Example

See Chapter 38 on the CD-ROM for examples of using the `screen` object to determine the video monitor characteristics of the computer running the browser.

Related Items: `screen` object.

screenLeft
screenTop

Value: Integer. Read-Only
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

WinIE5+ provides the `screenLeft` and `screenTop` properties of the `window` object to let you read the pixel position (relative to the top-left 0,0 coordinate of the video monitor) of what Microsoft calls the *client area* of the browser window. The client area excludes most window chrome, such as the title bar, address bar, and the window sizing bar. Therefore, when the WinIE5+ browser window is maximized (meaning that no sizing bars are exposed), the `screenLeft` property of the window is 0, while the `screenTop` property varies depending on the combination of toolbars the user has elected to display. For non-maximized windows, if the window has been positioned so that the top and/or left part of the client area are out of view, their property values will be negative integers.

These two properties are read-only. You can position the browser window via the `window.moveTo()` and `window.moveBy()` methods, but these methods position the top-left corner of the entire browser window, not the client area. IE browsers, through version 6, do not provide properties for the position of the entire browser window.

Example

Use The Evaluator (Chapter 13) to experiment with the `screenLeft` and `screenTop` properties. Start with the browser window maximized (if you are using Windows). Enter the following property name into the top text box:

```
window.screenLeft
```

Click the Evaluate button to see the current setting. Unmaximize the window and drag it around the screen. Each time you finish dragging, click the Evaluate button again to see the current value. Do the same for `window.screenTop`.

Related Items: `window.moveTo()`, `window.moveBy()` methods.

screenX
screenY

Value: Integer. Read/Write
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

NN6+/Moz/Safari provide the `screenX` and `screenY` properties to read the position of the outer boundary of the browser window relative to the top-left coordinates (0,0) of the video monitor. The browser window includes the 4-pixel wide window sizing bars that surround Win32 windows. Therefore, when the WinNN6+ browser window is maximized, the values for both `screenX` and `screenY` are -4. Netscape does not provide the equivalent measures of the browser window client area as found in the `screenLeft` and `screenTop` properties of IE5+. You can, however, find out if various toolbars are visible in the browser window (see `window.directories`).

Although you can assign a value to either property, current versions of supporting browsers do not adjust the window position in response. Moving and resizing windows by script is considered by many Web surfers to be unacceptable behavior.

Example

Use The Evaluator (Chapter 13) to experiment with the `screenX` and `screenY` properties. Start with the browser window maximized (if you are using Windows). Enter the following property name into the top text box:

```
window.screenY
```

Click the Evaluate button to see the current setting. Unmaximize the window and drag it around the screen. Each time you finish dragging, click the Evaluate button again to see the current value. Do the same for `window.screenY`.

Related Items: `window.moveTo()`, `window.moveBy()` methods.

scrollbars

(See `directories`)

scrollMaxX
scrollMaxY

Value: Integer. Read/Write
Compatibility: WinIE-, MacIE-, NN7.1+, Moz1.4+, Safari1-

The NN7.1+/Mozilla1.4+ `scrollMaxX` and `scrollMaxY` properties let you determine the maximum horizontal and vertical scrolling extents of a window. Scrolling is possible only if the window displays scrollbars along the desired axis. Values are pixel integers.

Related Items: `scrollX`, `scrollY` properties.

scrollX
scrollY

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The NN6+/Mozilla/Safari `scrollX` and `scrollY` properties let you determine the horizontal and vertical scrolling of a window. Scrolling is possible only if the window displays scroll bars along the desired axis. Values are pixel integers.

Although the IE DOM does not provide similar properties for the window, the same information can be derived from the `body.scrollLeft` and `body.scrollTop` properties.

Example

Use The Evaluator (Chapter 13) to experiment with the `scrollX` and `scrollY` properties. Enter the following property into the top text box:

```
window.scrollY
```

Now manually scroll the page down so that you can still see the Evaluate button. Click the button to see how far the window has scrolled along the *y*-axis.

Related Items: `body.scrollLeft`, `body.scrollTop` properties.

self

Value: Window object reference. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Just as the window object reference is optional, so too is the self property when the object reference points to the same window as the one containing the reference. In what may seem to be an unusual construction, the self property represents the same object as the window. For instance, to obtain the title of the document in a single-frame window, you can use any of the following three constructions:

```
window.document.title
self.document.title
document.title
```

Although self is a property of a window, you should not combine the references within a single-frame window script (for example, don't begin a reference with window.self, which has been known to cause numerous scripting problems). Specifying the self property, though optional for single-frame windows, can help make an object reference crystal clear to someone reading your code (and to you, for that matter). Multiple-frame windows are where you need to pay particular attention to this property.

JavaScript is pretty smart about references to a statement's own window. Therefore, you can generally omit the self part of a reference to a same-window document element. But when you intend to display a document in a multiframe window, complete references (including the self prefix) to an object make it much easier on anyone who reads or debugs your code to track who is doing what to whom. You are free to retrieve the self property of any window. The value that comes back is a window object reference.

Example

Listing 16-16 uses the same operations as Listing 16-5 but substitutes the self property for all window object references. The application of this reference is entirely optional, but it can be helpful for reading and debugging scripts if the HTML document is to appear in one frame of a multiframe window — especially if other JavaScript code in this document refers to documents in other frames. The self reference helps anyone reading the code know precisely which frame was being addressed.

Listing 16-16: Using the self Property

```
<html>
   <head>
      <title>self Property</title>
      <script type="text/javascript">
      self.defaultStatus = "Welcome to my Web site.";
      </script>
   </head>
```

```
   <body>
      <a href="http://www.microsoft.com"
      onmouseover="self.status = 'Visit Microsoft\'s Home page.';return true;"
      onmouseout="self.status = '';return true;">Microsoft</a>
      <p><a href="http://mozilla.org"
         onmouseover="self.status = 'Visit Mozilla\'s Home page.';return
            true;"
         onmouseout="self.status = self.defaultStatus;return
            true;">Mozilla</a></p>
   </body>
</html>
```

Related Items: `window.frames`, `window.parent`, `window.top` properties.

sidebar

(See `appCore`)

status

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

At the bottom of the browser window is a status bar. Part of that bar includes an area that normally discloses the document loading progress or the URL of a link that the mouse is pointing to at any given instant. You can control the temporary content of that field by assigning a text string to the `window` object's `status` property. You should adjust the `status` property only in response to events that have a temporary effect, such as a link or image map area object's `onmouseover` event handler. When the `status` property is set in this situation, it overrides any other setting in the status bar. If the user then moves the mouse pointer away from the object that changes the status bar, the bar returns to its default setting (which may be empty on some pages). Note that Safari 1.0 doesn't always display the new text.

Use this window property as a friendlier alternative to displaying the URL of a link as a user rolls the mouse around the page. For example, if you'd rather use the status bar to explain the nature of the destination of a link, put that text into the status bar in response to the `onmouseover` event handler. But be aware that experienced Web surfers like to see URLs down there. Therefore, consider creating a hybrid message for the status bar that includes both a friendly description followed by the URL in parentheses. In multiframe environments, you can set the `window.status` property without having to worry about referencing the individual frame.

Example

In Listing 16-17, the `status` property is set in a handler embedded in the `onmouseover` attribute of two HTML link tags. Notice that the handler requires a `return true` statement (or any expression that evaluates to `return true`) as the last statement of the handler. This statement is required or the status message will not display in all browsers.

Listing 16-17: **Links with Custom Status Bar Messages**

```
<html>
   <head>
      <title>window.status Property</title>
   </head>
   <body>
      <a href="http://www.dannyg.com"
      onmouseover="window.status = 'Go to my Home page. (www.dannyg.com)';
      return true;">Home</a>
      <p><a href="http://mozilla.org"
      onmouseover="window.status = 'Visit Mozilla Home page. (mozilla.org)';
      return true;">Mozilla</a></p>
   </body>
</html>
```

As a safeguard against platform-specific anomalies that affect the behavior of `onmouseover` event handlers and the `window.status` property, you should also include an `onmouseout` event handler for links and client-side image map area objects. Such `onmouseout` event handlers should set the `status` property to an empty string. This setting ensures that the status bar message returns to the `defaultStatus` setting when the pointer rolls away from these objects. If you want to write a generalizable function that handles all window status changes, you can do so, but word the `onmouseover` attribute carefully so that the event handler evaluates to `return true`. Listing 16-18 shows such an alternative.

Listing 16-18: **Handling Status Message Changes**

```
<html>
   <head>
      <title>Generalizable window.status Property</title>
      <script type="text/javascript">
      function showStatus(msg) {
         window.status = msg;
         return true;
      }
      </script>
   </head>
<body>
      <a href="http://www.example.com"
      onmouseover="return showStatus('Go to my Home page.')"
      onmouseout="return showStatus('')">Home</a>
      <p><a href="http://mozilla.org"
      onmouseover="return showStatus('Visit Mozilla Home page.')"
      onmouseout="return showStatus('')">Mozilla</a></p>
   </body>
</html>
```

Notice how the event handlers return the results of the `showStatus()` method to the event handler, allowing the entire handler to evaluate to `return true`.

One final example of setting the status bar (shown in Listing 16-19) also demonstrates how to create a simple scrolling banner in the status bar.

Listing 16-19: **Creating a Scrolling Banner**

```
<html>
   <head>
      <title>Message Scroller</title>
      <script type="text/javascript">
      var msg = "Welcome to my world...";
      var delay = 150;
      var timerId;
      var maxCount = 0;
      var currCount = 1;

      function scrollMsg() {
         // set the number of times scrolling message is to run
         if (maxCount == 0) {
            maxCount = 3 * msg.length;
         }
         window.status = msg;
         // keep track of how many characters have scrolled
         currCount++;
         // shift first character of msg to end of msg
         msg = msg.substring (1, msg.length) + msg.substring (0, 1);
         // test whether we've reached maximum character count
         if (currCount >= maxCount) {
            timerID = 0;   // zero out the timer
            window.status = "";   // clear the status bar
            return;   // break out of function
         } else {
            // recursive call to this function
            timerId = setTimeout("scrollMsg()", delay);
         }
      }
      </script>
   </head>
   <body onload="scrollMsg()">
   </body>
</html>
```

Because the status bar is being set by a standalone function (rather than by an `onmouseover` event handler), you do not have to append a `return true` statement to set the `status` property. The `scrollMsg()` function uses more advanced JavaScript concepts, such as the `window.setTimeout()` method (covered later in this chapter) and string methods (covered in Chapter 27). To speed the pace at which the words scroll across the status bar, reduce the value of `delay`.

Many Web surfers (myself included) don't care for these scrollers that run forever in the status bar. Rolling the mouse over links disturbs the banner display. Use scrolling bars sparingly or design them to run only a few times after the document loads.

***windowObject*.status**

Setting the `status` property with `onmouseover` event handlers has had a checkered career along various implementations in Navigator. A script that sets the status bar is always in competition against the browser itself, which uses the status bar to report loading progress. When a "hot" area on a page is at the edge of a frame, many times the `onmouseout` event fails to fire, thus preventing the status bar from clearing itself. Be sure to torture test any such implementations before declaring your page ready for public access.

Related Items: `window.defaultStatus` property; `onmouseover`, `onmouseout` event handlers; `link` object.

`statusbar`
`toolbar`

(See `locationbar`)

`top`

Value: Window object refererence. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `window` object's `top` property refers to the topmost window in a frameset object hierarchy. For a single-frame window, the reference is to the same object as the window itself (including the `self` and `parent` properties), so do not include `window` as part of the reference. In a multiframe window, the top window is the one that defines the first frameset (in case of nested framesets). Users don't ever really see the top window in a multiframe environment, but the browser stores it as an object in its memory. The reason is that the top window has the road map to the other frames (if one frame should need to reference an object in a different frame), and its children frames can call upon it. Such a reference looks like

```
top.functionName([parameters])
```

For more about the distinction between the `top` and `parent` properties, see the in-depth discussion about scripting frames at the beginning of this chapter. See also the example of the `parent` property for listings that demonstrate the values of the `top` property.

Related Items: `window.frames`, `window.self`, `window.parent` properties.

`window`

Value: Window object. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Listing the `window` property as a separate property may be more confusing than helpful. The `window` property is the same object as the `window` object. You do not need to use a reference that begins with `window.window`. Although the `window` object is assumed for many references, you can use `window` as part of a reference to items in the same window or frame as the script statement that makes that reference. You should not, however, use `window` as a part of a reference involving items higher up in the hierarchy (`top` or `parent`).

Methods

`alert("message")`

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

An alert dialog box is a modal window that presents a message to the user with a single OK button to dismiss the dialog box. As long as the alert dialog box is showing, no other application or window can be made active. The user must dismiss the dialog box before proceeding with any more work in the browser.

The single parameter to the `alert()` method can be a value of any data type, including representations of some unusual data types whose values you don't normally work with in JavaScript (such as complete objects). This makes the alert dialog box a handy tool for debugging JavaScript scripts. Anytime you want to monitor the value of an expression, use that expression as the parameter to a temporary `alert()` method in your code. The script proceeds to that point and then stops to show you the value. (See Chapter 45 on the CD-ROM for more tips on debugging scripts.)

What is often disturbing to application designers is that all JavaScript-created modal dialog boxes (via the `alert()`, `confirm()`, and `prompt()` methods) identify themselves as being generated by JavaScript or the browser. The purpose of this identification is to act as a security precaution against unscrupulous scripters who might try to spoof system or browser alert dialog boxes, inviting a user to reveal passwords or other private information. These identifying words cannot be overwritten or eliminated by your scripts. You can simulate a modal dialog box window in a cross-browser fashion with regular browser windows, but it is not as robust as a genuine modal window, which you can create in IE4+ via the `window.showModalDialog()` method.

Because the `alert()` method is of a global nature (that is, no particular frame in a multiframe environment derives any benefit from laying claim to the alert dialog box), a common practice is to omit all `window` object references from the statement that calls the method. Restrict the use of alert dialog boxes in your HTML documents and site designs. The modality of the windows is disruptive to the flow of a user's navigation around your pages. Communicate with users via forms or by writing to separate document window frames.

Example

The parameter for the example in Listing 16-20 is a concatenated string. It joins together two fixed strings and the value of the browser's `navigator.appName` property. Loading this document causes the alert dialog box to appear, as shown in several configurations in Figure 16-8. The JavaScript Alert: line cannot be deleted from the dialog box in earlier browsers, nor can the title bar be changed in later browsers.

Listing 16-20: **Displaying an Alert Dialog Box**

```
<html>
   <head>
      <title>window.alert() Method</title>
   </head>
   <body>
      <script type="text/javascript">
      alert("You are running the " + navigator.appName + " browser.")
      </script>
   </body>
</html>
```

Related Items: `window.confirm()`, `window.prompt()` methods.

Figure 16-8: Results of the alert() method in Listing 16-20 in Internet Explorer and Netscape Navigator.

back()
forward()

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

The purpose of the `window.back()` and `window.forward()` methods in NN4 is to offer a scripted version of the global back and forward navigation buttons, while allowing the `history` object to control navigation strictly within a particular window or frame — as it should. These window methods did not catch on in IE (and the `window` object is out of the scope of the W3C DOM Level 2), so you are better off staying with the `history` object's methods for navigating through browser history. For more information about version compatibility and the back and forward navigation, see the `history` object in Chapter 17.

Related Items: `history.back()`, `history.forward()`, `history.go()` methods.

captureEvents(*eventTypeList*)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

In Navigator 4, an event filters down from the `window` object and eventually reaches its intended target. For example, if you click a button, the click event first reaches the `window` object; then it goes to the `document` object; and eventually (in a split second) it reaches the button, where an `onclick` event handler is ready to act on that click.

The NN4 "trickle-down" event propagation mechanism allows `window`, `document`, and layer objects to intercept events and process them prior to reaching their intended targets (or preventing them from reaching their destinations entirely). But for one of these outer containers to grab an event, your script must instruct it to capture the type of event your application is interested in preprocessing. If you want the `window` object to intercept all events of a particular type, use the `window.captureEvents()` method to turn that facility on.

NN6+ and W3C browsers have both a trickle-down and bubble-up event model combination. The syntax for using event capture in NN6+ is quite different from that in NN4. The discussions of the `captureEvents()`, `releaseEvents()`, `handleEvent()`, and `routeEvent()` methods of the `window`, `document`, and layer objects apply only to Navigator 4. If your DHTML page design does not need to support NN4, you can skip these discussions.

The `window.captureEvents()` method takes one or more event types as parameters. An event type is a constant value built inside the Navigator 4 `Event` object. One event type exists for every kind of event handler you see in all of the Navigator 4 document objects. The syntax is the event object name (`Event`) and the event name in all uppercase letters. For example, if you want the window to intercept all click events, the statement is

```
window.captureEvents(Event.CLICK);
```

For multiple events, add them as parameters, separated by the pipe (`|`) character:

```
window.captureEvents(Event.MOUSEDOWN | Event.KEYPRESS);
```

After an event type is captured by the `window` object, a function must be ready to deal with the event. For example, perhaps the function looks through all `Event.MOUSEDOWN` events and looks to see if the right mouse button was the one that triggered the event and what form element (if any) is the intended target. The goal is to perhaps display a pop-up menu (as a separate layer) for a right-click. If the click comes from the left mouse button, the event is routed to its intended target.

To associate a function with a particular event type captured by a `window` object, assign a function to the event. For example, to assign a custom `doClickEvent()` function to `click` events captured by the `window` object, use the following statement:

```
window.onclick = doClickEvent;
```

Note that the function name is assigned only as a reference name (no quotes or parentheses), not like an event handler within a tag. The function itself is like any function, but it has the added benefit of automatically receiving an instance of the `Event` object as a parameter. To turn off event capture for one or more event types, use the `window.releaseEvent()` method.

Capturing events at the `window`, `document`, or layer level in NN4 does not always work the way you might like. This is especially true if your page contains tables. For example, capturing mouse events has no effect in the Windows version of NN4 unless the cursor is atop a cell border. Event capture works most reliably when a scriptable object has an event handler defined for it (even if it is an empty string) and the element is the target of the event (for example, you are about to type into a text field). For all other elements, events may simply not be captured at the document or window level.

Example

The page in Listing 16-21 is an exercise in capturing and releasing click events in the `window` object. Whenever the window is capturing click events, the `flash()` function runs. In that function, the event is examined so that only if the Control key is also being held down and the name of the button starts with "button" does the document background color flash red. For all click events (that is, those directed at objects on the page capable of their own `onclick` event handlers), the click is processed with the `routeEvent()` method to make sure the target buttons execute their own `onclick` event handlers.

Listing 16-21: **Capturing Click Events in the Window**

```
<html>
   <head>
      <title>Window Event Capture</title>
      <script type="text/javascript">
      // function to run when window captures a click event
      function flash(e) {
         if (e.modifiers == Event.CONTROL_MASK &&
            e.target.name.indexOf("button") == 0) {
            document.bgColor = "red";
            setTimeout("document.bgColor = 'white'", 500);
         }
         // let event continue to target
         routeEvent(e);
      }

      // default setting to capture click events
      window.captureEvents(Event.CLICK);
      // assign flash() function to click events captured by window
      window.onclick = flash;
      </script>
   </head>
   <body bgcolor="white">
      <form name="buttons">
         <b>Turn window click event capture on or off (Default is "On")</b>
         <p><input name="captureOn" type="button" value="Capture On"
            onclick="window.captureEvents(Event.CLICK)" />  <input
            name="captureOff" type="button" value="Capture Off"
            onclick="window.releaseEvents(Event.CLICK)" /></p>
         <hr />
         <b>Ctrl+Click on a button to see if clicks are being captured by the
         window (background color will flash red):</b>
         <ul>
            <li><input name="button1" type="button" value="Informix"
               onclick="alert('You clicked on Informix.')" /></li>
            <li><input name="button2" type="button" value="Oracle"
               onclick="alert('You clicked on Oracle.')" /></li>
            <li><input name="button3" type="button" value="Sybase"
               onclick="alert('You clicked on Sybase.')" /></li>
         </ul>
      </form>
   </body>
</html>
```

When you try this page, also turn off window event capture. Now only the buttons' `onclick` event handlers execute, and the page does not flash red.

Related Items: `window.disableExternalCapture()`, `window.enableExternalCapture()`, `window.handleEvent()`, `window.releaseEvents()`, `window.routeEvent()` methods.

`clearInterval(`*`intervalIDnumber`*`)`

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

Use the window.clearInterval() method to turn off an interval loop action started with the window.setInterval() method. The parameter is the ID number returned by the setInterval() method. A common application for the JavaScript interval mechanism is animation of an object on a page. If you have multiple intervals running, each has its own ID value in memory. You can turn off any interval by its ID value. As soon as an interval loop stops, your script cannot resume that interval: It must start a new one, which generates a new ID value.

Example

See Listings 16-36 and 16-37 for an example of how setInterval() and clearInterval() are used together on a page.

Related Items: window.setInterval(), window.setTimeout(), window.clearTimeout() methods.

clearTimeout(*timeoutIDnumber*)

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Use the window.clearTimeout() method in concert with the window.setTimeout() method, as described later in this chapter, when you want your script to cancel a timer that is waiting to run its expression. The parameter for this method is the ID number that the window.setTimeout() method returns when the timer starts ticking. The clearTimeout() method cancels the specified timeout. A good practice is to check your code for instances where user action may negate the need for a running timer — and to stop that timer before it goes off.

Example

The page in Listing 16-22 features one text field and two buttons (see Figure 16-9). One button starts a countdown timer coded to last one minute (easily modifiable for other durations); the other button interrupts the timer at any time while it is running. When the minute is up, an alert dialog box lets you know.

Listing 16-22: A Countdown Timer

```
<html>
   <head>
      <title>Count Down Timer</title>
      <script type="text/javascript">
      var running = false;
      var endTime = null;
      var timerID = null;

      function startTimer() {
         running = true;
         now = new Date();
         now = now.getTime();
         // change last multiple for the number of minutes
         endTime = now + (1000 * 60 * 1);
         showCountDown();
      }
```

Continued

Listing 16-22 *(continued)*

```
      function showCountDown() {
         var now = new Date();
         now = now.getTime();
         if (endTime - now <= 0) {
            stopTimer();
            alert("Time is up.  Put down your pencils.");
         } else {
            var delta = new Date(endTime - now);
            var theMin = delta.getMinutes();
            var theSec = delta.getSeconds();
            var theTime = theMin;
            theTime += ((theSec < 10) ? ":0" : ":") + theSec;
            document.forms[0].timerDisplay.value = theTime;
            if (running) {
               timerID = setTimeout("showCountDown()",1000);
            }
         }
      }

      function stopTimer() {
         clearTimeout(timerID);
         running = false;
         document.forms[0].timerDisplay.value = "0:00";
      }
      </script>
   </head>
   <body>
      <form>
         <input type="button" name="startTime" value="Start 1 min. Timer"
         onclick="startTimer()" /> <input type="button" name="clearTime"
         value="Clear Timer" onclick="stopTimer()" />
         <p><input type="text" name="timerDisplay" value="" /></p>
      </form>
   </body>
</html>
```

Notice that the script establishes three variables with global scope in the window: `running`, `endTime`, and `timerID`. These values are needed inside multiple functions, so they are initialized outside of the functions.

In the `startTimer()` function, you switch the running flag on, meaning that the timer should be going. Using some date functions (see Chapter 29), you extract the current time in milliseconds and add the number of milliseconds for the next minute (the extra multiplication by one is the place where you can change the amount to the desired number of minutes). With the end time stored in a global variable, the function now calls another function that compares the current and end times and displays the difference in the text field.

Early in the `showCountDown()` function, check to see if the timer has wound down. If so, you stop the timer and alert the user. Otherwise, the function continues to calculate the difference between the two times and formats the time in mm:ss format. As long as the `running` flag is set to `true`, the function sets the one-second timeout timer before repeating itself. To stop the timer before it has run out (in the `stopTimer()` function), the most important step is to cancel the timeout running inside the browser. The `clearTimeout()` method uses the global `timerID` value to do that. Then the function turns off the running switch and zeros out the display.

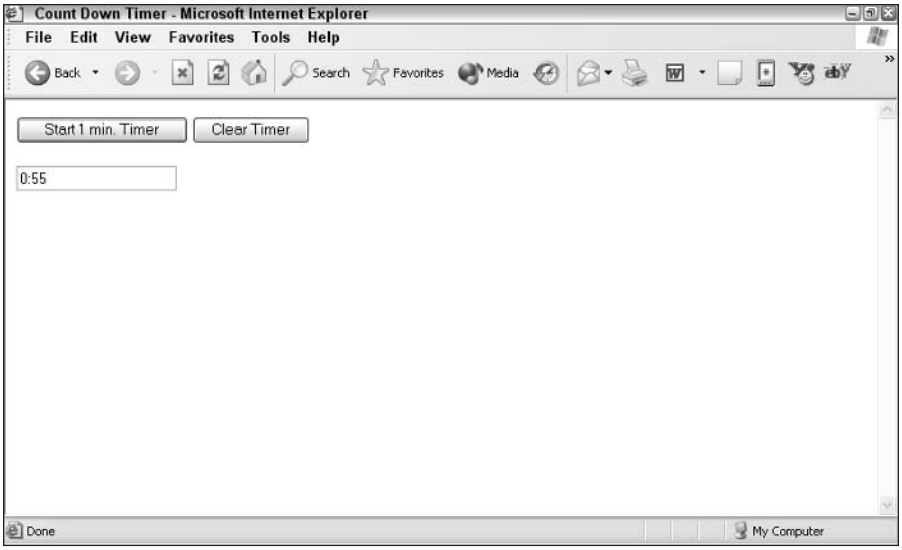


Figure 16-9: The countdown timer page as it displays the time remaining.

When you run the timer, you may occasionally notice that the time skips a second. It's not cheating. It just takes slightly more than one second to wait for the timeout and then finish the calculations for the next second's display. What you're seeing is the display catching up with the real time left.

Related Items: `window.setTimeout()` method.

close()

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `window.close()` method closes the browser window referenced by the `window` object. Most likely, you will use this method to close subwindows created from a main document window. If the call to close the window comes from a window other than the new subwindow, the original `window` object must maintain a record of the subwindow object. You accomplish this by storing the value returned from the `window.open()` method in a global variable that will be available to other objects later (for example, a variable not initialized inside a function). If, on the other hand, an object inside the new subwindow calls the `window.close()` method, the `window` or `self` reference is sufficient.

Be sure to include a window as part of the reference to this method. Failure to do so may cause JavaScript to regard the statement as a `document.close()` method, which has different behavior (see Chapter 18). Only the `window.close()` method can close the window via a script. Closing a window, of course, forces the window to trigger an `onunload` event handler before the window disappears from view; but after you've initiated the `window.close()` method, you cannot stop it from completing its task. Moreover, `onunload` event handlers that attempt to execute time-consuming processes (such as submitting a form in the closing window) may not complete because the window can easily close before the process completes — a behavior that has no workaround (with the exception of the `onbeforeunload` event handler in IE4+).

While I'm on the subject of closing windows, a special case exists when a subwindow tries to close the main window (via a statement such as `self.opener.close()`) when the main window has more than one entry in its session history. As a safety precaution against scripts closing windows they did not create, modern browsers ask the user whether he or she wants the main window to close (via a browser-generated dialog box). This security precaution cannot be overridden except in NN4+/Moz via a signed script when the user grants permission to control the browser (see Chapter 46 on the CD-ROM).

Example

See Listing 16-4 (for the `window.closed` property), which provides a cross-platform example of applying the `window.close()` method across multiple windows.

Related Items: `window.open()`, `document.close()` methods.

`confirm("message")`

Returns: Boolean.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

A confirm dialog box presents a message in a modal dialog box along with OK and Cancel buttons. Such a dialog box can be used to ask a question of the user, usually prior to a script performing actions that will not be undoable. Querying a user about proceeding with typical Web navigation in response to user interaction on a form element is generally a disruptive waste of the user's time and attention. But for operations that may reveal a user's identity or send form data to a server, a JavaScript confirm dialog box may make a great deal of sense. Users can also accidentally click buttons, so you should provide avenues for backing out of an operation before it executes.

Because this dialog box returns a Boolean value (OK = `true`; Cancel = `false`), you can use this method as a comparison expression or as an assignment expression. In a comparison expression, you nest the method within any other statement where a Boolean value is required. For example:

```
if (confirm("Are you sure?")) {
   alert("OK");
} else {
   alert("Not OK");
}
```

Here, the returned value of the confirm dialog box provides the desired Boolean value type for the `if...else` construction (see Chapter 31).

This method can also appear on the right side of an assignment expression, as in

```
var adult = confirm("You certify that you are over 18 years old?");
if (adult) {
   //statements for adults
} else {
   //statements for children
}
```

You cannot specify other alert icons or labels for the two buttons in JavaScript confirm dialog box windows.

The example in Listing 16-23 shows the user interface part of how you can use a confirm dialog box to query a user before clearing a table full of user-entered data. The line in the title bar, as shown in Figure 16-10, or the "JavaScript Confirm" legend in earlier browser versions cannot be removed from the dialog box.

Listing 16-23: **The Confirm Dialog Box**

```
<html>
   <head>
      <title>window.confirm() Method</title>
      <script type="text/javascript">
      function clearTable() {
         if (confirm("Are you sure you want to empty the table?")) {
            alert("Emptying the table...");   // for demo purposes
            //statements that actually empty the fields
         }
      }
      </script>
   </head>
   <body>
      <form>
         <!-- other statements that display and populate a large table -->
         <input type="button" name="clear" value="Reset Table"
         onclick="clearTable()" />
      </form>
   </body>
</html>
```

Figure 16-10: A JavaScript confirm dialog box in WinIE.

Related Items: `window.alert()`, `window.prompt()`, `form.submit()` methods.

`createPopup()`

Returns: Pop-up object reference.
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

An IE pop-up window is a chrome-less rectangular space that overlaps the current window. Unlike the dialog boxes generated by the `showModalDialog()` and `showModelessDialog()` methods, the pop-up window's entire content must be explicitly controlled by script. That also goes for the size and location of the window. Generating the window via the `createPopup()` method simply creates the object in memory without displaying it. You can then use the reference to the pop-up window that is returned by the method to position the window, populate its content, and make it visible. See details in the description of the `popup` object later in this chapter.

Example

See Listing 16-49 later in this chapter for an example of the `createPopup()` method.

Related Items: `popup` object.

***windowObject*.execScript()**

execScript("*exprList*"[, *language*])

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The IE-specific `window.execScript()` method executes one or more script statements that are passed as string expressions. The first parameter is a string version of one or more script statements (multiple statements must be separated by semicolons). The second, optional parameter is the language interpreter the browser should use to execute the script statement. Acceptable values for the language are `JavaScript`, `JScript`, `VBS`, and `VBScript`. The default value is `JScript`, so you can omit the second parameter when supplying expressions in JavaScript.

Unlike the JavaScript core language `eval()` function (which also executes string versions of JavaScript statements), the `execScript()` method returns no values. Even so, the method operates within the global variable space of the window holding the current document. For example, if a document's script declares a global variable as follows

```
var myVar;
```

the `execScript()` method can read or write to that variable:

```
window.execScript("myVar = 10; myVar += 5");
```

After this statement runs, the global variable `myVar` has a value of 15.

Example

Use The Evaluator (Chapter 13) to experiment with the `execScript()` method. The Evaluator has predeclared global variables for the lowercase letters `a` through `z`. Enter each of the following statements into the top text box and observe the results for each.

```
a
```

When first loaded, the variable is declared but assigned no value, so it is undefined.

```
window.execScript("a = 5")
```

The method returns no value, so the mechanism inside The Evaluator says that the statement is undefined.

```
a
```

The variable is now 5.

```
window.execScript("b = a * 50")
b
```

The `b` global variable has a value of 250. Continue exploring with additional script statements. Use semicolons to separate multiple statements within the string parameter.

Related Items: `eval()` function.

find(["*searchString*" [, *matchCaseBoolean*, *searchUpBoolean*]])

Returns: Boolean value for nondialog searches.
Compatibility: WinIE-, MacIE-, NN4+, Moz1+, Safari-

The `window.find()` method introduced in NN4 mimics the powers of the browser's Find dialog box, accessible from the Find button in the toolbar. This method was deactivated in NN6 but reactivated in NN7/Moz1.

If you specify no parameters, the browser's Find dialog box appears, just as if the user had clicked the Find button in the toolbar. With no parameters, this function does not return a value.

You can specify a search string as a parameter to the function. The search is based on simple string matching and is not in any way connected with the regular expression kind of search (see Chapter 42 on the CD-ROM). If the search finds a match, the browser scrolls to that matching word and highlights the word, just as if using the browser's own Find dialog box. The function also returns a Boolean `true` after a match is found. If no match is found in the document or no more matches occur in the current search direction (the default direction is from top to bottom), the function returns `false`.

Two optional Boolean parameters to the scripted find action let you specify whether the search should be case-sensitive and whether the search direction should be upward from the bottom of the document. These choices are identical to the ones that appear in the NN4+'s Find dialog box. Default behavior is case-insensitive and searches from top to bottom.

IE4+ also has a text search facility, but it is implemented in an entirely different way (using the `TextRange` object described in Chapter 35). The visual behavior also differs in that it does not highlight and scroll to a matching string in the text.

Example

A simple call to the `window.find()` method looks as follows:

```
var success = window.find("contract");
```

And if you want the search to be case-sensitive, add at least one of the two optional parameters:

```
success = window.find(matchString,caseSensitive,backward);
```

Because this method works only in NN4, refer to discussions of the `TextRange` and `Range` objects in Chapter 35 on the CD-ROM for more modern implementations of body text searching.

Related Items: `TextRange`, `Range` objects (Chapter 35).

forward()

(See `window.back()`)

handleEvent(*event*)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

When you explicitly capture events in the NN4 `window`, `document`, or layer object (by invoking the `captureEvents()` method for that object), you can control where the events go after their initial capture. To let an event continue to its original target (for example, a button that was clicked by a user), you use the `routeEvent()` method. But if you want to redirect an event (or class of events) to a particular event handler elsewhere in the document, use the `handleEvent()` method.

Every NN4 object that has event handlers associated with it also has a `handleEvent()` method. Thus, if you are capturing click events in a window, you can redirect the events to, say, a particular button or link on the page because both of those objects know what to do with click events. Consider the following code excerpt:

```
<script type="text/javaScript">
```

```
// function to run when window captures a click event
function doClicks(evt) {
   // send all clicks to the first link in the document
   document.links[0].handleEvent(evt);
}
// set window to capture click events
window.captureEvents(Event.CLICK);
// assign doClick() function to click events captured by window
window.onclick = doClicks;
</script>
```

The window is set up to capture all click events and invoke the `doClicks()` function each time the user clicks a clickable item in the window. In the `doClicks()` function is a single statement that instructs the first link in the document to handle the click event being passed as a parameter. The link must have an `onclick` event handler defined for this to be meaningful. Because an `event` object is passed along automatically, the link's event handler can examine event properties (for example, location of the click) and perhaps alter some of the link's properties before letting it perform its linking task. The preceding example is really showing how to use `handleEvent()` with a `link` object, rather than a `window` object. There is little opportunity for other objects to capture events that normally go to the window, but this method is part of every event-aware object in NN4.

The corresponding method in the W3C event model's capture mechanism is `dispatchEvent()`, and the IE5+ equivalent is `fireEvent()`.

See Chapter 25 for details and in-depth examples of working with event objects.

Related Items: `window.captureEvents()`, `window.releaseEvents()`, `window.routeEvent()` methods; `event` object.

home()

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

Like many of the window methods introduced in Navigator 4, the `window.home()` method provides an NN-specific scripted way of replicating the action of a toolbar button: the Home button. The action navigates the browser to whatever URL is set in the browser preferences for home page location. You cannot control the default home page of a visitor's browser.

Related Items: `window.back()`, `window.forward()` methods; `window.toolbar` property.

moveBy(deltaX,deltaY)
moveTo(*x*,*y*)

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

As of version 4 browsers, JavaScript can adjust the location of a browser window on the screen (no longer available in Mozilla-based browsers, however). This applies to the main window or any subwindow generated by script. Netscape regards the possibility of a window moved out of screen view as a potential security hole, so signed scripts are needed in NN4+ to move a window off screen.

You can move a window to an absolute position on the screen or adjust it along the horizontal and/or vertical axis by any number of pixels, irrespective of the absolute pixel position.

The coordinate space for the *x* (horizontal) and *y* (vertical) position is the entire screen, with the top-left corner representing 0,0. The point of the window you set with the `moveBy()` and `moveTo()` methods is the very top-left corner of the outer edge of the browser window. Therefore, when you move the window to point 0,0, that sets the window flush with the top-left corner of the screen. This may not be the equivalent of a truly maximized window for all browsers and operating systems, however, because a maximized window's coordinates may be negative by a handful of pixels.

If you try to adjust the position of the window in NN4 such that any edge falls beyond the screen area, the window remains at the edge of the screen — unless you are using a signed script and have the user's permission to adjust the window partially or completely off screen. Moving the only visible browser window entirely off screen is dangerous because the user has no way to get it back into view without quitting and relaunching the browser.

The difference between the `moveTo()` and `moveBy()` methods is that one is an absolute move, while the other is relative with respect to the current window position. Parameters you specify for `moveTo()` are the precise horizontal and vertical pixel counts on the screen where you want the upper-left corner of the window to appear. In contrast, the parameters for `moveBy()` indicate how far to adjust the window location in either direction. If you want to move the window 25 pixels to the right, you must still include both parameters, but the *y* value will be zero:

```
window.moveBy(25,0);
```

To move to the left, the first parameter must be a negative number.

Example

Several examples of using the `window.moveTo()` and `window.moveBy()` methods are shown in Listing 16-24. The page presents four buttons, each of which performs a different kind of browser window movement.

Listing 16-24: **Window Boogie**

```
<html>
   <head>
      <title>Window Gymnastics</title>
      <script type="text/javascript">
      var isNav4 = ((navigator.appName == "Netscape") &&
         (parseInt(navigator.appVersion) >= 4));

      // wait in onload for page to load and settle in IE
      function init() {
         // fill missing IE properties
         if (!window.outerWidth) {
            window.outerWidth = document.body.clientWidth;
            window.outerHeight = document.body.clientHeight + 30;
         }
         // fill missing IE4 properties
         if (!screen.availWidth) {
            screen.availWidth = 640;
            screen.availHeight = 480;
         }
      }
```

Continued

Listing 16-24 *(continued)*

```
    // function to run when window captures a click event
    function moveOffScreen() {
       // branch for NN security
       if (isNav4) {
          netscape.security.PrivilegeManager.enablePrivilege(
             "UniversalBrowserWrite");
       }
       var maxX = screen.width;
       var maxY = screen.height;
       window.moveTo(maxX+1, maxY+1);
       setTimeout("window.moveTo(0,0)",500);
       if (isNav4) {
          netscape.security.PrivilegeManager.disablePrivilege(
             "UniversalBrowserWrite");
       }
    }

    // moves window in a circular motion
    function revolve() {
       var winX = (screen.availWidth - window.outerWidth) / 2;
       var winY = 50;
       window.resizeTo(400,300);
       window.moveTo(winX, winY);

       for (var i = 1; i < 36; i++) {
          winX += Math.cos(i * (Math.PI/18)) * 5;
          winY += Math.sin(i * (Math.PI/18)) * 5;
          window.moveTo(winX, winY);
       }
    }

    // moves window in a horizontal zig-zag pattern
    function zigzag() {
       window.resizeTo(400,300);
       window.moveTo(0,80);
       var incrementX = 2;
       var incrementY = 2;
       var floor = screen.availHeight - window.outerHeight;
       var rightEdge = screen.availWidth - window.outerWidth;
       for (var i = 0; i < rightEdge; i += 2) {
          window.moveBy(incrementX, incrementY);
          if (i%60 == 0) {
             incrementY = -incrementY;
          }
       }
    }

    // resizes window to occupy all available screen real estate
    function maximize() {
       window.moveTo(0,0);
       window.resizeTo(screen.availWidth, screen.availHeight);
    }
    </script>
  </head>
```

```
   <body onload="init()">
      <form name="buttons">
         <b>Window Gymnastics</b>
         <ul>
            <li><input name="offscreen" type="button"
               value="Disappear a Second" onclick="moveOffScreen()" /></li>
            <li><input name="circles" type="button" value="Circular Motion"
               onclick="revolve()" /></li>
            <li><input name="bouncer" type="button" value="Zig Zag"
               onclick="zigzag()" /></li>
            <li><input name="expander" type="button" value="Maximize"
               onclick="maximize()" /></li>
         </ul>
      </form>
   </body>
</html>
```

To run successfully in NN (prior to NN7), the first button requires that you have codebase principals turned on (see Chapter 46 on the CD-ROM) to take advantage of what would normally be a signed script. The `moveOffScreen()` function momentarily moves the window entirely out of view. Notice how the script determines the size of the screen before deciding where to move the window. After the journey off screen, the window comes back into view at the upper-left corner of the screen.

If using the Web sometimes seems like going around in circles, then the second function, `revolve()`, should feel just right. After reducing the size of the window and positioning it near the top center of the screen, the script uses a bit of math to position the window along 36 places around a perfect circle (at 10-degree increments). This is an example of how to control a window's position dynamically based on math calculations. IE complicates the job a bit by not providing properties that reveal the outside dimensions of the browser window.

To demonstrate the `moveBy()` method, the third function, `zigzag()`, uses a `for` loop to increment the coordinate points to make the window travel in a sawtooth pattern across the screen. The *x* coordinate continues to increment linearly until the window is at the edge of the screen (also calculated on the fly to accommodate any size monitor). The y coordinate must increase and decrease as that parameter changes direction at various times across the screen.

In the fourth function, you see some practical code (finally) that demonstrates how best to simulate maximizing the browser window to fill the entire available screen space on the visitor's monitor.

Related Items: `window.outerHeight`, `window.outerWidth` properties; `window.resizeBy()`, `window.resizeTo()` methods.

navigate("*URL*")

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN-, Moz-, Safari-

The `window.navigate()` method is an IE-specific method that lets you load a new document into a window or frame. This method's action is the same as assigning a URL to the `location.href` property — a property that is available on all scriptable browsers. If your audience is entirely IE-based, this method is safe. Otherwise, I recommend the `location.href` property as the best navigation approach.

Example

Supply any valid URL as the parameter to the method, as in

```
window.navigate("http://www.dannyg.com");
```

Related Items: `location` object.

open("*URL*", "*windowName*" [, "*windowFeatures*"][,*replaceFlag*])

Returns: A window object representing the newly created window; `null` if method fails.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

With the `window.open()` method, a script provides a Web site designer with an immense range of options for the way a second or third Web browser window looks on the user's computer screen. Moreover, most of this control can work with all JavaScript-enabled browsers without the need for signed scripts. Because the interface elements of a new window are easier to envision, I cover those aspects of the `window.open()` method parameters first.

Setting new window features

The optional *windowFeatures* parameter is one string, consisting of a comma-separated list of assignment expressions (behaving something like HTML tag attributes). **Important:** For the best browser compatibility, do not put spaces after the commas. If you omit the third parameter, JavaScript creates the same type of new window you get from the New Web Browser menu choice in the File menu. But you can control which window elements appear in the new window with the third parameter. Remember this important rule: If you specify even one of the method's original set of third parameter values, all other features are turned off unless the parameters specify the features to be switched on. Table 16-3 lists the attributes that you can control for a newly created window in all browsers. Except where noted, all Boolean values default to `yes` if you do not specify the third parameter.

Table 16-3: window.open() Method Attributes Controllable via Script

Attribute	*Browsers*	*Description*
`alwaysLowered3`	NN4+	(Boolean) Always behind other browser windows
`alwaysRaised3`	NN4+	(Boolean) Always in front of other browser windows
`channelMode`	IE4+	(Boolean) Theater mode with channel band (default is no)
`copyhistory`	NN2+, IE3+	(Boolean) Duplicates Go menu history for new window
`dependent`	NN4+	(Boolean) Subwindow closes if the opener window closes
`directories`	NN2+, IE3+	(Boolean) "What's New" and other buttons in the row
`fullscreen`	IE4+	(Boolean) No title bar or menus (default is no)
`height`	NN2+, IE3+	(Integer) Content region height in pixels
`hotkeys`	NN4+	(Boolean) If true, disables menu shortcuts (except Quit and Security Info) when menu bar is turned off
`innerHeight4`	NN4+	(Integer) Content region height; same as old height property
`innerWidth4`	NN4+	(Integer) Content region width; same as old width property
`left`	NN6+, IE4+	(Integer) Horizontal position of top-left corner on screen
`location`	NN2+, IE3+	(Boolean) Field displaying the current URL
`menubar1`	NN2+, IE3+	(Boolean) Menu bar at top of window
`outerHeight4`	NN4+	(Integer) Visible window height

Attribute	*Browsers*	*Description*
outerWidth[4]	NN4+	(Integer) Visible window width
resizable[2]	NN2+, IE3+	(Boolean) Interface elements that allow resizing by dragging
screenX[4]	NN4+	(Integer) Horizontal position of top-left corner on screen
screenY[4]	NN4+	(Integer) Vertical position of top-left corner on screen
scrollbars	NN2+, IE3+	(Boolean) Displays scrollbars if document is larger than window
status	NN2+, IE3+	(Boolean) Status bar at bottom of window
titlebar[3]	NN4+	(Boolean) Title bar and all other border elements
toolbar	NN2+, IE3+	(Boolean) "Back," "Forward," and other buttons in the row
top	NN6+, IE4+	(Integer) Horizontal position of top-left corner on screen
width	NN2+, IE3+	(Integer) Content region width in pixels
z-lock[3]	NN4+	(Boolean) Window layer is fixed below browser windows

1 Not on Macintosh because the menu bar is not in the browser window; when off in MacNN4, displays an abbreviated Mac menu bar.

2 Macintosh windows are always resizable.

3 Requires a signed script.

4 Requires a signed script to size or position a window beyond safe threshold.

Boolean values are handled a bit differently than you might expect. The value for `true` can be either `yes`, `1`, or just the feature name by itself; for `false`, use a value of `no` or `0`. If you omit any Boolean attributes, they are rendered as `false`. Therefore, if you want to create a new window that shows only the toolbar and status bar and is resizable, the method looks like this:

```
window.open("newURL","NewWindow", "toolbar,status,resizable");
```

A new window that does not specify the height and width is set to the default size of the browser window that the browser creates from a File menu's New Web Browser command. In other words, a new window does not automatically inherit the size of the window making the `window.open()` method call. A new window created via a script is positioned somewhat arbitrarily, unless you use the window positioning attributes available in NN4+ and IE4+. Notice that the position attributes are different for each browser (`screenX` and `screenY` for NN; `left` and `top` for IE). You can include both sets of attributes in a single parameter string because the browser ignores attributes it doesn't recognize.

Invoking `window.open()` via a window's `onload` and `onunload` event handlers has led to severe abuse in the form of unwanted pop-up advertising windows. Browsers that include "pop-up blockers" (such as Mozilla-based browsers) prevent the method from being invoked by these event handlers. With more browsers and users employing pop-up blockers every day, you should not even think about blasting pop-up ads to Web surfers.

Netscape-only signed scripts

Many NN-specific attributes are deemed to be security risks and thus require signed scripts and the user's permission before they are recognized. If the user fails to grant permission, the secure parameter is ignored.

A couple of these attributes have different behaviors on different operating system platforms, due to the way the systems manage their application windows. For example, the `alwaysLowered`, `alwaysRaised`, and `z-locked` styles can exist in layers that range behind

Navigator's own windows in the Windows platform; on the Mac, however, such windows are confined to the levels occupied by Navigator. The difference is that Windows allows windows from multiple applications to interleave each other, while the Mac keeps each application's windows in contiguous layers.

To apply signed scripts to opening a new window with the secure window features, you must enable `UniversalBrowserWrite` privileges as you do for other signed scripts (see Chapter 46 on the CD-ROM). A code fragment that generates an `alwaysRaised` style window follows:

```
<script type="text/javaScript" archive="myJar.jar" id="1">
function newRaisedWindow() {
   netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite");
   var newWindow = window.open("","","height=100,width=300,alwaysRaised=yes");
   netscape.security.PrivilegeManager.disablePrivilege("UniversalBrowserWrite");
   var newContent = "<html><body><b> "On top of spaghetti!"</b>";
   newContent += "<form><center><input type='button' value='OK'";
   newContent += "onclick='self.close()'></center></form></body></html>";
   newWindow.document.write(newContent);
   newWindow.document.close();
}
</script>
```

You can experiment with the look and behavior of new windows with any combination of attributes with the help of the script in Listing 16-25. This page presents a table of all NN-specific new window Boolean attributes and creates a new 300 × 300 pixel window based on your choices. This page assumes that if you are using NN4, you have codebase principals turned on for signed scripts (see Chapter 46 on the CD-ROM).

Be careful with turning off the title bar and hotkeys. With the title bar off, the content appears to float in space, because absolutely no borders are displayed. With hotkeys still turned on, you can use Ctrl+W to close this borderless window (except on the Mac, for which the hotkeys are always disabled with the title bar off). This is how you can turn a computer into a kiosk by sizing a window to the screen's dimensions and setting the window options to `"titlebar=no,hotkeys=no,alwaysRaised=yes"`.

Listing 16-25: **New Window Laboratory**

```
<html>
   <head>
      <title>window.open() Options</title>
      <script type="text/javascript">
      var isNav4 = (navigator.appName == "Netscape" &&
         navigator.appVersion.charAt(0) >= 4) ? true : false;

      function makeNewWind(form) {
         if (isNav4) {
            netscape.security.PrivilegeManager.enablePrivilege(
               "UniversalBrowserWrite");
         }
         var attr = "width=300,height=300";
         for (var i = 0; i < form.elements.length; i++) {
            if (form.elements[i].type == "checkbox") {
               attr += "," + form.elements[i].name + "=";
               attr += (form.elements[i].checked) ? "yes" : "no";
            }
         }
         var newWind = window.open("bofright.htm","subwindow",attr);
```

```
         if (isNav4) {
            netscape.security.PrivilegeManager.revertPrivilege("CanvasAccess");
         }
      }
      </script>
   </head>
   <body>
      <b>Select new window options:</b>
      <form>
         <table border="2">
            <tr>
               <td colspan="2" bgcolor="yellow" align="middle">
               All Browsers Features:</td>
            </tr>
            <tr>
               <td><input type="checkbox" name="toolbar" />toolbar</td>
               <td><input type="checkbox" name="location" />location</td>
            </tr>
            <tr>
               <td><input type="checkbox" name="directories" />directories</td>
               <td><input type="checkbox" name="status" />status</td>
            </tr>
            <tr>
               <td><input type="checkbox" name="menubar" />menubar</td>
               <td><input type="checkbox" name="scrollbars" />scrollbars</td>
            </tr>
            <tr>
               <td><input type="checkbox" name="resizable" />resizable</td>
               <td><input type="checkbox" name="copyhistory" />copyhistory</td>
            </tr>
            <tr>
               <td colspan="2" bgcolor="yellow" align="middle">
               Communicator Features:</td>
            </tr>
            <tr>
               <td><input type="checkbox" name="alwaysLowered" />
               alwaysLowered</td>
               <td><input type="checkbox" name="alwaysRaised" />
               alwaysRaised</td>
            </tr>
            <tr>
               <td><input type="checkbox" name="dependent" />dependent</td>
               <td><input type="checkbox" name="hotkeys" checked="checked" />
               hotkeys</td>
            </tr>
            <tr>
               <td><input type="checkbox" name="titlebar" checked="checked" />
               titlebar</td>
               <td><input type="checkbox" name="z-lock" />z-lock</td>
            </tr>
            <tr>
               <td colspan="2" align="middle"><input type="button"
               name="forAll" value="Make New Window"
               onclick="makeNewWind(this.form)" /></td>
            </tr>
         </table>
      </form>
      <br />
   </body>
</html>
```

Specifying a window name

Getting back to the other parameters of `window.open()`, the second parameter is the name for the new window. Don't confuse this parameter with the document's title, which would normally be set by whatever HTML text determines the content of the window. A window name must be the same style of one-word identifier that you use for other object names and variables. This name is also an entirely different entity than the `window` object that the `open()` method returns. You don't use the name in your scripts. At most, the name can be used for `target` attributes of links and forms.

Loading content into a new window

A script generally populates a window with one of two kinds of information:

- ✦ An existing HTML document whose URL is known beforehand
- ✦ An HTML page created on the fly

To create a new window that displays an existing HTML document, supply the URL as the first parameter of the `window.open()` method. If your page is having difficulty loading a URL into a new page, try specifying the complete URL of the target document (instead of just the filename).

Leaving the first parameter as an empty string forces the window to open with a blank document, ready to have HTML written to it by your script (or loaded separately by another statement that sets that window's location to a specific URL). If you plan to write the content of the window on the fly, assemble your HTML content as one long string value and then use the `document.write()` method to post that content to the new window. If you plan to append no further writing to the page, also include a `document.close()` method at the end to tell the browser that you're finished with the layout (so that the `Layout:Complete` or `Done` message appears in the status bar, if your new window has one).

A call to the `window.open()` method returns a reference to the new window's object if the window opens successfully. This value is vitally important if your script needs to address elements of that new window (such as when writing to its document).

To allow other functions in your script to reference the subwindow, you should assign the result of a `window.open()` method to a global variable. Before writing to the new window the first time, test the variable to make sure that it is not a `null` value — the window may have failed to open because of low memory, for instance. If everything is okay, you can use that variable as the beginning of a reference to any property or object within the new window. For example:

```
var newWindow
...
function createNewWindow() {
   newWindow = window.open("","");
   if (newWindow != null) {
      newWindow.document.write("<html><head><title>Hi!</title></head>");
   }
}
```

That global variable reference continues to be available for another function that perhaps closes the subwindow (via the `close()` method).

When scripts in the subwindow need to communicate with objects and scripts in the originating window, you must make sure that the subwindow has an `opener` property if the level of JavaScript in the visitor's browser doesn't automatically supply one. See the discussion about the `window.opener` property earlier in this chapter.

Invoking multiple window.open() methods with the same window name parameter (the second parameter) does not create additional copies of that window in Netscape browsers (although it does in Internet Explorer). JavaScript prevents you from creating two windows with the same name. Also be aware that a window.open() method does not bring an existing window of that name to the front of the window layers: Use window.focus() for that.

Internet Explorer idiosyncracies

Creating subwindows in IE can be complicated at times by undesirable behavior by the browser. One of the most common problems occurs when you attempt to use document.write() to put content into a newly created window. IE, including some of the latest versions, fails to complete the window opening job before the script statement that uses document.write() executes. This causes a script error because the reference to the subwindow is not yet valid. To work around this, you should put the HTML assembly and document.write() statements in a separate function that gets invoked via a setTimeout() method after the window is created. You can see an example of this in Listing 16-26.

Another problem that affects IE is the occasional security violation ("access denied") warning when a script attempts to access a subwindow. This problem goes away when the page that includes the script for opening and accessing the subwindow is served from an http server, rather than accessed from a local hard disk.

Example

The page rendered by Listing 16-26 displays a single button that generates a new window of a specific size that has only the status bar turned on. The script here shows all the elements necessary to create a new window that has all the right stuff on most platforms. The new window object reference is assigned to a global variable, newWindow. Before a new window is generated, the script looks to see if the window has never been generated before (in which case newWindow would be null) or, for newer browsers, the window is closed. If either condition is true, the window is created with the open() method. Otherwise, the existing window is brought forward with the focus() method (NN3+ and IE4+).

Due to the timing problem that afflicts all IE generations, the HTML assembly and writing to the new window is separated into its own function that is invoked after a 50-millisecond delay (other browsers go along for the ride even if they could accommodate the assembly and writing without the delay). To build the string that is eventually written to the document, I use the += (add-by-value) operator, which appends the string on the right side of the operator to the string stored in the variable on the left side. In this example, the new window is handed an <h1>-level line of text to display.

Listing 16-26: **Creating a New Window**

```
<html>
   <head>
      <title>New Window</title>
      <script type="text/javascript">
      var newWindow;

      function makeNewWindow() {
         if (!newWindow || newWindow.closed) {
            newWindow = window.open("","","status,height=200,width=300");
            // force small delay for IE to catch up
            setTimeout("writeToWindow()", 50);
```

Continued

Listing 16-26 *(continued)*

```
         } else {
            // window's already open; bring to front
            newWindow.focus();
         }
      }

      function writeToWindow() {
         // assemble content for new window
         var newContent = "<html><head><title>One Sub Window<\/title><\/head>";
         newContent += "<body><h1>This window is brand new.<\/h1>";
         newContent += "<\/body><\/html>";
         // write HTML to new window document
         newWindow.document.write(newContent);
         newWindow.document.close();   // close layout stream
      }
      </script>
   </head>
   <body>
      <form>
         <input type="button" name="newOne" value="Create New Window"
         onclick="makeNewWindow()" />
      </form>
   </body>
</html>
```

Related Items: `window.close()`, `window.blur()`, `window.focus()` methods; `window.closed` property.

print()

Returns: Nothing.
Compatibility: WinIE5+, MacIE5+, NN4+, Moz1+, Safari-

The `print()` method provides a scripted way of sending the window or a frame from a frameset to the printer. In all cases, the Print dialog box appears for the user to make the typical printer choices when printing manually. This prevents a rogue `print()` command from tying up a printer without the user's permission.

The precise behavior of the `print()` method varies a bit with the ways different browsers (not to mention operating systems) handle printing. In NN4+ (except for the Windows OS), you can print all frames of a frameset in one `print()` command when it is invoked for the framesetting (parent) document. NN4 for Windows, however, does not print the entire frameset at once. You can write a script that iterates through all frames and prints them with delays to let the content be sent to the print spooler:

```
function printFrames(n) {
   parent.frames[n++].print();
   if (n < parent.frames.length) {
      setTimeout("printFrames(" + n + ")",5000);
   }
}
```

Invoke this function as `printFrames(0)`, and the function does the rest.

In IE5+, the print dialog box gives the user the choice of printing just one frame or all of the frames. Make sure that the `print()` method is invoked for the desired frame when you want only that frame to print. The browser defaults to printing just that frame.

WinIE5 introduced some print-specific event handlers that are triggered by scripted printing as well as manual printing. The events begin to fire after the user has accepted the Print dialog box. An `onbeforeprint` event handler can be used to show content that might be hidden from view but should appear in the printout. After the content has been sent to the print spooler, the `onafterprint` event can restore the page.

Example

Listing 16-27 is a frameset that loads Listing 16-28 into the top frame and a copy of the Bill of Rights into the bottom frame.

Listing 16-27: **Print Example Frameset**

```
<html>
   <head>
      <title>window.print() method</title>
   </head>
   <frameset rows="25%,75%">
      <frame name="controls" src="lst16-28.htm" />
      <frame name="display" src="bofright.htm" />
   </frameset>
</html>
```

Two buttons in the top control panel (see Listing 16-28) let you print the whole frameset (in those browsers and OSes that support it) or just the lower frame. To print the entire frameset, the reference includes the parent window; to print the lower frame, the reference is directed at the `parent.display` frame.

Listing 16-28: **Printing Control**

```
<html>
   <head>
      <title>Print()</title>
   </head>
   <body>
      <form>
         <input type="button" name="printWhole" value="Print Entire Frameset"
         onclick="parent.print()" />
         <p><input type="button" name="printFrame"
            value="Print Bottom Frame Only"
            onclick="parent.display.print()" /></p>
      </form>
   </body>
</html>
```

If you don't like some facet of the printed output, blame the browser's print engine, and not JavaScript. The `print()` method merely invokes the browser's regular printing routines.

Related Items: `window.back()`, `window.forward()`, `window.home()`, `window.find()` methods.

prompt("*message*", "*defaultReply*")

Returns: String of text entered by user or `null`.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The third kind of dialog box that JavaScript can display includes a message from the script author, a field for user entry, and two buttons (OK and Cancel, or Yes and No on Mac versions of Navigator 2 and 3). The script writer can supply a prewritten answer so that a user confronted with a prompt dialog box can click OK (or press Enter) to accept that answer without further typing. Supplying both parameters to the `window.prompt()` method is important. Even if you don't want to supply a default answer, enter an empty string as the second parameter:

```
prompt("What is your postal code?","");
```

If you omit the second parameter, JavaScript inserts the string `undefined` into the dialog box's field. This string is disconcerting to most Web page visitors.

The value returned by this method is a string in the dialog box's field when the user clicks the OK button. If you're asking the user to enter a number, remember that the value returned by this method is a string. You may need to perform data-type conversion with the `parseInt()` or `parseFloat()` functions (see Chapter 34) to use the returned values in math calculations.

When the user clicks the prompt dialog box's OK button without entering any text into a blank field, the returned value is an empty string (""). Clicking on the Cancel button, however, makes the method return a `null` value. Therefore, the scripter must test for the type of returned value to make sure that the user entered some data that can be processed later in the script, as in

```
var entry = prompt("Enter a number between 1 and 10:","");
if (entry != null) {
   //statements to execute with the value
}
```

This script excerpt assigns the results of the prompt dialog box to a variable and executes the nested statements if the returned value of the dialog box is not null (if the user clicked the OK button). The rest of the statements then include data validation to make sure that the entry is a number within the desired range (see Chapter 43 on the CD-ROM).

It may be tempting to use the prompt dialog box as a handy user input device. But, as with the other JavaScript dialog boxes, the modality of the prompt dialog box is disruptive to the user's flow through a document and can also trap automated macros that some users activate to capture Web sites. In forms, HTML fields are better user interface elements for attracting user text entry. Perhaps the safest way to use a prompt dialog box is to have it appear when a user clicks a button element on a page — and then only if the information you require of the user can be provided in a single prompt dialog box. Presenting a sequence of prompt dialog boxes is downright annoying to users.

Example

The function that receives values from the prompt dialog box in Listing 16-29 (see the dialog box in Figure 16-11) does some data-entry validation (but certainly not enough for a commercial site). The function first checks to make sure that the returned value is neither `null`

(Cancel) nor an empty string (the user clicked OK without entering any values). See Chapter 43 on the CD-ROM for more about data-entry validation.

Listing 16-29: **The Prompt Dialog Box**

```
<html>
   <head>
      <title>window.prompt() Method</title>
      <script type="text/javascript">
      function populateTable() {
         var howMany = prompt("Fill in table for how many factors?","");
         if (howMany != null && howMany != "") {
            alert("Filling the table for " + howMany);   // for demo
            //statements that validate the entry and
            //actually populate the fields of the table
         }
      }
      </script>
   </head>
   <body>
      <form>
         <!-- other statements that display and populate a large table -->
         <input type="button" name="fill" value="Fill Table..."
         onclick="populateTable()" />
      </form>
   </body>
</html>
```

Figure 16-11: The prompt dialog box displayed from Listing 16-29 in WinIE.

Notice one important user interface element in Listing 16-29. Because clicking the button leads to a dialog box that requires more information from the user, the button's label ends in an ellipsis (or, rather, three periods acting as an ellipsis character). The ellipsis is a common courtesy to let users know that a user interface element leads to a dialog box of some sort. Consistent with standalone applications, the user should be able to cancel out of that dialog box and return to the same screen state that existed before the button was clicked.

Related Items: `window.alert()`, `window.confirm()` method.

`releaseEvents(`*`eventTypeList`*`)`

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

If your scripts have enabled NN4-specific event capture for the `window` object (or `document` or layer, for that matter), you can turn off that capture with the `releaseEvents()` method.

This method does not inhibit events from reaching their intended target. In fact, by releasing capture from a higher object, released events don't bother stopping at those higher objects anymore. Parameters for the `releaseEvents()` method are one or more event types. Each event type is its own entity, so if your window captures three event types at one point, you can release some or all of those event types as the visitor interacts with your page. For example, if the page loads and captures three types of events, as in

```
window.captureEvents(Event.CLICK | Event.KEYPRESS | Event.CHANGE);
```

you can later turn off window event capture for all but the `click` event:

```
window.releaseEvents(Event.KEYPRESS | Event.CHANGE);
```

The window still captures and processes `click` events, but `keypress` and `change` events go directly to their target objects.

An improved mechanism (removing an event listener) is implemented in NN6+/W3C browsers based on the W3C event model. See Chapters 14 and 25 for more information.

Related Items: `window.captureEvents()`, `window.routeEvent()` methods.

resizeBy(*deltaX,deltaY*)
resizeTo(*outerwidth,outerheight*)

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN4, Moz-, Safari1+

Starting with version 4 browsers, scripts can control the size of the current browser window on the fly (no longer available in Mozilla-based browsers). While you can set the individual inner and (in NN4) outer width and height properties of a window, the `resizeBy()` and `resizeTo()` methods let you adjust both axis measurements in one statement. In both instances, all adjustments affect the lower-right corner of the window: To move the top-left corner, use the `window.moveBy()` or `window.moveTo()` methods.

Each resize method requires a different kind of parameter. The `resizeBy()` method adjusts the window by a certain number of pixels along one or both axes. Therefore, it is not concerned with the specific size of the window beforehand — only by how much each axis is to change. For example, to increase the current window size by 100 pixels horizontally and 50 pixels vertically, the statement is

```
window.resizeBy(100, 50);
```

Both parameters are required, but if you only want to adjust the size in one direction, set the other to zero. You may also shrink the window by using negative values for either or both parameters.

You find a greater need for the `resizeTo()` method, especially when you know that on a particular platform the window needs adjustment to a specific width and height to best accommodate that platform's display of form elements. Parameters for the `resizeTo()` method are the actual pixel width and height of the outer dimension of the window — the same as NN4's `window.outerWidth` and `window.outerHeight` properties.

To resize the window such that it occupies all screen real estate (except for the Windows Taskbar and Macintosh menu bar), use the `screen` object properties that calculate the available screen space:

```
window.resizeBy(screen.availWidth, screen.availHeight);
```

This action, however, is not precisely the same in Windows as maximizing the window. To achieve that same effect, you must move the window to coordinates -4, -4 and add eight to the two parameters of `resizeBy()`:

```
window.moveTo(-4,-4);
window.resizeTo(screen.availWidth + 8, screen.availHeight + 8);
```

This hides the window's own 4-pixel wide border, as occurs during OS-induced window maximizing. See also the `screen` object discussion (see Chapter 38 on the CD-ROM) for more OS-specific details.

On some platforms, the dimensions are applied to the inner width and height, rather than outer. If a specific outer size is necessary, use the NN-specific `window.outerHeight` and `window.outerWidth` properties instead.

Navigator 4 imposes some security restrictions for maximum and minimum size for a window. For both methods, you are limited to the viewable area of the screen and visible minimums unless the page uses signed scripts (see Chapter 46 on the CD-ROM). With signed scripts and the user's permission, for example, you can adjust windows beyond the available screen borders.

Example

You can experiment with the resize methods with the page in Listing 16-30. Two parts of a form let you enter values for each method. The one for `window.resize()` also lets you enter a number of repetitions to better see the impact of the values. Enter zero and negative values to see how those affect the method. Also test the limits of different browsers.

Listing 16-30: **Window Resize Methods**

```
<html>
   <head>
      <title>Window Resize Methods</title>
      <script type="text/javascript">
      function doResizeBy(form) {
         var x = parseInt(form.resizeByX.value);
         var y = parseInt(form.resizeByY.value);
         var count = parseInt(form.count.value);
         for (var i = 0; i < count; i++) {
            window.resizeBy(x, y);
         }
      }
      function doResizeTo(form) {
         var x = parseInt(form.resizeToX.value);
         var y = parseInt(form.resizeToY.value);
         window.resizeTo(x, y);
      }
      </script>
   </head>
   <body>
      <form>
         <b>Enter the x and y increment, plus how many times the window should
         be resized by these increments:</b><br />
         Horiz:<input type="text" name="resizeByX" size="4" /> Vert:<input
         type="text" name="resizeByY" size="4" /> How Many:<input type="text"
         name="count" size="4" /> <input type="button" name="ResizeBy"
```

Continued

Listing 16-30 *(continued)*

```
        value="Show resizeBy()" onclick="doResizeBy(this.form)" />
        <hr />
        <b>Enter the desired width and height of the current window:</b><br />
        Width:<input type="text" name="resizeToX" size="4" /> Height:<input
        type="text" name="resizeToY" size="4" /> <input type="button"
        name="ResizeTo" value="Show resizeTo()"
        onclick="doResizeTo(this.form)" />
     </form>
   </body>
</html>
```

Related Items: `window.outerHeight`, `window.outerWidth` properties; `window.moveTo()`, `window.sizeToContent()` methods.

`routeEvent(event)`

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

If you turn on NN4-specific event capturing in the `window`, `document`, or layer objects (via their respective `captureEvents()` methods), the handlers you assign to those events really capture those events, preventing them from ever reaching their intended targets. For some page designs, this is intentional, because it allows the higher-level object to handle all events of a particular type. But if your goal is to perform some preprocessing of events before they reach their destination, you need a way to pass that event along its regular path. That's what the `routeEvent()` method is for.

Perhaps a more common reason for capturing events at the window (or similar) level is to look for special cases, such as when someone Ctrl+clicks on an element. In this case, even though the window event handler receives all `click` events, it performs further processing only when the `event.modifiers` property indicates the Ctrl key is also pressed and the *`eventObj`*`.target` property reveals the item being clicked is a link rather than a button. All other instances of the `click` event are routed on their way to their destinations. The `event` object knows where it's going, so that your `routeEvent()` method doesn't have to worry about that.

The parameter for the `routeEvent()` method is the event object that is passed to the function that processes the high-level event, as shown here:

```
function flashRed(evt) {
   [statements that filter specific events to flash background color red]
   routeEvent(evt);
}
```

The event object, `evt`, comes into the function while passing unmodified to the object that was clicked.

In the W3C DOM event model (as implemented in NN6+/Moz1+/Safari1+), a captured event continues onward to the target after event handlers higher up the containment chain finish their work.

Example

The `window.routeEvent()` method is used in the example for `window.captureEvents()`, Listing 16-21.

Related Items: `window.captureEvents()`, `window.releaseEvents()`, `window.handleEvent()` methods; `event` object (Chapter 25).

scroll(*horizontalCoord, verticalCoord*)

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

The `window.scroll()` method was introduced in NN3 and has been implemented in all scriptable browsers since then. But in the meantime, the method has been replaced by the `window.scrollTo()` method, which is in more syntactic alliance with many other window methods. Use the `window.scroll()` method only if your audience is still using NN3; for an audience of NN4+ and IE4+, use the `window.scrollTo()` method instead.

The `window.scroll()` method takes two parameters, the horizontal (x) and vertical (y) coordinates of the document that is to be positioned at the top-left corner of the window or frame. You must realize that the window and document have two similar, but independent, coordinate schemes. From the window's point of view, the top-left pixel (of the content area) is point 0,0. All documents also have a 0,0 point: the very top-left of the document. The window's 0,0 point doesn't move, but the document's 0,0 point can move — via manual or scripted scrolling. Although `scroll()` is a window method, it seems to behave more like a document method, as the document appears to reposition itself within the window. Conversely, you can also think of the window moving to bring its 0,0 point to the designated coordinate of the document.

Although you can set values beyond the maximum size of the document or to negative values, the results vary from platform to platform. For the moment, the best usage of the `window.scroll()` method is as a means of adjusting the scroll to the very top of a document (`window.scroll(0,0)`) when you want the user to be at a base location in the document. For vertical scrolling within a text-heavy document, an HTML anchor may be a better alternative for now (though it doesn't readjust horizontal scrolling).

Example

To demonstrate the `scroll()` method, Listing 16-31 defines a frameset with a document in the top frame (see Listing 16-32) and a control panel in the bottom frame (see Listing 16-33). A series of buttons and text fields in the control panel frame directs the scrolling of the document. I've selected an arbitrary, large GIF image to use in the example. To see results of some horizontal scrolling values, you may need to shrink the width of the browser window until a horizontal scrollbar appears in the top frame.

Listing 16-31: A Frameset for the scroll() Demonstration

```
<html>
   <head>
      <title>
         window.scroll() Method
      </title>
   </head>
   <frameset rows="50%,50%">
      <frame src="lst16-32.htm" name="display" />
      <frame src="lst16-33.htm" name="control" />
   </frameset>
</html>
```

Listing 16-32: **The Image to Be Scrolled**

```
<html>
   <head>
      <title>
         Arch
      </title>
   </head>
   <body>
      <h1>
         A Picture is Worth...
      </h1>
      <hr />
      <center>
         <table border="3">
            <caption align="bottom">
               A Splendid Arch
            </caption>
            <tr>
               <td>
                  <img alt="image" src="arch.gif" />
               </td>
            </tr>
         </table>
      </center>
   </body>
</html>
```

Listing 16-33: **Controls to Adjust Scrolling of the Upper Frame**

```
<html>
   <head>
      <title>
         Scroll Controller
      </title>

      <script type="text/javascript">
      function scroll(x,y) {
         parent.frames[0].scroll(x,y);
      }
      function customScroll(form) {
         parent.frames[0].scroll(parseInt(form.x.value),
            parseInt(form.y.value));
      }
      </script>
   </head>
   <body>
      <h2>
         Scroll Controller
      </h2>
      <hr />
      <form name="fixed">
         Click on a scroll coordinate for the upper frame:
         <p>
```

```
            <input type="button" value="0,0" onclick="scroll(0,0)" /> <input
            type="button" value="0,100" onclick="scroll(0,100)" /> <input
            type="button" value="100,0" onclick="scroll(100,0)" />
         </p>
         <p>
            <input type="button" value="-100,100" onclick="scroll(-100,100)" />
            <input type="button" value="20,200" onclick="scroll(20,200)" />
            <input type="button" value="1000,3000"
            onclick="scroll(1000,3000)" />
         </p>
      </form>
      <hr />
      <form name="custom">
         Enter a Horizontal <input type="text" name="x" value="0" size="4" />
         and Vertical <input type="text" name="y" value="0" size="4" /> value.
         Then <input type="button" value="click to scroll"
         onclick="customScroll(this.form)" />
      </form>
   </body>
</html>
```

Notice that in the `customScroll()` function, JavaScript must convert the string values from the two text boxes to integers (with the `parseInt()` method) for the `scroll()` method to accept them. Nonnumeric data can produce very odd results. Also be aware that although this example shows how to adjust the scroll values in another frame, you can set such values in the same frame or window as the script, as well as in subwindows, provided that you use the correct object references to the window.

Related Items: `window.scrollBy()`, `window.scrollTo()` methods.

scrollBy(*deltaX*,*deltaY*)
scrollTo(*x*,*y*)

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

Modern browsers provide a related pair of window scrolling methods. The `window.scrollTo()` method is the newer version of the `window.scroll()` method. The two work identically to position a specific coordinate point of a document at the top-left corner of the inner window region.

In contrast, the `window.scrollBy()` method allows for relative positioning of the document. Parameter values indicate by how many pixels the document should scroll in the window (horizontally and vertically). Negative numbers are allowed if you want to scroll to the left and/or upward. The `scrollBy()` method comes in handy if you elect to hide the scrollbars of a window or frame and offer other types of scrolling controls for your users. For example, to scroll down one entire screen of a long document, you can use the `window.innerHeight` (in NN) or `document.body.clientHeight` (in IE) properties to determine what the offset from the current position would be:

```
// assign IE body clientHeight to window.innerHeight
if (document.body && document.body.clientHeight) {
   window.innerHeight = document.body.clientHeight;
}
window.scrollBy(0, window.innerHeight);
```

Unwanted User Scrolling

Many Windows-compatible personal computers ship with a mouse that includes a scroll wheel that is activated by pressing down on the wheel and spinning the wheel. Be aware that even if your page design loads into frames or new windows that intentionally lack scrollbars, the page will be scrollable via this wheel if the document or its background image are larger than the window or frame. Users may not even be aware that they have scrolled the page (because there are no scroll bar visual clues). If this affects your design, you may need to build in a routine (via `setTimeout()`) that periodically sets the scroll of the window to 0,0.

To scroll upward, use a negative value for the second parameter:

```
window.scrollBy(0, -window.innerHeight);
```

Scrolling the document in Macintosh versions of NN4 and IE exhibits some buggy behavior. At times it appears as though you are allowed to scroll well beyond the document edges. In truth, the document has stopped at the border, but the window or frame may not have refreshed properly.

The window scroll methods are not the ones to use to produce the scrolling effect of a positioned element. That kind of animation is accomplished by adjusting `style` position properties (see Chapter 39 on the CD-ROM).

Example

To work with the `scrollTo()` method, you can use Listings 16-31 through 16-33 (the `window.scroll()` method) but substitute `window.scrollTo()` for `window.scroll()`. The results should be the same. For `scrollBy()`, the example starts with the frameset in Listing 16-34. It loads the same content document as the `window.scroll()` example (see Listing 16-32), but the control panel (see Listing 16-35) provides input to experiment with the `scrollBy()` method.

Listing 16-34: Frameset for ScrollBy Controller

```
<html>
   <head>
      <title>
         window.scrollBy() Method
      </title>
   </head>
   <frameset rows="50%,50%">
      <frame src="lst16-32.htm" name="display" />
      <frame src="lst16-35.htm" name="control" />
   </frameset>
</html>
```

Notice in Listing 16-35 that all references to window properties and methods are directed to the `display` frame. String values retrieved from text fields are converted to numbers with the `parseInt()` global function.

Listing 16-35: ScrollBy Controller

```
<html>
   <head>
      <title>
         ScrollBy Controller
      </title>

      <script type="text/javascript">
      function page(direction) {
         var pixFrame = parent.display;
         var deltaY = (pixFrame.innerHeight) ?
            pixFrame.innerHeight : pixFrame.document.body.scrollHeight;
         if (direction == "up") {
            deltaY = -deltaY;
         }
         parent.display.scrollBy(0, deltaY);
      }
      function customScroll(form) {
         parent.display.scrollBy(parseInt(form.x.value),
            parseInt(form.y.value));
      }
      </script>
   </head>
   <body>
      <b>ScrollBy Controller</b>
      <form name="custom">
         Enter an Horizontal increment <input type="text" name="x" value="0"
         size="4" /> and Vertical <input type="text" name="y" value="0"
         size="4" /> value.<br />
         Then <input type="button" value="click to scrollBy()"
         onclick="customScroll(this.form)" />
         <hr />
         <input type="button" value="PageDown" onclick="page('down')" /> <input
         type="button" value="PageUp" onclick="page('up')" />
      </form>
   </body>
</html>
```

Related Items: `window.pageXOffset`, `window.pageYOffset` properties; `window.scroll()` method.

`setInterval("`*`expr`*`", `*`msecDelay`*` [, `*`language`*`])`
`setInterval(`*`funcRef, msecDelay`*` [, `*`funcarg1, ..., funcargn`*`])`

Returns: Interval ID integer.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

It is important to understand the distinction between the `setInterval()` and `setTimeout()` methods. Before the `setInterval()` method was part of JavaScript, authors replicated the behavior with `setTimeout()`, but the task often required reworking scripts a bit.

Use `setInterval()` when your script needs to call a function or execute some expression repeatedly with a fixed time delay between calls to that function or expression. The delay is not at all like a wait state in some languages: Other processing does not halt while the delay is in effect. Typical applications include animation by moving an object around the page under controlled speed (instead of letting the JavaScript interpreter whiz the object through its path at CPU-dependent speeds). In a kiosk application, you can use `setInterval()` to advance "slides" that appear in other frames or as layers, perhaps changing the view every 10 seconds. Clock displays and countdown timers would also be suitable usage of this method (even though you see examples in this book that use the old-fashioned `setTimeout()` way to perform timer and clock functions).

In contrast, `setTimeout()` is best suited for those times when you need to carry out a function or expression one time in the future — even if that future is only a second or two away. See the discussion of the `setTimeout()` method later in this chapter for details on this application.

While the primary functionality of the `setInterval()` method is the same in all browsers, NN and IE offer some extra possibilities depending on the way you use parameters to the method. For simple invocations of this method, the same parameters work in all browsers that support the method. First, I address the parameters that all browsers have in common.

The first parameter of the `setInterval()` method is the name of the function or expression to run after the interval elapses. This item must be a quoted string. If the parameter is a function, no function arguments are allowed inside the function's parentheses unless the arguments are literal strings (but see the next section, "Passing function parameters").

The second parameter of this method is the number of milliseconds (1,000 per second) that JavaScript should use as the interval between invocations of the function or expression. Even though the measure is in extremely small units, don't rely on 100 percent accuracy of the intervals. Various other internal processing delays may throw off the timing just a bit.

Just as with `setTimeout()`, `setInterval()` returns an integer value that is the ID for the interval process. That ID value lets you turn off the process with the `clearInterval()` method. That method takes the ID value as its sole parameter. This mechanism allows for the setting of multiple interval processes running, while giving your scripts the power to stop individual processes at any time without interrupting the others.

IE4+ uses the optional third parameter to specify the scripting language of the statement or function being invoked in the first parameter. As long as you are scripting exclusively in JavaScript (the same as JScript), there is no need to include this parameter.

Passing function parameters

NN4+ provides a mechanism for easily passing evaluated parameters to a function invoked by `setInterval()`. To use this mechanism, the first parameter of `setInterval()` must not be a string, but rather a reference to the function (no trailing parentheses). The second parameter remains the amount of delay. But beginning with the third parameter, you can include evaluated function arguments as a comma-delimited list:

```
intervalID = setInterval(cycleAnimation, 500, "figure1");
```

The function definition receives those parameters in the same form as any function:

```
function cycleAnimation(elemID) {...}
```

For use with a wider range of browsers, you can also cobble together the ability to pass parameters to a function invoked by `setInterval()`. Because the call to the other function is a string expression, you can use computed values as part of the strings via string concatenation. For example, if a function uses event handling to find the element that a user clicked (to initiate some animation sequence), that element's ID, referenced by a variable, can be passed to the function invoked by `setInterval()`:

```
function findAndCycle() {
   var elemID;
   // statements here that examine the event info
   // and extract the ID of the clicked element,
   // assigning that ID to the elemID variable
   intervalID = setInterval("cycleAnimation(" + elemID + ")", 500);
}
```

If you need to pass ever-changing parameters with each invocation of the function from `setInterval()`, look instead to using `setTimeout()` at the end of a function to invoke that very same function again.

Example

The demonstration of the `setInterval()` method entails a two-framed environment. The framesetting document is shown in Listing 16-36.

Listing 16-36: SetInterval() Demonstration Frameset

```
<html>
   <head>
      <title>setInterval() Method</title>
   </head>
   <frameset rows="50%,50%">
      <frame src="lst16-37.htm" name="control" />
      <frame src="bofright.htm" name="display" />
   </frameset>
</html>
```

In the top frame is a control panel with several buttons that control the automatic scrolling of the Bill of Rights text document in the bottom frame. Listing 16-37 shows the control panel document. Many functions here control the interval, scrolling jump size, and direction, and they demonstrate several aspects of applying `setInterval()`.

Notice that in the beginning the script establishes a number of global variables. Three of them are parameters that control the scrolling; the last one is for the ID value returned by the `setInterval()` method. The script needs that value to be a global value so that a separate function can halt the scrolling with the `clearInterval()` method.

All scrolling is performed by the `autoScroll()` function. For the sake of simplicity, all controlling parameters are global variables. In this application, placement of those values in global variables helps the page restart autoscrolling with the same parameters as it had when it last ran.

Listing 16-37: SetInterval() Control Panel

```
<html>
   <head>
      <title>ScrollBy Controller</title>
      <script type="text/javascript">
      var scrollSpeed = 500;
      var scrollJump = 1;
      var scrollDirection = "down";
      var intervalID;

      function autoScroll() {
         if (scrollDirection == "down") {
            scrollJump = Math.abs(scrollJump);
         } else if (scrollDirection == "up" && scrollJump > 0) {
            scrollJump = -scrollJump;
         }
         parent.display.scrollBy(0, scrollJump);
         if (parent.display.pageYOffset <= 0) {
            clearInterval(intervalID);
         }
      }

      function reduceInterval() {
         stopScroll();
         scrollSpeed -= 200;
         startScroll();
      }
      function increaseInterval() {
         stopScroll();
         scrollSpeed += 200;
         startScroll();
      }
      function reduceJump() {
         scrollJump -= 2;
      }
      function increaseJump() {
         scrollJump += 2;
      }
      function swapDirection() {
         scrollDirection = (scrollDirection == "down") ? "up" : "down";
      }
      function startScroll() {
         parent.display.scrollBy(0, scrollJump);
         if (intervalID) {
            clearInterval(intervalID);
         }
         intervalID = setInterval("autoScroll()",scrollSpeed);
      }
      function stopScroll() {
         clearInterval(intervalID);
      }
      </script>
   </head>
   <body onload="startScroll()">
      <b>AutoScroll by setInterval() Controller</b>
      <form name="custom">
         <input type="button" value="Start Scrolling"
```

```
            onclick="startScroll()" /> <input type="button" value="Stop Scrolling"
            onclick="stopScroll()" />
            <p><input type="button" value="Shorter Time Interval"
               onclick="reduceInterval()" /> <input type="button"
               value="Longer Time Interval" onclick="increaseInterval()" /></p>
            <p><input type="button" value="Bigger Scroll Jumps"
               onclick="increaseJump()" /> <input type="button"
               value="Smaller Scroll Jumps" onclick="reduceJump()" /></p>
            <p><input type="button" value="Change Direction"
               onclick="swapDirection()" /></p>
         </form>
      </body>
   </html>
```

The `setInterval()` method is invoked inside the `startScroll()` function. This function initially "burps" the page by one `scrollJump` interval so that the test in `autoScroll()` for the page being scrolled all the way to the top doesn't halt a page from scrolling before it gets started. Notice, too, that the function checks for the existence of an interval ID. If one is there, it is cleared before the new one is set. This is crucial within the design of the example page, because repeated clicking of the Start Scrolling button triggers multiple interval timers inside the browser. Only the most recent one's ID would be stored in `intervalID`, allowing no way to clear the older ones. But this little side trip makes sure that only one interval timer is running. One of the global variables, `scrollSpeed`, is used to fill the delay parameter for `setInterval()`. To change this value on the fly, the script must stop the current interval process, change the `scrollSpeed` value, and start a new process.

The intensely repetitive nature of this application is nicely handled by the `setInterval()` method.

Related Items: `window.clearInterval()`, `window.setTimeout()` methods.

setTimeout("*expr*", *msecDelay* [, *language*])
setTimeout(*functionRef*, *msecDelay* [, *funcarg1*, ..., *funcargn*])

Returns: ID value for use with `window.clearTimeout()` method.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The name of this method may be misleading, especially if you have done other kinds of programming involving timeouts. In JavaScript, a *timeout* is an amount of time (in milliseconds) before a stated expression evaluates. A timeout is not a wait or script delay, but rather a way to tell JavaScript to hold off executing a statement or function for a desired amount of time. Other statements following the one containing `setTimeout()` execute immediately.

Say that you have a Web page designed to enable users to interact with a variety of buttons or fields within a time limit (this is a Web page running at a free-standing kiosk). You can turn on the timeout of the window so that if no interaction occurs with specific buttons or fields lower in the document after, say, two minutes (120,000 milliseconds), the window reverts to the top of the document or to a help screen. To tell the window to switch off the timeout after a user does navigate within the allotted time, you need to have any button that the user interacts with call the other side of a `setTimeout()` method — the `clearTimeout()` method — to cancel the current timer. (The `clearTimeout()` method is explained earlier in this chapter.) Multiple timers can run concurrently and are completely independent of each other.

While the primary functionality of the `setTimeout()` method is the same in all browsers, NN and IE offer some extra possibilities depending on the way you use parameters to the method. For simple invocations of this method, the same parameters work in all browsers that support the method. I first address the parameters that all browsers have in common.

The expression that comprises the first parameter of the method `window.setTimeout()` is a quoted string that can contain either a call to any function or method or a standalone JavaScript statement. The expression evaluates after the time limit expires.

Understanding that this timeout does not halt script execution is very important. In fact, if you use a `setTimeout()` method in the middle of a script, the succeeding statements in the script execute immediately; after the delay time, the expression in the `setTimeout()` method executes. Therefore, I've found that the best way to design a timeout in a script is to plug it in as the last statement of a function: Let all other statements execute and then let the `setTimeout()` method appear to halt further execution until the timer goes off. In truth, however, although the timeout is "holding," the user is not prevented from performing other tasks. And after a timeout timer is ticking, you cannot adjust its time. Instead, clear the timeout and start a new one.

If you need to use `setTimeout()` as a delay inside a function, break the function into two parts, using the `setTimeout()` method as a bridge between the two functions. You can see an example of this in Listing 16-26, where IE needs a little delay to finish opening a new window before content can be written for it. If it weren't for the required delay, the HTML assembly and writing would have been accomplished in the same function that opens the new window.

It is not uncommon for a `setTimeout()` method to invoke the very function in which it lives. For example, if you have written a Java applet to perform some extra work for your page and you need to connect to it via LiveConnect, your scripts must wait for the applet to load and carry out its initializations. Although an `onload` event handler in the document ensures that the applet object is visible to scripts, it doesn't know whether the applet has finished its initializations. A JavaScript function that inspects the applet for a clue might need to poll the applet every 500 milliseconds until the applet sets some internal value indicating all is ready, as shown here:

```
var t;
function autoReport() {
   if (!document.myApplet.done) {
      t = setTimeout("autoReport()",500);
   } else {
      clearTimeout(t);
      // more statements using applet data //
   }
}
```

JavaScript provides no built-in equivalent for a `wait` command. The worst alternative is to devise a looping function of your own to trap script execution for a fixed amount of time. Unfortunately, this approach prevents other processes from being carried out, so you should consider reworking your code to rely on a `setTimeout()` method instead.

NN4+ provides a mechanism for passing parameters to functions invoked by `setTimeout()`. See the section "Passing function parameters" in the discussion of `window.setInterval()` for details on this and passing parameters in other browser versions.

As a note to experienced programmers, neither `setInterval()` nor `setTimeout()` spawn new threads in which to run their invoked scripts. When the timer expires and invokes a function, the process gets at the end of the queue of any pending script processing in the JavaScript execution thread.

Example

When you load the HTML page in Listing 16-38, it triggers the `updateTime()` function, which displays the time (in hh:mm am/pm format) in the status bar. Instead of showing the seconds incrementing one by one (which may be distracting to someone trying to read the page), this function alternates the last character of the display between an asterisk and nothing, like a visual "heartbeat."

Listing 16-38: **Display the Current Time**

```
<html>
   <head>
      <title>Status Bar Clock</title>
      <script type="text/javascript">
      var flasher = false;
      // calculate current time, determine flasher state,
      // and insert time into status bar every second
      function updateTime() {
         var now = new Date();
         var theHour = now.getHours();
         var theMin = now.getMinutes();
         var theTime = "" + ((theHour > 12) ? theHour - 12 : theHour);
         theTime += ((theMin < 10) ? ":0" : ":") + theMin;
         theTime  += (theHour >= 12) ? " pm" : " am";
         theTime += ((flasher) ? " " : "*");
         flasher = !flasher;
         window.status = theTime;
         // recursively call this function every second to keep timer going
         timerID = setTimeout("updateTime()",1000);
      }
      </script>
   </head>
   <body onload="updateTime()">
   </body>
</html>
```

In this function, the `setTimeout()` method works in the following way: Once the current time (including the flasher status) appears in the statusbar, the function waits approximately one second (1,000 milliseconds) before calling the same function again. You don't have to clear the `timerID` value in this application because JavaScript does it for you every time the 1,000 milliseconds elapse.

A logical question to ask is whether this application should be using `setInterval()` instead of `setTimeout()`. This is a case in which either one does the job. To use `setInterval()` here would require that the interval process start outside of the `updateTime()` function, because you need only one process running that repeatedly calls `updateTime()`. It would be a cleaner implementation in that regard, instead of the tons of timeout processes spawned by Listing 16-38. On the other hand, the application would not run in any browsers before NN4 or IE4, as Listing 16-38 does.

To demonstrate passing parameters, you can modify the `updateTime()` function to add the number of times it gets invoked to the display in the status bar. For that to work, the function must have a parameter variable so that it can catch a new value each time it is invoked by `setTimeout()`'s expression. For all browsers, the function would be modified as follows (unchanged lines are represented by the ellipsis):

```
function updateTime(i) {
   ...
   window.status = theTime + "  (" + i + ")";
   // pass updated counter value with next call to this function
   timerID = setTimeout("updateTime(" + i+1 + ")",1000);
}
```

If you were running this exclusively in NN4+, which is likely, you could use its more convenient way of passing parameters to the function:

```
timerID = setTimeout(updateTime,1000, i+1);
```

In either case, the `onload` event handler would also have to be modified to get the ball rolling with an initial parameter:

```
onload = "updateTime(0)";
```

Related Items: `window.clearTimeout()`, `window.setInterval()`, `window.clearInterval()` methods.

showHelp("*URL*",["*contextID*"])

Returns: Nothing.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The IE-specific `showHelp()` method lets a script open a Winhelp window with a particular `.hlp` file. This method is specific to Win32 operating systems.

If your Winhelp file has context identifiers specified in various places, you can pass the ID as an optional second parameter. This lets the call to `showHelp()` navigate to a particular area of the `.hlp` file that applies to a specific element on the page.

See the Microsoft Visual Studio authoring environment for details on building Winhelp files.

showModalDialog("*URL*"[, *arguments*][, *features*])
showModelessDialog("*URL*"[, *arguments*][, *features*])

Returns: `returnValue` (modal) or `window` object (modeless).
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

IE4+ provides methods for opening a modal dialog box window, which always stays in front of the main browser window while making the main window inaccessible to the user. In WinIE5, Microsoft added the modeless type of dialog box, which also stays in front, but allows user access to whatever can be seen in the main window. You can load any HTML page or image that you like into the dialog box window, by providing a URL as the first parameter. Optional parameters let you pass data to a dialog box and give you considerable control over the look of the window. Unfortunately, these types of dialog box windows are not available in Navigator. At best, you can simulate modal and modeless dialog box windows, but the job is not for beginners.

The windows generated by both methods are (almost) full-fledged `window` objects with some extra properties that are useful for what these windows are intended to do. Perhaps the most important property is the `window.dialogArgument` property. This property lets a script read the data that is passed to the window via the second parameter of both `showModalDialog()` and `showModelessDialog()`. Passed data can be in any valid JavaScript data type, including objects and arrays.

Displaying a modal dialog box has some ramifications for scripts. In particular, script execution in the main window halts at the statement that invokes the `showModalDialog()` method as long as the modal dialog box remains visible. Scripts are free to run in the dialog box window during this time. The instant the user closes the dialog box, execution resumes in the main window. A call to show a modeless dialog box, on the other hand, does not halt processing because scripts in the main page or dialog box window are allowed to communicate "live" with the other window.

Retrieving dialog data

To send data back to the main window's script from a modal dialog box window, a script in the dialog box window can set the `window.returnValue` property to any JavaScript value. It is this value that gets assigned to the variable receiving the returned value from the `setModelDialog()` method, as shown in the following example:

```
var specifications = window.showModalDialog("preferences.html");
```

The makeup and content of the returned data is in the hands of your scripts. No data is automatically returned for you.

Because a modeless dialog box coexists with your live main page window, returning data is not as straightforward as for a modal dialog box. The second parameter of the `showModelessDialog()` method takes on a special task that isn't exactly the same as passing parameters to the dialog box. Instead, if you define a global variable or a function in the main window's script, pass a reference to that variable or function as the second parameter to display the modeless dialog box. A script in the modeless dialog box can then point to that reference as the way to send data back to the main window before the dialog box closes (or when a user clicks something, such as an Apply button). This mechanism even allows for passing data back to a function in the main window. For example, say that the main window has a function defined as the following:

```
function receivePrefsDialogData(a, b, c) {
   // statements to process incoming values //
}
```

Then pass a reference to this function when opening the window:

```
dlog = showModelessDialog("prefs.html", receivePrefsDialogData);
```

A script statement in the dialog box window's document can pick up that reference so that other statements can use it, such as a function for an Apply button's `onclick` event handler:

```
var returnFunc = window.dialogArguments;
...
function apply(form) {
   returnFunc(form.color.value, form.style.value, form.size.value);
}
```

While this approach seems to block ways of getting parameters to the dialog box when it opens, you can always reference the dialog box in the main window's script and set form or variable values directly:

```
dlog = showModelessDialog("prefs.html", receivePrefsDialogData);
dlog.document.forms[0].userName.value = GetCookie("userName");
```

Be aware that a dialog box window opened with either of these methods does not maintain a connection to the originating window via the `opener` property. The `opener` property for both dialog box types is undefined.

Dialog window features

Both methods provide an optional third property that lets you specify visible features of the dialog box window. Omitting the property sets all features to their default values. All parameters are to be contained by a single string, and each parameter's name-value pair is in the form of CSS `attribute:value` syntax. Table 16-4 lists all of the window features available for the two window styles. If you are designing for compatibility with IE4, you are restricted to the modal dialog box and a subset of features, as noted in the table. All values listed as Boolean take only the following four values: `yes`, `no`, `1`, `0`.

Table 16-4: IE Dialog Box Window Features

Feature	*Type*	*Default*	*Description*
`center`	Boolean	`yes`	Whether to center dialog box (overridden by `dialogLeft` and/or `dialogTop`).
`dialogHeight`	Length	varies	Outer height of the dialog box window. IE4 default length unit is `em`; IE5+ is pixel (`px`).
`dialogLeft`	Integer	varies	Pixel offset of dialog box from left edge of screen.
`dialogTop`	Integer	varies	Pixel offset of dialog box from top edge of screen.
`dialogWidth`	Length	varies	Outer width of the dialog box window. IE4 default length unit is `em`; IE5+ is pixel (`px`).
`edge`	String	`raised` | `sunken`	Border style.
`help`	Boolean	`yes`	Display Help icon in title bar.
`resizable`	Boolean	`no`	Dialog box is resizable (IE5+ only).
`status`	Boolean	varies	Display status bar at window bottom (IE5+ only). Default is `yes` for untrusted dialog box; `no` for trusted dialog box.

The CSS-type of syntax for these features lets you string multiple features together by separating each pair with a semicolon within the string. For example:

```
var dlogData = showModalDialog("prefs.html", defaultData,
   "dialogHeight:300px; dialogWidth:460px; help:no");
```

Although not explicitly listed as one of the window features, scroll bars are normally displayed in the window if the content exceeds the size assigned or available to the dialog box. If you don't want scroll bars to appear, have your dialog box document's script set the `document.body.scroll` property to `false` as the page opens.

Dialog cautions

A potential user problem to watch for is that typically a dialog box window does not open until the HTML file for the dialog box has loaded. Therefore, if there is substantial delay before a complex document loads, the user does not see any action indicating that something is happening. You may want to experiment with setting the `cursor` stylesheet property and restoring it when the dialog box's document loads.

One of the reasons I call a dialog box window an (almost) `window` object is that some normal behavior is not available in IE4. For example, if you load a frameset into the dialog box

window, scripts in documents within the frames cannot refer back to the parent document to access variables or parent window methods. Thus, a button in a frame of an IE4 modal dialog box cannot issue `parent.close()` to close the dialog box. This anomaly is repaired in IE5+.

Example

To demonstrate the two styles of dialog boxes, I have implemented the same functionality (setting some session visual preferences) for both modal and modeless dialog boxes. This tactic shows you how to pass data back and forth between the main page and both styles of dialog box windows.

The first example demonstrates how to use a modal dialog box. In the process, data is passed into the dialog box window and values are returned. Listing 16-39 is the HTML and scripting for the main page. A button's `onclick` event handler invokes a function that opens the modal dialog box. The dialog box's document (see Listing 16-40) contains several form elements for entering a user name and selecting a few color styles for the main page. Data from the dialog is fashioned into an array to be sent back to the main window. That array is initially assigned to a local variable, `prefs`, as the dialog box closes. If the user cancels the dialog box, the returned value is an empty string, so nothing more in `getPrefsData()` executes. But when the user clicks OK, the array comes back. Each of the array items is read and assigned to its respective form value or style property. These values are also preserved in the global `currPrefs` array. This allows the settings to be sent to the modal dialog box (as the second parameter to `showModalDialog()`) the next time the dialog box is opened.

Listing 16-39: **Main Page for showModalDialog()**

```
<html>
   <head>
      <title>window.setModalDialog() Method</title>
      <script type="text/javascript">
      var currPrefs = new Array();

      function getPrefsData() {
         var prefs = showModalDialog("lst16-40.htm", currPrefs,
            "dialogWidth:400px; dialogHeight:300px");
         if (prefs) {
            if (prefs["name"]) {
               document.all.firstName.innerText = prefs["name"];
               currPrefs["name"] = prefs["name"];
            }
            if (prefs["bgColor"]) {
               document.body.style.backgroundColor = prefs["bgColor"];
               currPrefs["bgColor"] = prefs["bgColor"];
            }
            if (prefs["textColor"]) {
               document.body.style.color = prefs["textColor"];
               currPrefs["textColor"] = prefs["textColor"];
            }
            if (prefs["h1Size"]) {
               document.all.welcomeHeader.style.fontSize = prefs["h1Size"];
               currPrefs["h1Size"] = prefs["h1Size"];
            }
         }
      }
```

Continued

Listing 16-39 *(continued)*

```
      function init() {
         document.all.firstName.innerText = "friend";
      }
      </script>
   </head>
   <body bgcolor="#EEEEEE" style="margin:20px" onload="init()">
      <h1>window.setModalDialog() Method</h1>
      <hr />
      <h2 id="welcomeHeader">Welcome, <span id="firstName"> </span>!</h2>
      <hr />
      <p>Use this button to set style preferences for this page: <button
         id="prefsButton" onclick="getPrefsData()">Preferences</button></p>
   </body>
</html>
```

The dialog box's document, shown in Listing 16-40, is responsible for reading the incoming data (and setting the form elements accordingly) and assembling form data for return to the main window's script. Notice when you load the example that the `title` element of the dialog box's document appears in the dialog box window's title bar.

When the page loads into the dialog box window, the `init()` function examines the `window.dialogArguments` property. If it has any data, the data is used to preset the form elements to mirror the current settings of the main page. A utility function, `setSelected()`, preselects the option of a `select` element to match the current settings.

Buttons at the bottom of the page are explicitly positioned to be at the lower-right corner of the window. Each button invokes a function to do what is needed to close the dialog box. In the case of the OK button, the `handleOK()` function sets the `window.returnValue` property to the data that come back from the `getFormData()` function. This latter function reads the form element values and packages them in an array using the form elements' names as array indices. This helps keep everything straight back in the main window's script, which uses the index names, and is therefore not dependent upon the precise sequence of the form elements in the dialog box window.

Listing 16-40: **Document for the Modal Dialog**

```
<html>
   <head>
      <title>User Preferences</title>
      <script type="text/javascript">
      // Close the dialog
      function closeme() {
         window.close();
      }

      // Handle click of OK button
      function handleOK() {
         window.returnValue = getFormData();
         closeme();
      }

          // Handle click of Cancel button
```

```
function handleCancel() {
   window.returnValue = "";
   closeme();
}

// Generic function converts form element name-value pairs
// into an array
function getFormData() {
   var form = document.prefs;
   var returnedData = new Array();
   // Harvest values for each type of form element
   for (var i = 0; i < form.elements.length; i++) {
      if (form.elements[i].type == "text") {
         returnedData[form.elements[i].name] = form.elements[i].value;
      } else if (form.elements[i].type.indexOf("select") != -1) {
         returnedData[form.elements[i].name] = form.elements[i].
            options[form.elements[i].selectedIndex].value;
      } else if (form.elements[i].type == "radio") {
         returnedData[form.elements[i].name] = form.elements[i].value;
      } else if (form.elements[i].type == "checkbox") {
         returnedData[form.elements[i].name] = form.elements[i].value;
      } else
         continue;
   }
   return returnedData;
}

// Initialize by setting form elements from passed data
function init() {
   if (window.dialogArguments) {
      var args = window.dialogArguments;
      var form = document.prefs;
      if (args["name"]) {
         form.name.value = args["name"];
      }
      if (args["bgColor"]) {
         setSelected(form.bgColor, args["bgColor"]);
      }
      if (args["textColor"]) {
         setSelected(form.textColor, args["textColor"]);
      }
      if (args["h1Size"]) {
         setSelected(form.h1Size, args["h1Size"]);
      }
   }
}

// Utility function to set a SELECT element to one value
function setSelected(select, value) {
   for (var i = 0; i < select.options.length; i++) {
      if (select.options[i].value == value) {
         select.selectedIndex = i;
         break;
      }
   }
   return;
}

// Utility function to accept a press of the
```

Continued

Listing 16-40 *(continued)*

```
      // Enter key in the text field as a click of OK
      function checkEnter() {
         if (window.event.keyCode == 13) {
            handleOK();
         }
      }
      </script>
   </head>
   <body bgcolor="#EEEEEE" onload="init()">
      <h2>Web Site Preferences</h2>
      <hr />
      <form name="prefs" onsubmit="return false">
         <table>
            <tr>
               <td>Enter your first name:<input name="name" type="text"
                  value="" size="20" onkeydown="checkEnter()" /></td>
            </tr>
            <tr>
               <td>Select a background color: <select name="bgColor">
                     <option value="beige">Beige</option>
                     <option value="antiquewhite">Antique White</option>
                     <option value="goldenrod">Goldenrod</option>
                     <option value="lime">Lime</option>
                     <option value="powderblue">Powder Blue</option>
                     <option value="slategray">Slate Gray</option>
                  </select></td>
            </tr>
            <tr>
               <td>Select a text color: <select name="textColor">
                     <option value="black">Black</option>
                     <option value="white">White</option>
                     <option value="navy">Navy Blue</option>
                     <option value="darkorange">Dark Orange</option>
                     <option value="seagreen">Sea Green</option>
                     <option value="teal">Teal</option>
                  </select></td>
            </tr>
            <tr>
               <td>Select "Welcome" heading font point size: <select
                  name="h1Size">
                     <option value="12">12</option>
                     <option value="14">14</option>
                     <option value="18">18</option>
                     <option value="24">24</option>
                     <option value="32">32</option>
                     <option value="48">48</option>
                  </select></td>
            </tr>
         </table>
      </form>
      <div style="position:absolute; left:200px; top:220px">
         <button style="width:80px"
         onclick="handleOK()">OK</button>   <button
         style="width:80px" onclick="handleCancel()">Cancel</button>
      </div>
   </body>
</html>
```

One last convenience feature of the dialog box window is the `onkeypress` event handler in the text box. The function it invokes looks for the Enter key. If that key is pressed while the box has focus, the same `handleOK()` function is invoked, as if the user had clicked the OK button. This feature makes the dialog box behave as if the OK button is an automatic default, just as "real" dialog boxes.

You should observe several important structural changes that were made to turn the modal approach into a modeless one. Listing 16-41 shows the version of the main window modified for use with a modeless dialog box. Another global variable, `prefsDlog`, is initialized to eventually store the reference to the modeless window returned by the `showModelessWindow()` method. The variable gets used to invoke the `init()` function inside the modeless dialog box, but also as conditions in an `if` construction surrounding the generation of the dialog box. The reason this is needed is to prevent multiple instances of the dialog box being created (the button is still alive while the modeless window is showing). The dialog box won't be created again as long as there is a value in `prefsDlog`, and the dialog box window has not been closed (picking up the `window.closed` property of the dialog box window).

The `showModelessDialog()` method's second parameter is a reference to the function in the main window that updates the main document. As you see in a moment, that function is invoked from the dialog box when the user clicks the OK or Apply buttons.

Listing 16-41: **Main Page for showModelessDialog()**

```
<html>
   <head>
      <title>window.setModelessDialog() Method</title>
      <script type="text/javascript">
      var currPrefs = new Array();
      var prefsDlog;
      function getPrefsData() {
         if (!prefsDlog || prefsDlog.closed) {
            prefsDlog = showModelessDialog("lst16-42.htm", setPrefs,
               "dialogWidth:400px; dialogHeight:300px");
            prefsDlog.init(currPrefs);
         }
      }

      function setPrefs(prefs) {
         if (prefs["bgColor"]) {
            document.body.style.backgroundColor = prefs["bgColor"];
            currPrefs["bgColor"] = prefs["bgColor"];
         }
         if (prefs["textColor"]) {
            document.body.style.color = prefs["textColor"];
            currPrefs["textColor"] = prefs["textColor"];
         }
         if (prefs["h1Size"]) {
            document.all.welcomeHeader.style.fontSize = prefs["h1Size"];
            currPrefs["h1Size"] = prefs["h1Size"];
         }
         if (prefs["name"]) {
            document.all.firstName.innerText = prefs["name"];
            currPrefs["name"] = prefs["name"];
         }
      }
```

Continued

Listing 16-41 *(continued)*

```
      function init() {
         document.all.firstName.innerText = "friend";
      }
      </script>
   </head>
   <body bgcolor="#EEEEEE" style="margin:20px" onload="init()">
      <h1>window.setModelessDialog() Method</h1>
      <hr />
      <h2 id="welcomeHeader">Welcome, <span id="firstName"> </span>!</h2>
      <hr />
      <p>Use this button to set style preferences for this page: <button
         id="prefsButton" onclick="getPrefsData()">Preferences</button></p>
   </body>
</html>
```

Changes to the dialog box window document for a modeless version (see Listing 16-42) are rather limited. A new button is added to the bottom of the screen for an Apply button. As in many dialog box windows you see in Microsoft products, the Apply button lets current settings in dialog boxes be applied to the current document but without closing the dialog box. This approach makes experimenting with settings easier.

The Apply button invokes a `handleApply()` function, which works the same as `handleOK()`, except the dialog box is not closed. But these two functions communicate back to the main window differently than a modal dialog box. The main window's processing function is passed as the second parameter of `showModelessDialog()` and is available as the `window.dialogArguments` property in the dialog box window's script. That function reference is assigned to a local variable in both functions, and the remote function is invoked, passing the results of the `getFormData()` function as parameter values back to the main window.

Listing 16-42: **Document for the Modeless Dialog Box**

```
<html>
   <head>
      <title>User Preferences</title>
      <script type="text/javascript">
      // Close the dialog
      function closeme() {
         window.close();
      }

      // Handle click of OK button
      function handleOK() {
         var returnFunc = window.dialogArguments;
         returnFunc(getFormData());
         closeme();
      }

      // Handle click of Apply button
      function handleApply() {
         var returnFunc = window.dialogArguments;
         returnFunc(getFormData());
      }
```

```
// Handle click of Cancel button
function handleCancel() {
   window.returnValue = "";
   closeme();
}

// Generic function converts form element name-value pairs
// into an array
function getFormData() {
   var form = document.prefs;
   var returnedData = new Array();
   // Harvest values for each type of form element
   for (var i = 0; i < form.elements.length; i++) {
      if (form.elements[i].type == "text") {
         returnedData[form.elements[i].name] = form.elements[i].value;
      } else if (form.elements[i].type.indexOf("select") != -1) {
         returnedData[form.elements[i].name] = form.elements[i].
            options[form.elements[i].selectedIndex].value;
      } else if (form.elements[i].type == "radio") {
         returnedData[form.elements[i].name] = form.elements[i].value;
      } else if (form.elements[i].type == "checkbox") {
         returnedData[form.elements[i].name] = form.elements[i].value;
      } else
         continue;
   }
   return returnedData;
}

// Initialize by setting form elements from passed data
function init(currPrefs) {
   if (currPrefs) {
      var form = document.prefs;
      if (currPrefs["name"]) {
         form.name.value = currPrefs["name"];
      }
      if (currPrefs["bgColor"]) {
         setSelected(form.bgColor, currPrefs["bgColor"]);
      }
      if (currPrefs["textColor"]) {
         setSelected(form.textColor, currPrefs["textColor"]);
      }
      if (currPrefs["h1Size"]) {
         setSelected(form.h1Size, currPrefs["h1Size"]);
      }
   }
}

// Utility function to set a SELECT element to one value
function setSelected(select, value) {
   for (var i = 0; i < select.options.length; i++) {
      if (select.options[i].value == value) {
         select.selectedIndex = i;
         break;
      }
   }
   return;
}
```

Continued

Listing 16-42 *(continued)*

```
    // Utility function to accept a press of the
    // Enter key in the text field as a click of OK
    function checkEnter() {
       if (window.event.keyCode == 13) {
          handleOK();
       }
    }
    </script>
 </head>
 <body bgcolor="#EEEEEE" onload="init()">
    <h2>Web Site Preferences</h2>
    <hr />
    <form name="prefs" onsubmit="return false">
       <table>
          <tr>
             <td>Enter your first name:<input name="name" type="text"
                value="" size="20" onkeydown="checkEnter()" /></td>
          </tr>
          <tr>
             <td>Select a background color: <select name="bgColor">
                   <option value="beige">Beige</option>
                   <option value="antiquewhite">Antique White</option>
                   <option value="goldenrod">Goldenrod</option>
                   <option value="lime">Lime</option>
                   <option value="powderblue">Powder Blue</option>
                   <option value="slategray">Slate Gray</option>
                </select></td>
          </tr>
          <tr>
             <td>Select a text color: <select name="textColor">
                   <option value="black">Black</option>
                   <option value="white">White</option>
                   <option value="navy">Navy Blue</option>
                   <option value="darkorange">Dark Orange</option>
                   <option value="seagreen">Sea Green</option>
                   <option value="teal">Teal</option>
                </select></td>
          </tr>
          <tr>
             <td>Select "Welcome" heading font point size: <select
                name="h1Size">
                   <option value="12">12</option>
                   <option value="14">14</option>
                   <option value="18">18</option>
                   <option value="24">24</option>
                   <option value="32">32</option>
                   <option value="48">48</option>
                </select></td>
          </tr>
       </table>
    </form>
    <div style="position:absolute; left:120px; top:220px">
       <button style="width:80px"
       onclick="handleOK()">OK</button>   <button
       style="width:80px"
       onclick="handleCancel()">Cancel</button>   <button
```

```
            style="width:80px" onclick="handleApply()">Apply</button>
        </div>
    </body>
</html>
```

The biggest design challenge you probably face with respect to these windows is deciding between a modal and modeless dialog box style. Some designers insist that modality has no place in a graphical user interface; others say that there are times when you need to focus the user on a very specific task before any further processing can take place. That's where a modal dialog box makes perfect sense.

Related Items: `window.open()` method.

sizeToContent()

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The NN6+ `window.sizeToContent()` method can be a valuable aid in making sure that a window (especially a subwindow) is sized for the optimum display of the window's content. But you must also be cautious with this method, or it will do more harm than good.

Invoking the `sizeToContent()` method resizes the window so that all content is visible. Concerns about variations in OS-specific rendering become a thing of the past. Naturally, you should perform this action only on a window whose content at the most occupies a space smaller than the smallest video monitor running your code (typically 640 × 480 pixels, but conceivably much smaller for future versions of the browser used on handheld computers).

You can get the user in trouble, however, if you invoke the method twice on the same window that contains the resizing script. This action can cause the window to expand to a size that may exceed the pixel size of the user's video monitor. Successive invocations fail to cinch up the window's size to its content again. Multiple invocations are safe, however, on subwindows when the resizing script statement is in the main window.

Example

Use The Evaluator (Chapter 13) in NN6 to try the `sizeToContent()` method. Assuming that you are running The Evaluator from the `Chap13` directory on the CD-ROM (or the directory copied as-is to your hard disk), you can open a subwindow with one of the other files in the directory, and then size the subwindow. Enter the following statements into the top text box:

```
a = window.open("lst13-02.htm","")
a.sizeToContent()
```

The resized subwindow is at the minimum recommended width for a browser window, and at a height tall enough to display the little bit of content in the document.

Related Item: `window.resizeTo()` method.

stop()

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

The Navigator-specific `stop()` method offers a scripted equivalent of clicking the Stop button in the toolbar. Availability of this method allows you to create your own toolbar on your page and hide the toolbar (in the main window with signed scripts or in a subwindow).

For example, if you have an image representing the Stop button in your page, you can surround it with a link whose action stops loading, as in the following:

```
<a href="javascript: void stop()"><img src="myStop.gif" border="0"></a>
```

A script cannot stop its own document from loading, but it can stop loading of another frame or window. Similarly, if the current document dynamically loads a new image or a multimedia MIME type file as a separate action, the `stop()` method can halt that process. Even though the `stop()` method is a window method, it is not tied to any specific window or frame: Stop means stop.

Related Items: `window.back()`, `window.find()`, `window.forward()`, `window.home()`, `window.print()` methods.

Event handlers

onafterprint
onbeforeprint

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

Each of these event handlers fires after the user has clicked the OK button in IE's Print dialog box. This goes for printing that is invoked manually (via menus and browser shortcut buttons) and the `window.print()` method.

Although printing is usually WYSIWYG, it is conceivable that you may want the printed version of a document to display more or less of the document than is showing at that instant. For example, you may have a special copyright notice that you want printed at the end of a page whenever it goes to the printer. In that case, the element with that content can have its `display` stylesheet property set to `none` when the page loads. Before the document is sent to the printer, a script needs to adjust that style property to display the element as a block item; after printing, have your script revert the setting to `none`.

Immediately after the user clicks the OK button in the Print dialog box, the `onbeforeprint` event handler fires. As soon as the page(s) is sent to the printer or spooler, the `onafterprint` event handler fires.

Example

The following script fragment assumes that the page includes a `div` element whose stylesheet includes a setting of `display:none` as the page loads. Somewhere in the Head, the print-related event handlers are set as properties:

```
function showPrintCopyright() {
   document.all.printCopyright.style.display = "block";
}
function hidePrintCopyright() {
   document.all.printCopyright.style.display = "none";
}
window.onbeforeprint = showPrintCopyright;
window.onafterprint = hidePrintCopyright;
```

onbeforeunload

Compatibility: WinIE4+, MacIE5+, NN-, Moz-, Safari-

Any user or scripted action that normally forces the current page to be unloaded or replaced causes the `onbeforeunload` event handler to fire. Unlike the `onunload` event handler,

however, onbeforeunload is a bit better behaved when it comes to allowing complex scripts to finish before the actual unloading takes place. Moreover, you can assign a string value to the event.returnValue property in the event handler function. That string becomes part of a message in an alert window that gives the user a chance to stay on the page. If the user agrees to stay, the page does not unload, and any action that caused the potential replacement is cancelled.

Example

The simple page in Listing 16-43 shows you how to give the user a chance to stay on the page.

Listing 16-43: Using the onbeforeunload Event Handler

```
<html>
   <head>
      <title>onbeforeunload Event Handler</title>
      <script type="text/javascript">
      function verifyClose() {
         event.returnValue =
            "We really like you and hope you will stay longer.";
      }

      window.onbeforeunload = verifyClose;
      </script>
   </head>
   <body>
      <h1>onbeforeunload Event Handler</h1>
      <hr />
      <p>Use this button to navigate to the previous page: <button id="go"
         onclick="history.back()">Go Back</button></p>
   </body>
</html>
```

Related Items: onunload event handler.

ondragdrop

Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

With closer integration between the computer desktop and browsers these days, it is increasingly possible that shortcuts (or aliases) to Web URLs can be represented on our desktops and other kinds of documents. With NN4 (only), you can script awareness of dragging and dropping of such items onto the browser window. The window's dragdrop event fires whenever a user drops a file or other URL-filled object onto the window.

Example

You can add an ondragdrop event handler to the <body> tag of your document and pass along the event object that has some juicy tidbits about the drop: the object on which the item was dropped and the URL of the item. The function called by the event handler receives the event object information and can process it from there. Because this event is a window event, you don't have to turn on window.captureEvents() to get the window to feel the effect of the event.

The juiciest tidbit of the event, the URL of the dropped item, can be retrieved only with a signed script and the user's permission (see Chapter 46 on the CD-ROM). Listing 16-44 shows a simple document that reveals the URL and screen location, as derived from the event object passed with the `dragdrop` event. You must have codebase principals turned on to get the full advantage of this listing, and it works best with Windows.

Listing 16-44: **Analyzing a dragdrop Event**

```
<html>
   <head>
      <title>DragDrop Event</title>
      <script type="text/javascript">
      function reportDrag(e) {
         var msg = "You dropped the file:\n";
         netscape.security.PrivilegeManager.enablePrivilege(
            "UniversalBrowserRead");
         msg += e.data;
         netscape.security.PrivilegeManager.disablePrivilege(
            "UniversalBrowserRead");
         msg += "\nonto the window object at screen location (";
         msg += e.screenX + "," + e.screenY + ").";
         alert(msg);
         return false;
      }
      </script>
   </head>
   <body ondragdrop="return reportDrag(event)">
      <b>Drag and Drop a file onto this window</b>
   </body>
</html>
```

The `dragdrop` event is the only one that uses the `data` property of the NN4 event object. That property contains the URL. The `target` property reveals only the `window` object, but you can access the event object's `screenX` and `screenY` properties to get the location of the mouse release.

Related Items: `event` object (Chapter 25).

onerror

Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

(See the discussion of the `window.onerror` property earlier in this chapter.)

onhelp

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The generic `onhelp` event handler is discussed in Chapter 15, but it also fires when the user activates the context-sensitive help within a modal or modeless dialog box. In the latter case, a user can click the Help icon in the dialog box's title bar, at which time the cursor changes to a question mark. The user can then click on any element in the window. At that second click, the `onhelp` event handler fires, and the `event` object contains information about the element clicked (the `event.srcElement` is a reference to the specific element), allowing a script to supply help about that element.

To prevent the brower's built-in help window from appearing, the event handler must evaluate to `return false` (IE4+) or set the `event.returnValue` property to `false` (IE5+).

Example

The following script fragment can be embedded in the IE5+-only modeless dialog box code in Listing 16-44 to provide context-sensitive help within the dialog box. Help messages for only two of the form elements are shown here, but in a real application you add messages for the rest.

```
function showHelp() {
   switch (event.srcElement.name) {
      case "bgColor" :
         alert("Choose a color for the main window\'s background.");
         break;
      case "name" :
         alert("Enter your first name for a friendly greeting.");
         break;
      default :
         alert("Make preference settings for the main page styles.");
   }
   event.returnValue = false;
}
window.onhelp = showHelp
```

Because this page's help focuses on form elements, the `switch` construction cases are based on the `name` properties of the form elements. For other kinds of pages, the `id` properties may be more appropriate.

Related Items: `event` object (Chapter 25); `switch` construction (Chapter 31).

onload

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `onload` event handler fires in the current window at the end of the document loading process (after all text and image elements have been transferred from the source file server to the browser, and after all plug-ins and Java applets have loaded and started running). At that point, the browser's memory contains all the objects and script components in the document that the browser can possibly know about.

The `onload` handler is an attribute of a `<body>` tag for a single-frame document or of the `<frameset>` tag for the top window of a multiple-frame document. When the handler is an attribute of a `<frameset>` tag, the event triggers only after all frames defined by that frameset have completely loaded.

Use either of the following scenarios to insert an `onload` handler into a document:

```
<html>
   <head>
   </head>
   <body [other attributes] onload="statementOrFunction">
      [body content]
   </body>
</html>

<html>
   <head>
   </head>
   <frameset [other attributes] onload="statementOrFunction">
```

```
      <frame>frame specifications</frame>
   </frameset>
</html>
```

This handler has a special capability when part of a frameset definition: The handler won't fire until the `onload` event handlers of all child frames in the frameset have fired. Therefore, if some initialization scripts depend on components existing in other frames, trigger them from the frameset's `onload` event handler. This brings up a good general rule of thumb for writing JavaScript: Scripts that execute during a document's loading should contribute to the process of generating the document and its objects. To act immediately on those objects, design additional functions that are called by the `onload` event handler for that window.

The type of operations suited for an `onload` event handler are those that can run quickly and without user intervention. Users shouldn't be penalized by having to wait for considerable post-loading activity to finish before they can interact with your pages. At no time should you present a modal dialog box as part of an `onload` handler. Users who design macros on their machines to visit sites unattended may get hung up on a page that automatically displays an alert, confirm, or prompt dialog box. On the other hand, an operation such as setting the `window.defaultStatus` property is a perfect candidate for an `onload` event handler, as are initializing event handlers as properties of element objects in the page.

Note Browsers equipped with pop-up window blockers ignore all `window.open()` method calls in `onload` event handler functions.

Related Items: `onunload` event handler; `window.defaultStatus` property.

onmove

Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

If a user drags an NN4 window around the screen, the action triggers a `move` event for the `window` object. When you assign a function to the event (for example, `window.onmove = handleMoves`), the function receives an event object whose `screenX` and `screenY` properties reveal the coordinate point (relative to the entire screen) of the top-left corner of the window after the move.

Related Items: `event` object (Chapter 25).

onresize

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari-

If a user resizes a window, the action causes the `onresize` event handler to fire for the `window` object. When you assign a function to the event (for example, `window.onresize = handleResizes`), the NN event object conveys `width` and `height` properties that reveal the outer width and height of the entire window. A window resize should not reload the document such that an `onload` event handler fires (although some early Navigator versions did fire the extra event).

Note Resizing the Navigator 4 browser window, especially if that window contains positioned elements (as `div` or `layer` elements), causes serious problems not only for the content, but also for scripts in the page. Content can get jumbled, and scripts may disappear. Your only hope is to use an `onresize` event handler to reload the page and get back to a known point. For some ideas on handling this problem, see the article at `http://developer.netscape.com/viewsource/goodman_resize/goodman_resize.html`. One point not covered in the article is that the Windows version of NN4 issues a `resize` event when scrollbars appear

in a window. This resize event can make any reload-on-resize strategy turn into an infinite loop. To guard against this, you have to inspect the window.innerWidth and window.innerHeight properties to see if the window has really changed (the property values don't change when the scrollbars appear). Here is an example of script statements that go in the Head script of a page that has to worry about this problem in NN4:

```
var Nav4 = (navigator.appName == "Netscape" &&
   parseInt(navigator.appVersion) == 4);
if (Nav4) {
   var loadWidth = window.innerWidth;
   var loadHeight = window.innerHeight;
}

function restore() {
   if (loadWidth != window.innerWidth || loadHeight !=
window.innerHeight) {
      history.go(0);
   }
}

if (Nav4)
   window.onresize = restore;
```

This problem does not occur in NN6+ or Mozilla-based browsers.

Related Items: event object (Chapter 25).

onunload

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

An unload event reaches the current window just before a document is cleared from view. The most common ways windows are cleared are when new HTML documents are loaded into them or when a script begins writing new HTML on the fly for the window or frame.

Limit the extent of the onunload event handler to quick operations that do not inhibit the transition from one document to another. Do not invoke any methods that display dialog boxes. You specify onunload event handlers in the same places in an HTML document as the onload handlers: as a <body> tag attribute for a single-frame window or as a <frameset> tag attribute for a multiframe window. Both onload and onunload event handlers can appear in the same <body> or <frameset> tag without causing problems. The onunload event handler merely stays safely tucked away in the browser's memory, waiting for the unload event to arrive for processing as the document gets ready to clear the window.

Let me pass along one caution about the onunload event handler. Even though the event fires before the document goes away, don't burden the event handler with time-consuming tasks, such as generating new objects or submitting a form. The document will probably go away before the function completes, leaving the function looking for objects and values that no longer exist. The best defense is to keep your onunload event handler processing to a minimum.

Browsers equipped with pop-up window blockers ignore all window.open() method calls in onunload event handler functions.

Related Items: onload event handler.

frame Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
allowTransparency		
borderColor		
contentDocument		
contentWindow		
frameBorder		
height		
longDesc		
marginHeight		
marginWidth		
name		
noResize		
scrolling		
src		
width		

Syntax

Accessing properties or methods of a `frame` element object from a `frameset`:

```
(IE4+)     document.all.frameID. property | method([parameters])
(IE5+/W3C) document.getElementById("frameID"). property | method([parameters])
```

Accessing properties of methods of a `frame` element from a frame document:

```
(IE4+)     parent.document.all.frameID. property | method([parameters])
(IE5+/W3C) parent.document.getElementById("frameID"). property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About this object

As noted in the opening section of this chapter, a `frame` element object is distinct from the frame object that acts as a `window` object in a document hierarchy. The `frame` element object is available to scripts only when all HTML elements are exposed in the object model, as in IE4+, NN6+, Mozilla, and Safari.

Because the `frame` element object is an HTML element, it shares the properties, methods, and event handlers of all HTML elements, as described in Chapter 15. By and large, you access the `frame` element object to set or modify an attribute value in the `<frame>` tag. If so, you simplify matters if you assign an identifier to the `id` attribute of the tag. Your tag still needs a `name` attribute if your scripts refer to frames through the original object model (a `parent.frameName` reference). While there is no law against using the same identifier for

both `name` and `id` attributes, using different names to prevent potential conflict with references in browsers that recognize both attributes is best.

To modify the dimensions of a frame, you must go the `frameset` element object that defines the `cols` and `rows` attributes for the frameset. These properties can be modified on the fly in IE4+ and W3C DOM–compatible browsers.

Properties

allowTransparency

Value: Boolean. Read/Write
Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-

The `allowTransparency` property indicates whether or not the frame's background is transparent. This property applies primarily to the `iframe` object, since framesets don't have background colors or images to show through a transparent frame.

borderColor

Value: Hexadecimal triplet or color name string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

If a frame displays a border (as determined by the `frameborder` attribute of the `frame` element or `border` attribute of the `frameset` element), it can have a color set separately from the rest of the frames. The initial color (if different from the rest of the frameset) is usually set by the `bordercolor` attribute of the `<frame>` tag. After that, scripts can modify settings as needed.

Modifying a single frame's border can be risky at times, depending on your color combinations. In practice, different browers appear to follow different rules when it comes to negotiating conflicts or defining just how far a single frame's border extends into the border space. Color changes to individual frame borders do not always render. Verify your designs on as many browsers and operating system variations as you can to test your combinations.

Example

Although you may experience problems changing the color of a single frame border, the W3C DOM syntax would look like the following if the script were inside the framesetting document:

```
document.getElementById("contentsFrame").borderColor = "red";
```

The IE-only version would be:

```
document.all["contentsFrame"].borderColor = "red";
```

These examples assume the frame name arrives to a script function as a string. If the script is executing in one of the frames of the frameset, add a reference to `parent` in the preceding statements.

Related Items: `frame.frameBorder`, `frameset.frameBorder` properties.

contentDocument

Value: `document` object reference. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `contentDocument` property of a `frame` element object is nothing more than a reference to the document contained by that frame. This property bridges the gap between the `frame`

element object and the frame object. Both of these objects contain the same `document` object, but from a scripting point of view, references most typically use the frame object to reach the document inside a frame, while the `frame` element is used to access properties equated with the `frame` tag's attributes. But if your script finds that it has a reference to the `frame` element object, you can use the `contentDocument` property to get a valid reference to the document, and therefore any other content of the frame.

Example

A framesetting document script might be using the ID of a `frame` element to read or adjust one of the element properties, and then need to perform some action on the content of the page through its `document` object. You can get the reference to the `document` object via a statement, such as the following:

```
var doc = document.getElementById("Frame3").contentDocument;
```

Then your script can, for example, dive into a form in the document:

```
var val = doc.mainForm.entry.value;
```

Related Items: `contentWindow` property; `document` object.

contentWindow

Value: `document` object reference. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN7+, Moz1+, Safari-

The `contentWindow` property of a `frame` element object is simply a reference to the window generated by that frame. This property provides access to the frame's window, which can then be used to reach the document inside the frame.

Example

You can get the reference to the `window` object associated with a frame via a statement, such as the following:

```
var win = document.getElementById("Frame3").contentWindow;
```

Related Items: `window` object.

frameBorder

Value: `yes` | `no` | `1` | `0` as strings. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `frameBorder` property offers scripted access to a `frame` element object's `frameborder` attribute setting. IE4+ does not respond well to modifying this property after the page has loaded.

Values for the `frameBorder` property are strings that substitute for Boolean values. Values `yes` or `1` mean that the border is (supposed to be) turned on; `no` or `0` turn off the border.

Example

The default value for the `frameBorder` property is `yes`. You can use this setting to create a toggle script (which, unfortunately, does not change the appearance in IE). The W3C-compatible version looks like the following:

```
function toggleFrameScroll(frameID) {
   var theFrame = document.getElementById(frameID);
   if (theFrame.frameBorder == "yes") {
      theFrame.frameBorder = "no";
   } else {
      theFrame.frameBorder = "yes";
   }
}
```

Related Items: `frameset.frameBorder` properties.

height
width

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

IE4+ lets you retrieve the height and width of a `frame` element object. These values are not necessarily the same as the `document.body.clientHeight` and `document.body.clientWidth`, because the frame dimensions include chrome associated with the frame, such as scrollbars. These values are read-only. If you need to modify the dimensions of a frame, do so via the `frameset` element object's `rows` and/or `cols` properties. Reading integer values for a frame's height and width properties is much easier than trying to parse the `rows` and `cols` string properties.

Example

The following fragment assumes a frameset defined with two frames set up as two columns within the frameset. The statements here live in the framesetting document. They retrieve the current width of the left frame and increase the width of that frame by 10 percent. Syntax shown here is for the W3C DOM, but can be easily adapted to IE-only terminology.

```
var frameWidth = document.getElementById("leftFrame").width;
document.getElementById("mainFrameset").cols =
   (Math.round(frameWidth * 1.1)) + ",*";
```

Notice how the numeric value of the existing frame width is first increased by 10 percent and then concatenated to the rest of the string property assigned to the frameset's `cols` property. The asterisk after the comma means that the browser should figure out the remaining width and assign it to the right-hand frame.

Related Items: `frameset` object.

longDesc

Value: URL string. Read/Write
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The `longDesc` property is the scripted equivalent of the `longdesc` attribute of the `<frame>` tag. This HTML 4.0 attribute is intended to provide browsers with a URL to a document that contains a long description of the element. Future browsers can use this feature to provide information about the frame for visually impaired site visitors.

marginHeight
marginWidth

Value: Integer. Read/Write
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

Browsers tend to automatically insert content within a frame by adding a margin between the content and the edge of the frame. These values are represented by the `marginHeight` (top and bottom edges) and `marginWidth` (left and right edges) properties. Although the properties are not read-only, changing the values after the frameset has loaded does not alter the appearance of the document in the frame. If you need to alter the margin(s) of a document inside a frame, adjust the `document.body.style` margin properties.

Also be aware that although the default values of these properties are empty (meaning when no `marginheight` or `marginwidth` attributes are set for the `<frame>` tag), margins are built into the page. The precise pixel count of those margins varies with operating system.

Related Items: `style` object (Chapter 26).

name

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `name` property is the identifier associated with the frame for use as a frame reference. Scripts can reference the frame through the `name` property (e.g., `top.frames["myFrame"]`), which is typically assigned via the `name` attribute.

noResize

Value: Boolean. Read/Write
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

Web designers commonly fix their framesets so that users cannot resize the frames (by dragging any divider border between frames). The `noResize` property lets you read and adjust that behavior of a frame after the page has loaded. For example, during some part of the interaction with a user on a page, you may allow the user to modify the frame size manually while in a certain mode. Or you may grant the user one chance to resize the frame. When the `onresize` event handler fires, a script sets the `noResize` property of the `frame` element to `false`. If you turn off resizing for a frame, all edges of the frame become non-resizable, regardless of the `noResize` value setting of adjacent frames. Turning off resizability has no effect on the ability of scripts to alter the sizes of frames via the `frameset` element object's `cols` or `rows` properties.

Example

The following statement turns off the ability for a frame to be resized:

```
parent.document.getElementById("myFrame1").noResize = true;
```

Because of the negative nature of the property name, it may be difficult to keep the logic straight (setting `noResize` to `true` means that resizability is turned off). Keep a watchful eye on your Boolean values.

Related Items: `frameset.cols`, `frameset.rows` properties.

scrolling

Value: yes | no | 1 | 0 as strings. Read/Write
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The scrolling property lets scripts turn scrollbars on and off inside a single frame of a frameset. By default, scrolling is turned on unless overridden by the scroll attribute of the <frame> tag.

Values for the scrolling property are strings that substitute for Boolean values. Values yes or 1 mean that scrollbars are visible (provided there is more content than can be viewed without scrolling); no or 0 hide scrollbars in the frame. IE4+ also recognizes (and sets as default) the auto value.

Although this property is read/write, changing its value by script does not alter a frame's appearance in WinIE5.5+, Mozilla browsers, or Safari.

Example

Listing 16-45 produces a frameset consisting of eight frames. The content for the frames is generated by a script within the frameset (via the fillFrame() function). Event handlers (onclick) in the Body of each frame invoke the toggleFrameScroll() function. Both ways of referencing the frame element object are shown, with the IE-only version commented out.

In the toggleFrameScroll() function, the if condition checks whether the property is set to something other than no. This allows the condition to evaluate to true if the property is set to either auto (the first time) or yes (as set by the function). Note that the scrollbars don't disappear from the frames in IE5.5+, NN6+, Moz, or Safari.

Listing 16-45: Controlling the frame.scrolling Property

```
<html>
   <head>
      <title>frame.scrolling Property</title>
      <script type="text/javascript">
      function toggleFrameScroll(frameID) {
         // IE5+/W3C version
         var theFrame = document.getElementById(frameID);
         // IE4+ version
         // var theFrame = document.all[frameID];

         if (theFrame.scrolling != "no") {
            theFrame.scrolling = "no";
         } else {
            theFrame.scrolling = "yes";
         }
      }

      // generate content for each frame
      function fillFrame(frameID) {
         var page = "<html><body onclick='parent.toggleFrameScroll(\"" +
         frameID + "\")'><span style='font-size:24pt'>";
         page += "<p>This frame has the ID of:<\/p><p>" + frameID + ".<\/p>";
         page += "<\/span><\/body><\/html>";
         return page;
```

Continued

Listing 16-45 *(continued)*

```
      }
      </script>
   </head>
   <frameset id="outerFrameset" cols="50%,50%">
      <frameset id="innerFrameset1" rows="25%,25%,25%,25%">
         <frame id="myFrame1" src="javascript:parent.fillFrame('myFrame1')" />
         <frame id="myFrame2" src="javascript:parent.fillFrame('myFrame2')" />
         <frame id="myFrame3" src="javascript:parent.fillFrame('myFrame3')" />
         <frame id="myFrame4" src="javascript:parent.fillFrame('myFrame4')" />
      </frameset>
      <frameset id="innerFrameset2" rows="25%,25%,25%,25%">
         <frame id="myFrame5" src="javascript:parent.fillFrame('myFrame5')" />
         <frame id="myFrame6" src="javascript:parent.fillFrame('myFrame6')" />
         <frame id="myFrame7" src="javascript:parent.fillFrame('myFrame7')" />
         <frame id="myFrame8" src="javascript:parent.fillFrame('myFrame8')" />
      </frameset>
   </frameset>
</html>
```

src

Value: URL string. Read/Write
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The `src` property of a `frame` element object offers an additional way of navigating to a different page within a frame (meaning other than assigning a new URL to the `location.href` property of the frame object). For backward compatibility with older browsers, however, continue using `location.href` for scripted navigation. Remember that the `src` property belongs to the `frame` element object, not the `window` object it represents. Therefore, references to the `src` property must be via the element's ID and/or node hierarchy.

Example

For best results, use fully formed URLs as value for the `src` property, as shown here:

```
parent.document.getElementById("mainFrame").src = "http://www.dannyg.com";
```

Relative URLs and `javascript:` pseudo-URLs will also work most of the time.

Related Items: `location.href` property.

frameset Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	Methods	Event Handlers
border		
borderColor		
cols		
frameBorder		
frameSpacing		
rows		

Syntax

Accessing properties or methods of a `frameset` element object from a `frameset`:

```
(IE4+)     document.all.framesetID. property | method([parameters])
(IE5+/W3C) document.getElementById("framesetID"). property |
method([parameters])
```

Accessing properties of methods of a `frameset` element from a frame document:

```
(IE4+)     parent.document.all.framesetID. property | method([parameters])
(IE5+/W3C) parent.document.getElementById("framesetID"). property |
           method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About this object

The `frameset` element object is the script-accessible equivalent of the element generated via the `<frameset>` tag. This element is different from the parent (window-type) object from the original object model. A `frameset` element object has properties and methods that impact the HTML element; in contrast, the `window` object referenced from documents inside frames via the `parent` or `top` window references contains a document and all the content that goes along with it.

When framesets are nested in one another, a node parent–child relationship exists between containing and contained framesets. For example, consider the following skeletal nested frameset structure:

```
<frameset id="outerFrameset" cols="30%, 70%">
   <frame id="frame1">
   <frameset id="innerFrameset" rows="50%,50%">
      <frame id="frame2">
      <frame id="frame3">
   </frameset>
</frameset>
```

When writing scripts for documents that go inside any of the frames of this structure, references to the framesetting window and frames are a flatter hierarchy than the HTML signifies. A script in any frame references the framesetting window via the `parent` reference; a script in any frame references another frame via the `parent.frameName` reference. In other words, the `window` objects of the frameset defined in a document are all siblings and share the same parent.

Such is not the case when viewing the preceding structure from the perspective of W3C node terminology. Parent–child relationships are governed by the nesting of HTML elements, irrespective of whatever windows get generated by the browser. Therefore, frame `frame2` has only one sibling, `frame3`. Both of those share one parent, `innerFrameset`. Both `innerFrameset` and `frame1` are children of `outerFrameset`. If your script were sitting on a reference to `frame2`, and you wanted to change the `cols` property of `outerFrameset`, you would have to traverse two generations of nodes:

```
frame2Ref.parentNode.parentNode.cols = "40%,60%";
```

What might confuse matters ever more in practice is that a script belonging to one of the frames must use window object terminology to jump out of the current `window` object to the frameset that generated the frame window for the document. In other words, there is no immediate way to jump directly from a document to the `frame` element object that defines the frame in which the document resides. The document's script accesses the node hierarchy of its frameset via the `parent.document` reference. But this reference is to the `document` object that contains the entire frameset structure. Fortunately, the W3C DOM provides the `getElementById()` method to extract a reference to any node nested within the document. Thus, a document inside one of the frames can access the `frame` element object just as if it were any element in a typical document (which it is):

```
parent.document.getElementById("frame2")
```

No reference to the containing `frameset` element object is necessary. Or, to make that column width change from a script inside one of the frame windows, the statement would be:

```
parent.document.getElementById("outerFrame").cols = "40%,60%";
```

The inner frameset is equally accessible by the same syntax.

Properties

border

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari1+

The `border` property of a `frameset` element object lets you read the thickness (in pixels) of the borders between frames of a frameset. If you do not specify a `border` attribute in the frameset's tag, the property is empty, rather than reflecting the actual border thickness applied by default.

Example

Even though the property is read/write, changing the value does not change the thickness of the border you see in the browser. If you need to find the thickness of the border, a script reference from one of the frame's documents would look like the following:

```
var thickness = parent.document.all.outerFrameset.border;
```

Related Items: `frameset.frameBorder` property.

borderColor

Value: Hexadecimal triplet or color name string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `borderColor` property lets you read the value of the color assigned to the `bordercolor` attribute of the frameset's tag. Although the property is read/write, changing the color by script does not alter the border colors rendered in the browser window. Attribute values set as color names are returned as hexadecimal triplets when you read the property value.

Example

To retrieve the current color setting in a frameset, a script reference from one of the frame's documents would look like the following:

```
var borderColor = parent.document.all.outerFrameset.borderColor;
```

Related Items: `frame.borderColor`, `frameset.frameBorder` properties.

cols
rows

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `cols` and `rows` properties of a `frameset` element object let you read and modify the sizes of frames after the frameset has loaded. These two properties are defined in the W3C DOM. Values for both properties are strings, which may include percent symbols or asterisks. Therefore if you are trying to increase or decrease the size of a frame column or row gradually, you must parse the string for the necessary original values before performing any math on them (or, in IE4+, use the `frame` element object's `height` and `width` properties to gauge the current frame size in pixels).

Adjusting these two properties lets you completely modify the frameset. This includes adding or removing columns or rows to the frameset grid. Because a change in the frameset structure could impact scripts by changing the size of the frames array associated with the parent window or unloading documents that contain needed data, be sure to test your scripts with both states of your frameset. If you want to remove a frame from a frameset view, you might be safer to specify the size of zero for that particular row or column in the frameset. Of course a size of zero still leaves a one-pixel frame, but it is essentially invisible if borders are not turned on and the 1-pixel frame shares the same background color as the other frames. Another positive by-product of this technique is that you can restore the other frame with its document state identical from when it was hidden.

When you have nested framesets defined in a single document, be sure to reference the desired `frameset` element object. One object may be specifying the columns, while another (nested) one specifies the rows for the grid. Assign a unique ID to each `frameset` element so that references can be reliably directed to the proper object.

Example

Listings 16-46 through 16-48 show the HTML for a frameset and two of the three documents that go into the frameset. The final document is an HTML version of the U.S. Bill of Rights, which is serving here as a content frame for the demonstration.

frameset.cols

The frameset listing (see Listing 16-46) shows a three-frame setup. Down the left column is a table of contents (see Listing 16-47). The right column is divided into two rows. In the top row is a simple control (see Listing 16-48) that hides and shows the table of contents frame. As the user clicks the hot text of the control (located inside a `span` element), the `onclick` event handler invokes the `toggleTOC()` function in the frameset.

Syntax used in this example is W3C-compatible. To modify this for IE-only, you replace `document.getElementById("outerFrameset")` with `document.all.outerFrameset` and `elem.firstChild.nodeValue` to `elem.innerText`. You can also branch within the scripts to accommodate both styles.

Listing 16-46: **Frameset and Script for Hiding/Showing a Frame**

```
<html>
   <head>
      <title>Hide/Show Frame Example</title>
      <script type="text/javascript">
      var origCols;
      function toggleTOC(elem, frm) {
         if (origCols) {
            showTOC(elem);
         } else {
            hideTOC(elem, frm);
         }
      }
      function hideTOC(elem, frm) {
         var frameset = document.getElementById("outerFrameset");
         origCols = frameset.cols;
         frameset.cols = "0,*";
      }
      function showTOC(elem) {
         if (origCols) {
            document.getElementById("outerFrameset").cols = origCols;
            origCols = null;
         }
      }
      </script>
   </head>
   <frameset id="outerFrameset" frameborder="no" cols="150,*">
      <frame id="TOC" name="TOCFrame" src="lst16-47.htm" />
      <frameset id="innerFrameset1" rows="80,*">
         <frame id="controls" name="controlsFrame" src="lst16-48.htm" />
         <frame id="content" name="contentFrame" src="bofright.htm" />
      </frameset>
   </frameset>
</html>
```

When a user clicks the hot spot to hide the frame, the script copies the original `cols` property settings to a global variable. The variable is used in `showTOC()` to restore the frameset to its original proportions. This allows a designer to modify the HTML for the frameset without also having to dig into scripts to hard-wire the restored size.

Listing 16-47: **Table of Contents Frame Content**

```
<html>
   <head>
      <title>Table of Contents</title>
   </head>
   <body bgcolor="#EEEEEE">
      <h3>
         Table of Contents
      </h3>
      <hr />
      <ul style="font-size:10pt">
         <li><a href="bofright.htm#article1" target="contentFrame">Article
            I</a></li>
         <li><a href="bofright.htm#article2" target="contentFrame">Article
            II</a></li>
         <li><a href="bofright.htm#article3" target="contentFrame">Article
            III</a></li>
         <li><a href="bofright.htm#article4" target="contentFrame">Article
            IV</a></li>
         <li><a href="bofright.htm#article5" target="contentFrame">Article
            V</a></li>
         <li><a href="bofright.htm#article6" target="contentFrame">Article
            VI</a></li>
         <li><a href="bofright.htm#article7" target="contentFrame">Article
            VII</a></li>
         <li><a href="bofright.htm#article8" target="contentFrame">Article
            VIII</a></li>
         <li><a href="bofright.htm#article9" target="contentFrame">Article
            IX</a></li>
         <li><a href="bofright.htm#article10" target="contentFrame">Article
            X</a></li>
      </ul>
   </body>
</html>
```

Listing 16-48: **Control Panel Frame**

```
<html>
   <head>
      <title>Control Panel</title>
   </head>
   <body>
      <p><span id="tocToggle" style="text-decoration:underline; cursor:pointer"
         onclick="parent.toggleTOC()">&lt;&lt;Hide/Show&gt;&gt;</span> Table of
         Contents</p>
   </body>
</html>
```

Related Items: `frame` object.

frameBorder

Value: yes | no | 1 | 0 as strings. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The frameBorder property offers scripted access to a frameset element object's frameborder attribute setting. IE4+ does not respond well to modifying this property after the page has loaded.

Values for the frameBorder property are strings that substitute for Boolean values. Values yes or 1 mean that the border is (supposed to be) turned on; no or 0 turn off the border.

Example

The default value for the frameBorder property is yes. You can use this setting to create a toggle script (which, unfortunately, does not change the appearance in IE). The IE4+-compatible version looks like the following:

```
function toggleFrameScroll(framesetID) {
   var theFrameset = document.all(framesetID);
   if (theFrameset.frameBorder == "yes") {
      theFrameset.frameBorder = "no";
   } else {
      theFrameset.frameBorder = "yes";
   }
}
```

Related Items: frame.frameBorder properties.

frameSpacing

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The frameSpacing property of a frameset element object lets you read the spacing (in pixels) between frames of a frameset. If you do not specify a framespacing attribute in the frameset's tag, the property is empty, rather than reflecting the actual border thickness applied by default (usually 2).

Example

Even though the property is read/write in IE4+, changing the value does not change the thickness of the frame spacing you see in the browser. If you need to find the spacing as set by the tag's attribute, a script reference from one of the frame's documents would look like the following:

```
var spacing = parent.document.all.outerFrameset.frameSpacing;
```

Related Items: frameset.border property.

iframe Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	Methods	Event Handlers
align		
allowTransparency		
contentDocument		
contentWindow		
height		
hspace		
longDesc		
marginHeight		
marginWidth		
name		
scrolling		
src		
vspace		
width		

Syntax

Accessing properties or methods of an `iframe` element object from a containing document:

```
(IE4+)     document.all.iframeID. property | method([parameters])
(IE4+/NN6) window.frames["iframeName"]. property | method([parameters])
(IE5+/W3C) document.getElementById("iframeID"). property | method([parameters])
```

Accessing properties of methods of an `iframe` element from a document inside the `iframe` element:

```
(IE4+)     parent.document.all.iframeID. property | method([parameters])
(IE5+/W3C) parent.document.getElementById("iframeID"). property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About this object

An `iframe` element allows HTML content from a separate source to be loaded within the body of another document. In some respects, the NN4 `layer` element was a precursor to the `iframe` concept, but unlike the `layer`, an `iframe` element is not inherently positionable. It is positionable the same way as any other HTML element, by assigning positioning attributes to a stylesheet associated with the `iframe`. Without explicit positioning, an `iframe` element appears in the body of a document in normal source code order of elements. Unlike a frame of a frameset, you can place an `iframe` arbitrarily in the middle of any document. If the `frame` changes size under script control, the surrounding content moves out of the way or cinches up.

What truly separates the `iframe` apart from other HTML elements is its ability to load and display external HTML files and, with the help of scripts, have different pages loaded into the `iframe` without disturbing the rest of the content of the main document. Pages loaded into the `iframe` can also have scripts and any other features that you may like to put into an HTML document (including XML in IE for Windows).

iframe

The iframe element has a rich set of attributes that let the HTML author control the look, size (height and width), and, to some degree, behavior of the frame. Most of those are accessible to scripts as properties of an iframe element object.

It is important to bear in mind that an iframe element is in many respects like a frame element, especially when it comes to window kinds of relationships. If you plant an iframe element in a document of the main window, that element shows up in the main window's object model as a frame, accessible via common frames terminology:

```
window.frames[i]
window.frames[frameName]
```

Within that iframe frame object is a document and all its contents. All references to the document objects inside the iframe must flow through the "portal" of the iframe frame.

Conversely, scripts in the document living inside an iframe can communicate with the main document via the parent reference. Of course, you cannot replace the content of the main window with another HTML document (using location.href, for instance) without destroying the iframe that was in the original document.

Properties

align

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The align property governs how an iframe element aligns itself with respect to surrounding content on the page. Two of the possible values (left and right) position the iframe along the left and right edge (respectively) of the iframe's containing element (usually the body). Just as with an image, when an iframe is floated along the left and right edges of a container, other content wraps around the element. Table 16-5 shows all possible values and their meanings.

Table 16-5: Values of the align Property

Value	*Description*
absbottom	Aligns the bottom of the iframe with the imaginary line that extends along character descenders of surrounding text.
absmiddle	Aligns the middle of the iframe with the center point between the surrounding text's top and absbottom.
baseline	Aligns the bottom of the iframe with the baseline of surrounding text.
bottom	Same as baseline in IE.
left	Aligns the iframe flush with left edge of the containing element.
middle	Aligns the imaginary vertical centerline of surrounding text with the same for the iframe element.
right	Aligns the iframe flush with the right edge of the containing element.
texttop	Aligns the top of the iframe element with the imaginary line that extends along the tallest ascender of surrounding text.
top	Aligns the top of the iframe element with the surrounding element's top.

As your script changes the value of the `align` property, the page automatically reflows the content to suit the new alignment.

Example

The default setting for an `iframe` alignment is `baseline`. A script can shift the `iframe` to be flush with the right edge of the containing element as follows:

```
document.getElementById("iframe1").align = "right";
```

Related Items: `iframe.hspace`, `iframe.vspace` properties.

allowTransparency

Value: Boolean. Read/Write
Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-

The `allowTransparency` property indicates whether or not the frame's background is transparent. By setting this property to `true`, you allow a background color or image to show through the transparent frame.

contentDocument

Value: `document` object reference. Read-Only
Compatibility: WInIE-, MacIE-, NN6+, Moz1+, Safari1+

The `contentDocument` property of an `iframe` element object is nothing more than a reference to the document contained by that frame. If your script finds that it has a reference to an `iframe` element object, you can use the `contentDocument` property to get a valid reference to the document, and therefore any other content of the frame.

Example

A document script might be using the ID of an `iframe` element to read or adjust one of the element properties; it then needs to perform some action on the content of the page through its `document` object. You can get the reference to the `document` object via a statement, such as the following:

```
var doc = document.getElementById("Frame3").contentDocument;
```

Then your script can, for example, dive into a form in the document:

```
var val = doc.mainForm.entry.value;
```

Related Items: `contentWindow` property; `document` object.

contentWindow

Value: `document` object reference. Read-Only
Compatibility: WInIE5.5+, MacIE-, NN7+, Moz1+, Safari-

The `contentWindow` property of an `iframe` element object serves as a reference to the `window` object generated by the frame. You can then use this `window` object as a means of accessing the `document` object and any document elements.

Related Items: `contentDocument` property; `window` object.

iframe.height

height
width

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `height` and `width` properties provide access to the height and width of the `iframe` object, and allow you to alter the size of the frame. Both properties are specified in pixels.

hspace
vspace

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

These IE-specific properties allow for margins to be set around an `iframe` element. In general, `hspace` and `vspace` properties (and their HTML attributes) have been replaced by CSS margins and padding. These properties and their attributes are not recognized by any W3C standard (including HTML 4.0).

Values for these properties are integers representing the number of pixels of padding between the element and surrounding content. The `hspace` value assigns the same number of pixels to the left and right sides of the element; the `vspace` value is applied to both the top and bottom edges. Scripted changes to these values have no effect in WinIE5+.

Related Items: `style.padding` property.

longDesc

Value: URL string. Read/Write
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The `longDesc` property is the scripted equivalent of the `longdesc` attribute of the `<iframe>` tag. This HTML 4.0 attribute is intended to provide browsers with a URL to a document that contains a long description of the element. Future browsers can use this feature to provide information about the frame for visually impaired site visitors.

marginHeight
marginWidth

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Browsers tend to automatically insert content within a frame by adding a margin between the content and the edge of the frame. These values are represented by the `marginHeight` (top and bottom edges) and `marginWidth` (left and right edges) properties. Although the properties are not read-only, changing the values after the frameset has loaded does not alter the appearance of the document in the frame. If you need to alter the margin(s) of a document inside a frame, adjust the `document.body.style` margin properties.

Also be aware that although the default values of these properties are empty (that is, when no `marginheight` or `marginwidth` attributes are set for the `<iframe>` tag), margins are built into the page. The precise pixel count of those margins varies with different operating systems.

Related Items: `style` object (Chapter 26).

name

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `name` property is the identifier associated with the frame for use as a frame reference. Scripts can reference the frame through the `name` property (e.g., `window.frames["myIframe"]`), which is typically assigned via the `name` attribute.

scrolling

Value: `yes | no | 1 | 0` as strings. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `scrolling` property lets scripts turn scroll bars on and off inside an `iframe` element. By default, scrolling is turned on unless overridden by the `scroll` attribute of the `<iframe>` tag.

Values for the `scrolling` property are strings that substitute for Boolean values. Values `yes` or `1` mean that scroll bars are visible (provided there is more content than can be viewed without scrolling); `no` or `0` hide scrollbars in the frame. IE4+ also recognizes (and sets as default) the auto value.

Example

The following `toggleIFrameScroll()` function accepts a string of the `iframe` element's ID as a parameter and switches between on and off scrollbars in the `iframe`. The `if` condition checks whether the property is set to something other than `no`. This test allows the condition to evaluate to `true` if the property is set to either `auto` (the first time) or `yes` (as set by the function).

```
function toggleFrameScroll(frameID) {
   // IE5 & NN6 version
   var theFrame = document.getElementById(frameID);
   // IE4+ version
   // var theFrame = document.all[frameID]
   if (theFrame.scrolling != "no") {
      theFrame.scrolling = "no";
   } else {
      theFrame.scrolling = "yes";
   }
}
```

Related Items: `frame.scrolling` property.

src

Value: URL string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `src` property of an `iframe` element object offers an additional way of navigating to a different page within an inline frame (that is, other than assigning a new URL to the `location.href` property of the frame object). Remember that the `src` property belongs to the `iframe` element object, not the `window` object it represents. Therefore, references to the `src` property must be via the element's ID and/or node hierarchy.

Example

For best results, use fully formed URLs as value for the `src` property, as shown here:

```
document.getElementById("myIframe").src = "http://www.dannyg.com";
```

Relative URLs and `javascript:` pseudo-URLs also work most of the time.

Related Items: `location.href` property.

popup Object

Properties	*Methods*	*Event Handlers*
`document`	`hide()`	
`isOpen`	`show()`	

Syntax

Creating a `popup` object:

```
var popupObj = window.createPopup()
```

Accessing properties or methods of a `popup` object from a document in the window that created the pop-up:

```
popupObj.property | method([parameters])
```

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

About this object

A `popup` object is a chrome-less window space, which overlaps the window whose document generates the pop-up. A pop-up also appears in front of any dialog boxes. Unlike the dialog box windows generated via IE's `showModalDialog()` and `showModelessDialog()` methods, your scripts must not only create the window, but also put content into it, and then define where on the screen and how big it will be.

Because the pop-up window has no chrome (that is, title bar, resize handles, and so forth), you should populate its content with a border and/or background color so that it stands out from the main window's content. The following statements reflect a typical sequence of creating, populating, and showing a `popup` object:

```
var popup = window.createPopup();
var popupBody = popup.document.body;
popupBody.style.border = "solid 2px black";
popupBody.style.padding = "5px";
popupBody.innerHTML = "<p>Here is some text in a popup window</p>";
popup.show(200,100, 200, 50, document.body);
```

The pop-up window that IE creates is, in fact, a window, but only from the point of view of the document that it contains. In other words, while the number of properties and methods for the `popup` object is small, the `parentWindow` property of the document inside the pop-up

points to a genuine window property. Even so, be aware that this pop-up does not appear as a distinct window among windows listed in the Windows Taskbar. If a user clicks outside of the pop-up or switches to another application, the pop-up disappears, and you must reinvoke the show() method by script (complete with dimension and position parameters) to force the pop-up to reappear.

When you assign content to a pop-up, you are also responsible for making sure that the content fits the size of the pop-up you specify. If the content runs past the rectangular space (body text word wraps within the pop-up's rectangle), no scroll bars appear.

Properties

document

Value: document object reference. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

Use the document property as a gateway to the content of a pop-up window. This property is the only access point available from the script that creates the pop-up to the pop-up itself. The most common application of this property is to set document properties governing the content of the pop-up window. For example, to give the pop-up a border (because the pop-up itself has no window chrome), the script that creates the window can assign values to the style property of the document in the pop-up window, as follows:

```
myPopup.document.body.style.border = "solid 3px gray";
```

Beware that the document object of a pop-up window may not implement the full flexibility you know about primary window document objects. For example, you are not allowed to assign a URL to the document.URL property in a pop-up window.

Example

Use The Evaluator (Chapter 13) to experiment with the popup object and its properties. Enter the following statements into the top text box. The first statement creates a pop-up window, whose reference is assigned to the a global variable. Next, a reference to the body of the pop-up's document is preserved in the b variable for the sake of convenience. Further statements work with these two variables.

```
a = window.createPopup()
b = a.document.body
b.style.border = "solid 2px black"
b.style.padding = "5px"
b.innerHTML = "<P>Here is some text in a popup window</P>"
a.show(200,100, 200, 50, document.body)
```

See the description of the show() method for details on the parameters.

Related Items: document object.

isOpen

Value: Boolean. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

While a pop-up window is visible, its isOpen property returns true; otherwise the property returns false. Because any user action in the browser causes the pop-up to hide itself, the property is useful only for script statements that are running on their own after the pop-up is made visible.

Example

Use The Evaluator (Chapter 13) to experiment with the `isOpen` property. Enter the following statements into the top text box. The sequence begins with a creation of a simple pop-up window, whose reference is assigned to the `a` global variable. Note that the final statement is actually two statements, designed so that the second statement executes while the pop-up window is still open.

```
a = window.createPopup();
a.document.body.innerHTML = "<p>Here is a popup window</p>";
a.show(200,100, 200, 50, document.body); alert("Popup is open:" + a.isOpen);
```

If you then click into the main window to hide the pop-up, you will see a different result if you enter the following statement into the top text box by itself:

```
alert("Popup is open:" + a.isOpen);
```

Related Items: `popup.show()` method.

Methods

`hide()`
`show(`*`left, top, width, height`*`[, `*`positioningElementRef`*`])`

Returns: Nothing.
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

After you have created a `popup` object with the `window.createPopup()` method and populated it with content, you must explicitly show the window via the `show()` method. If the window is hidden because a user clicked the main browser window somewhere, the `show()` method (and all its parameters) must be invoked again. To have a script hide the window, invoke the `hide()` method for the `popup` object.

The first four parameters of the `show()` method are required and define the pixel location and size of the pop-up window. By default, the coordinate space for the `left` and `top` parameters is the video display. Thus, a `left` and `top` setting of zero places the pop-up in the upper-left corner of the video screen. But you can also define a different coordinate space by adding an optional fifth parameter. This parameter must be a reference to an element on the page. To confine the coordinate space to the content region of the browser window, specify the `document.body` object as the positioning element reference.

Example

Listing 16-49 demonstrates both the `show()` and `hide()` methods for a `popup` object. A click of the button on the page invokes the `selfTimer()` function, which acts as the main routine for this page. The goal is to produce a pop-up window that "self-destructs" five seconds after it appears. Along the way, a message in the pop-up counts down the seconds.

A reference to the pop-up window is preserved as a global variable, called `popup`. After the `popup` object is created, the `initContent()` function stuffs the content into the pop-up by way of assigning `style` properties and some `innerHTML` for the body of the document that is automatically created when the pop-up is generated. A `span` element is defined so that another function later on can modify the content of just that segment of text in the pop-up. Notice that the assignment of content to the pop-up is predicated on the pop-up window having been initialized (by virtue of the `popup` variable having a value assigned to it) and that the pop-up window is not showing. Although invoking `initContent()` under any other circumstances is probably impossible, the validation of the desired conditions is good programming practice.

Back in `selfTimer()`, the `popup` object is displayed. Defining the desired size requires some trial and error to make sure the pop-up window comfortably accommodates the text that is put into the pop-up in the `initContent()` function.

With the pop-up window showing, now is the time to invoke the `countDown()` function. Before the function performs any action, it validates that the pop-up has been initialized and is still visible. If a user clicks the main window while the counter is counting down, this changes the value of the `isOpen` property to `false`, and nothing inside the `if` condition executes.

This `countDown()` function grabs the inner text of the `span` and uses `paresInt()` to extract just the integer number (using base 10 numbering, because we're dealing with zero-leading numbers that can potentially be regarded as octal values). The condition of the `if` construction decreases the retrieved integer by one. If the decremented value is zero, then the time is up, and the pop-up window is hidden with the `popup` global variable returned to its original, `null` value. But if the value is other than zero, then the inner text of the `span` is set to the decremented value (with a leading zero), and the `setTimeout()` method is called upon to reinvoke the `countDown()` function in one second (1,000 milliseconds).

Listing 16-49: **Hiding and Showing a Pop-up**

```
<html>
   <head>
      <title>popup Object</title>
      <script type="text/javascript">
      var popup;
      function initContent() {
         if (popup && !popup.isOpen) {
            var popBody = popup.document.body;
            popBody.style.border = "solid 3px red";
            popBody.style.padding = "10px";
            popBody.style.fontSize = "24pt";
            popBody.style.textAlign = "center";
            var bodyText = "<P>This popup will self-destruct in ";
            bodyText += "<span id='counter'>05<\/span>";
            bodyText += " seconds...<\/P>";
            popBody.innerHTML = bodyText;
         }
      }
      function countDown() {
         if (popup && popup.isOpen) {
            var currCount = parseInt(popup.document.all.counter.innerText, 10);
            if (--currCount == 0) {
               popup.hide();
               popup = null;
            } else {
               popup.document.all.counter.innerText = "0" + currCount;
               setTimeout("countDown()", 1000);
            }
         }
      }
      function selfTimer() {
         popup = window.createPopup();
         initContent();
```

Continued

Listing 16-49 *(continued)*

```
            popup.show(200,200,400,100,document.body);
            setTimeout("countDown()", 1000);
         }
         </script>
      </head>
      <body>
         <form>
            <input type="button" value="Impossible Mission"
            onclick="selfTimer()" />
         </form>
      </body>
</html>
```

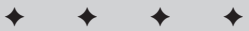
The `hide()` method here is invoked by a script that is running while the pop-up window is showing. Because a pop-up window automatically goes away if a user clicks the main window, it is highly unlikely that the `hide()` method would ever be invoked by itself in response to user action in the main window. If you want a script in the pop-up window to close the pop-up, use `parentWindow.close()`.

Related Items: `popup.isOpen` property, `window.createPopup()` method.

✦ ✦ ✦

CHAPTER 17

Location and History Objects

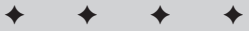


In This Chapter

Loading new pages and other media types via the `location` object

Security restrictions across frames

Navigating through the browser history under script control



Not all objects in the document object model are "things" you can see in the content area of the browser window. Each browser window or frame maintains a bunch of other information about the page you are currently visiting and where you have been. The URL of the page you see in the window is called the *location*, and browsers store this information in the `location` object. As you surf the Web, the browser stores the URLs of your past pages in the `history` object. You can manually view what that object contains by looking in the browser menu that enables you to jump back to a previously visited page. This chapter is all about these two nearly invisible, but important, objects.

Not only are these objects valuable to your browser, but they are also valuable to snoopers who might want to write scripts to see what URLs you're viewing in another frame or the URLs of other sites you've visited in the last dozen mouse clicks. As a result, security restrictions built into browsers limit access to some of these objects' properties (unless you use signed scripts in NN4+). For older browsers, these properties simply are not available from a script.

location Object

Properties	*Methods*	*Event Handlers*
`hash`	`assign()`	None
`host`	`reload()`	
`hostname`	`replace()`	
`href`		
`pathname`		
`port`		
`protocol`		
`search`		

Syntax

Loading a new document into the current window:

```
[window.]location.href = "URL";
```

Accessing `location` object properties or methods:

```
[window.]location.property | method([parameters])
```

About this object

In its place one level below `window`-style objects in the original document object hierarchy, the `location` object represents information about the URL of any currently open window or of a specific frame. A multiple-frame window displays the parent window's URL in the Location (or Address) field of the browser. Each frame also has a location associated with it, although you may not see any overt reference to the frame's URL in the browser. To get URL information about a document located in another frame, the reference to the `location` object must include the window frame reference. For example, if you have a window consisting of two frames, Table 17-1 shows the possible references to the `location` objects for all frames comprising the Web presentation.

Scripts cannot alter the URL displayed in the browser's Location/Address box. For security and privacy reasons, that text box cannot display anything other than the URL of a current page or URL in transit.

Table 17-1: Location Object References in a Two-Frame Browser Window

Reference	*Description*
`location` (or `window.location`)	URL of frame displaying the document that runs the script statement containing this reference
`parent.location`	URL information for parent window that defines the `<frameset>`
`parent.frames[0].location`	URL information for first visible frame
`parent.frames[1].location`	URL information for second visible frame
`parent.`*`otherFrameName`*`.location`	URL information for another named frame in the same frameset

Most properties of a `location` object deal with network-oriented information. This information involves various data about the physical location of the document on the network including the host server, the protocol being used, and other components of the URL. Given a complete URL for a typical WWW page, the `window.location` object assigns property names to various segments of the URL, as shown here:

```
http://www.example.com:80/promos/newproducts.html#giantGizmo
```

Property	*Value*
protocol	"http:"
hostname	"www.example.com"
port	"80"
host	"www.example.com:80"
pathname	"/promos/newproducts.html"
hash	"#giantGizmo"
href	"http://www.example.com:80/promos newproducts.html#giantGizmo"

The `window.location` object is handy when a script needs to extract information about the URL, perhaps to obtain a base reference on which to build URLs for other documents to be fetched as the result of user action. This object can eliminate a nuisance for Web authors who develop sites on one machine and then upload them to a server (perhaps at an Internet service provider) with an entirely different directory structure. By building scripts to construct base references from the directory location of the current document, you can construct the complete URLs for loading documents. You don't have to manually change the base reference data in your documents as you shift the files from computer to computer or from directory to directory. To extract the segment of the URL and place it into the enclosing directory, use the following:

```
var baseRef = location.href.substring(0,location.href.lastIndexOf("/") + 1);
```

Security alert: To allay fears of Internet security breaches and privacy invasions, scriptable browsers prevent your script in one frame from retrieving `location` object properties from other frames whose domain and server are not your own (unless you use signed scripts in NN4+ or the user has set the IE browser to trust your site). This restriction puts a damper on many scripters' well-meaning designs and aids for Web watchers and visitors. If you attempt such property accesses, however, you receive an "access denied" (or similar) security warning dialog box.

Setting the value of some `location` properties is the preferred way to control which document gets loaded into a window or frame. Though you may expect to find a method somewhere in JavaScript that contains a plain language "Go" or "Open" word (to simulate what you see in the browser menu bar), you "point your browser" to another URL by setting the `window.location.href` property to that URL, as in

```
window.location.href = "http://www.dannyg.com/";
```

The equals assignment operator (=) in this kind of statement is a powerful weapon. In fact, setting the `location.href` object to a URL of a different MIME type, such as one of the variety of sound and video formats, causes the browser to load those files into the plug-in or helper application designated in your browser's settings. The `location.assign()` method was originally intended for internal use by the browser, but it is available for scripters (although I don't recommend using it for navigation). Internet Explorer's object model includes a `window.navigate()` method that also loads a document into a window, but you can't use it for cross-browser applications.

Two other methods complement the `location` object's capability to control navigation. One method is the script equivalent of clicking Reload; the other method enables you to replace the current document's entry in the history with that of the next URL of your script's choice.

Properties

hash

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The *hash mark* (#) is a URL convention that directs the browser to an anchor located in the document. Any name you assign to an anchor (with the `<a name="..."> ...</a>` tag pair) becomes part of the URL after the hash mark. A `location` object's `hash` property is the name of the anchor part of the current URL (which consists of the hash mark and the name).

If you have written HTML documents with anchors and directed links to navigate to those anchors, you have probably noticed that although the destination location shows the anchor as part of the URL (for example, in the Location field), the window's anchor value does not change as the user manually scrolls to positions in the document where other anchors are defined. An anchor appears in the URL only when the window has navigated there as part of a link or in response to a script that adjusts the URL.

Just as you can navigate to any URL by setting the `window.location.href` property, you can navigate to another hash in the same document by adjusting only the `hash` property of the location without the hash mark (as shown in the following example). Such navigation, even within a document, sometimes causes IE to reload the document.

Listing 17-1 demonstrates how to use the `hash` property to access the anchor part of a URL. When you load the script in Listing 17-1, adjust the size of the browser window so only one section is visible at a time. When you click a button, the script navigates to the next logical section in the progression and eventually takes you back to the top.

Listing 17-1: **A Document with Anchors**

```
<html>
   <head>
      <title>location.hash Property</title>
      <script type="text/javascript">
      function goNextAnchor(where) {
         window.location.hash = where;
      }
      </script>
   </head>
   <body>
      <h1><a id="start" name="start">Top</a></h1>
      <form>
         <input type="button" name="next" value="NEXT"
         onclick="goNextAnchor('sec1')" />
      </form>
      <hr />
      <h1><a id="sec1" name="sec1">Section 1</a></h1>
      <form>
         <input type="button" name="next" value="NEXT"
         onclick="goNextAnchor('sec2')" />
      </form>
```

```
      <hr />
      <h1><a id="sec2" name="sec2">Section 2</a></h1>
      <form>
         <input type="button" name="next" value="NEXT"
         onclick="goNextAnchor('sec3')" />
      </form>
      <hr />
      <h1><a id="sec3" name="sec3">Section 3</a></h1>
      <form>
         <input type="button" name="next" value="BACK TO TOP"
         onclick="goNextAnchor('start')" />
      </form>
   </body>
</html>
```

Anchor names are passed as parameters with each button's `onclick` event handler. Instead of going through the work of assembling a `window.location` value in the function by appending a literal hash mark and the value for the anchor, here I simply modify the `hash` property of the current window's location. This is the preferred, cleaner method.

If you attempt to read back the `window.location.hash` property in an added line of script, however, the window's actual URL probably will not have been updated yet, and the browser will appear to be giving your script false information. To prevent this problem in subsequent statements of the same function, construct the URLs of those statements from the same variable values you use to set the `window.location.hash` property — don't rely on the browser to give you the values you expect.

Related Item: `location.href` property.

host

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `location.host` property describes both the hostname and port of a URL. The port is included in the value only when the port is an explicit part of the URL. If you navigate to a URL that does not display the port number in the Location field of the browser, the `location.host` property returns the same value as the `location.hostname` property.

Use the `location.host` property to extract the `hostname:port` part of the URL of any document loaded in the browser. This capability may be helpful for building a URL to a specific document that you want your script to access on the fly.

Use the documents in Listings 17-2 through 17-4 as tools to help you learn the values that the various `window.location` properties return. In the browser, open the file for Listing 17-2. This file creates a two-frame window. The left frame contains a temporary placeholder (see Listing 17-4) that displays some instructions. The right frame has a document (see Listing 17-3) that enables you to load URLs into the left frame and get readings on three different windows available: the parent window (which creates the multiframe window), the left frame, and the right frame.

Listing 17-2: **Frameset for the Property Picker**

```
<html>
   <head>
      <title>window.location Properties</title>
   </head>
   <frameset cols="50%,50%" border="1" bordercolor="black">
      <frame name="Frame1" src="lst17-04.htm" />
      <frame name="Frame2" src="lst17-03.htm" />
   </frameset>
</html>
```

Listing 17-3: **Property Picker**

```
<html>
   <head>
      <title>Property Picker</title>
      <script type="text/javascript">
      var isNav = (typeof netscape != "undefined") ? true : false;

      function fillLeftFrame() {
         newURL = prompt("Enter the URL of a document to show in the left
            frame:","");
         if (newURL != null && newURL != "") {
            parent.frames[0].location = newURL;
         }
      }

      function showLocationData(form) {
         for (var i = 0; i <3; i++) {
            if (form.whichFrame[i].checked) {
               var windName = form.whichFrame[i].value;
               break;
            }
         }
         var theWind = "" + windName + ".location";
         if (isNav) {
            netscape.security.PrivilegeManager.enablePrivilege(
               "UniversalBrowserRead");
         }
         var theObj = eval(theWind);
         form.windName.value = windName;
         form.windHash.value = theObj.hash;
         form.windHost.value = theObj.host;
         form.windHostname.value = theObj.hostname;
         form.windHref.value = theObj.href;
         form.windPath.value = theObj.pathname;
         form.windPort.value = theObj.port;
         form.windProtocol.value = theObj.protocol;
         form.windSearch.value = theObj.search;
         if (isNav) {
            netscape.security.PrivilegeManager.disablePrivilege(
               "UniversalBrowserRead");
         }
```

```
   }
   </script>
</head>
<body>
   Click the "Open URL" button to enter the location of an HTML document to
   display in the left frame of this window.
   <form>
      <input type="button" name="opener" value="Open URL..."
      onclick="fillLeftFrame()" />
      <hr />
      <center>
         Select a window/frame. Then click the "Show Location Properties"
         button to view each window.location property value for the desired
         window.
         <p><input type="radio" name="whichFrame" value="parent"
            checked="checked" />Parent window <input type="radio"
            name="whichFrame" value="parent.frames[0]" />Left frame <input
            type="radio" name="whichFrame" value="parent.frames[1]" />This
            frame</p>
         <p><input type="button" name="getProperties"
            value="Show Location Properties"
            onclick="showLocationData(this.form)" /> <input type="reset"
            value="Clear" /></p>
         <table border="2">
            <tr>
               <td align="right">Window:</td>
               <td><input type="text" name="windName" size="30" /></td>
            </tr>
            <tr>
               <td align="right">hash:</td>
               <td><input type="text" name="windHash" size="30" /></td>
            </tr>
            <tr>
               <td align="right">host:</td>
               <td><input type="text" name="windHost" size="30" /></td>
            </tr>
            <tr>
               <td align="right">hostname:</td>
               <td><input type="text" name="windHostname" size="30" /></td>
            </tr>
            <tr>
               <td align="right">href:</td>
               <td><textarea name="windHref" rows="3" cols="30" wrap="soft">
                  </textarea></td>
            </tr>
            <tr>
               <td align="right">pathname:</td>
               <td><textarea name="windPath" rows="3" cols="30" wrap="soft">
                  </textarea></td>
            </tr>
            <tr>
               <td align="right">port:</td>
               <td><input type="text" name="windPort" size="30" /></td>
            </tr>
            <tr>
               <td align="right">protocol:</td>
               <td><input type="text" name="windProtocol" size="30" /></td>
            </tr>
```

Continued

Listing 17-3 *(continued)*

```
            <tr>
               <td align="right">search:</td>
               <td><textarea name="windSearch" rows="3" cols="30"
                  wrap="soft"></textarea></td>
            </tr>
         </table>
      </center>
   </form>
</body>
</html>
```

Listing 17-4: **Placeholder Document for Listing 17-2**

```
<html>
   <head>
      <title>Opening Placeholder</title>
   </head>
   <body>
      Initial placeholder. Experiment with other URLs for this frame (see
      right).
   </body>
</html>
```

For the best results, open a URL to a Web document on the network from the same domain and server from which you load the listings (perhaps your local hard disk). If possible, load a document that includes anchor points to navigate through a long document. Click the Left frame radio button and then click the button that shows all properties. This action fills the table in the right frame with all the available `location` properties for the selected window.

Related Items: `location.port`, `location.hostname` properties.

hostname

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The hostname of a typical URL is the name of the server on the network that stores the document you view in the browser. For most Web sites, the server name includes not only the domain name, but also the `www.` prefix. The hostname does not, however, include the port number if the URL specifies such a number.

See Listings 17-2 through 17-4 for a set of related pages to help you view the hostname data for a variety of other pages.

Related Items: `location.host`, `location.port` properties.

href

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Of all the `location` object properties, `href` (hypertext reference) is probably the one most often called upon in scripting. The `location.href` property supplies a string of the entire URL of the specified `window` object.

Using this property on the left side of an assignment statement is the JavaScript way of opening a URL for display in a window. Any of the following statements can load my Web site's index page into a single-frame browser window:

```
window.location = "http://www.dannyg.com";
window.location.href = "http://www.dannyg.com";
```

At times, you may encounter difficulty by omitting a reference to a window. JavaScript may get confused and reference the `document.location` property. To prevent this confusion, the `document.location` property was deprecated (put on the no-no list) and replaced by the `document.URL` property. In the meantime, you can't go wrong by always specifying a window in the reference.

You should be able to omit the `href` property name when assigning a new URL to the `location` object (for example, `location = "http://www.dannyg.com"`). While this works in most browsers most of the time, some early browsers behave more reliably if you assign a URL explicitly to the `location.href` property. I recommend using `location.href` at all times.

Sometimes you must extract the name of the current directory in a script so another statement can append a known document to the URL before loading it into the window. Although the other `location` object properties yield an assortment of a URL's segments, none of them provides the full URL to the current URL's directory. But you can use JavaScript string manipulation techniques to accomplish this task. Listing 17-5 shows such a possibility.

Depending on your browser, the values for the `location.href` property may be encoded with ASCII equivalents of non-alphanumeric characters. Such an ASCII value includes the `%` symbol and the ASCII numeric value. The most common encoded character in a URL is the space: `%20`. If you need to extract a URL and display that value as a string in your documents, you can safely pass all such potentially encoded strings through the JavaScript `unescape()` function. For example, if a URL is `http://www.example.com/product%20list`, you can convert it by passing it through the `unescape()` function, as in the following example.

```
var plainURL = unescape(window.location.href);
// result = "http://www.example.com/product list";
```

The inverse function, `escape()`, is available for sending encoded strings to CGI programs on servers. See Chapter 34 for more details on these functions.

Listing 17-5 shows how the `href` property can be used to view the directory URL of the current page. This example includes the `unescape()` function in front of the part of the script that captures the URL. This function serves cosmetic purposes by displaying the pathname in alert dialog boxes for browsers that normally display the ASCII-encoded version.

Listing 17-5: Extracting the Directory of the Current Document

```
<html>
   <head>
      <title>Extract pathname</title>
      <script type="text/javascript">
      // general purpose function to extract URL of current directory
      function getDirPath(URL) {
         var result = unescape(URL.substring(0,(URL.lastIndexOf("/")) + 1));
         return result;
      }

      // handle button event, passing work onto general purpose function
      function showDirPath(URL) {
         alert(getDirPath(URL));
      }
      </script>
   </head>
   <body>
      <form>
         <input type="button" value="View directory URL"
         onclick="showDirPath(window.location.href)" />
      </form>
   </body>
</html>
```

Related Items: `location.pathname`, `document.location` properties; `String` object (Chapter 34).

pathname

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The pathname component of a URL consists of the directory structure relative to the server's root volume. In other words, the root (the server name in an `http:` connection) is not part of the pathname. If the URL's path is to a file in the root directory, the `location.pathname` property is a single slash (`/`) character. Any other pathname starts with a slash character, indicating a directory nested within the root. The value of the `location.pathname` property also includes the document name.

See Listings 17-2 through 17-4 earlier in this chapter for a multiple-frame example you can use to view the `location.pathname` property for a variety of URLs of your choice.

Related Item: `location.href` property.

port

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

These days, few consumer-friendly Web sites need to include the port number as part of their URLs. You see port numbers mostly in the less-popular protocols, in URLs to sites used for private development purposes, or in URLs to sites that have no assigned domain names. You can retrieve the value with the `location.port` property. If you extract the value from one

URL and intend to build another URL with that component, be sure to include the colon delimiter between the server's IP address and port number.

If you have access to URLs containing port numbers, use the documents in Listings 17-2 through 17-4 to experiment with the output of the `location.port` property.

Related Item: `location.host` property.

protocol

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The first component of any URL is the protocol used for the particular type of communication. For World Wide Web pages, the Hypertext Transfer Protocol (`http`) is the standard. Other common protocols you may see in your browser include HTTP-Secure (`https`), File Transfer Protocol (`ftp`), File (`file`), and Mail (`mailto`). Values for the `location.protocol` property include not only the name of the protocol, but also the trailing colon delimiter. Thus, for a typical Web page URL, the `location.protocol` property is

```
http:
```

Notice that the usual slashes after the protocol in the URL are not part of the `location.protocol` value. Of all the `location` object properties, only the full URL (`location.href`) reveals the slash delimiters between the protocol and other components.

See Listings 17-2 through 17-4 for a multiple-frame example you can use to view the `location.protocol` property for a variety of URLs. Also try loading an FTP site to see the `location.protocol` value for that type of URL.

Related Item: `location.href` property.

search

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Perhaps you've noticed the long, cryptic URL that appears in the Location/Address field of your browser whenever you ask one of the WWW search services to look up matches for items you enter into the keyword field. The URL starts the regular way — with protocol, host, and pathname values. But following the more traditional URL are search commands that are submitted to the search engine (a CGI program running on the server). You can retrieve or set that trailing search query by using the `location.search` property.

Each search engine has its own formula for query submissions based on the designs of the HTML forms that obtain details from users. These search queries come in an encoded format that appears in anything but plain language. If you plan to script a search query, be sure you fully understand the search engine's format before you start assembling a string to assign to the `location.search` property of a window.

The most common format for search data is a series of name/value pairs. An equal symbol (`=`) separates a name and its value. Multiple name/value pairs have ampersands (`&`) between them. You should use the `escape()` function to convert the data into URL-friendly format, especially when the content includes spaces.

The `location.search` property also applies to any part of a URL after the filename, including parameters being sent to CGI programs on the server.

Passing data among pages via URLs

It is not uncommon to want to preserve some pieces of data that exist in one page so that a script in another page can pick up where the script processing left off in the first page. You can achieve persistence across page loads through one of three techniques: the `document.cookie` (see Chapter 18), variables in framesetting documents, and the search string of a URL. That's really what happens when you visit search and e-commerce sites that return information to your browser. Rather than store, say, your search criteria on the server, they spit the criteria back to the browser as part of the URL. The next time you activate that URL, the values are sent to the server for processing (for example, to send you the next page of search results for a particular query).

Passing data among pages is not limited to client/server communication. You can use the search string strictly on the client-side to pass data from one page to another. Unless some CGI process on the server is programmed to do something with the search string, a Web server regurgitates the search string as part of the location data that comes back with a page. A script in the newly loaded page can inspect the search string (via the `location.search` property) and tear it apart to gather the data and put it into script variables. Take a look at Listings 17-6 through 17-8 to see a powerful application of this technique.

As mentioned in the opening of Chapter 16 about frames, you can force a particular HTML page to open inside the frameset for which it is designed. But with the help of the search string, you can reuse the same framesetting document to accommodate any number of content pages that go into one of the frames (rather than specifying a separate frameset for each possible combination of pages in the frameset). The listings in this section create a simple example of how to force a page to load in a frameset by passing some information about the page to the frameset. Thus, if a user has a URL to one of the content frames (perhaps it has been bookmarked by right-clicking the frame or it comes up as a search engine result), the page appears in its designated frameset the next time the user visits the page.

The fundamental task going on in this scheme has two parts. The first is in each of the content pages where a script checks whether the page is loaded inside a frameset. If the frameset is missing, a search string is composed and appended to the URL for the framesetting document. The framesetting document has its own short script that looks for the presence of the search string. If the string is there, the script extracts the search string data and uses it to load that specific page into the content frame of the frameset.

Listing 17-6 is the framesetting document. The `getSearchAsArray()` function is more complete than necessary for this simple example, but you can use it in other instances to convert any number of name/value pairs passed in the search string (in traditional format of `name1=value1&name2=value2&etc.`) into an array whose indexes are the names (making it easier for scripts to extract a specific piece of passed data). Version branching takes place because, for convenience, the `getSearchAsArray()` function uses text and array methods that don't exist in browsers prior to NN3 or IE4.

Listing 17-6: **A Smart Frameset**

```
<html>
   <head>
      <title>Example Frameset</title>
      <script type="text/javascript">
      // Convert location.search into an array of values
      // indexed by name.
      function getSearchAsArray() {
         var results = new Array();
```

```
         var input = unescape(location.search.substr(1));
         if (input) {
            var srchArray = input.split("&");
            var tempArray = new Array();
            for (var i = 0; i < srchArray.length; i++) {
               tempArray = srchArray[i].split("=");
               results[tempArray[0]] = tempArray[1];
            }
         }
         return results;
      }

      function loadFrame() {
         if (location.search) {
            var srchArray = getSearchAsArray();
            if (srchArray["content"]) {
               self.content.location.href = srchArray["content"];
            }
         }
      }
      </script>
   </head>
   <frameset cols="250,*" onload="loadFrame()">
      <frame name="toc" src="lst17-07.htm" />
      <frame name="content" src="lst17-08.htm" />
   </frameset>
</html>
```

Listing 17-7 is the HTML for the table of contents frame. Nothing elaborate goes on here, but you can see how normal navigation works for this simplified frameset.

Listing 17-7: **The Table of Contents**

```
<html>
   <head>
      <title>Table of Contents</title>
   </head>
   <body bgcolor="#EEEEEE">
      <h3>Table of Contents</h3>
      <hr />
      <ul>
         <li><a href="lst17-08.htm" target="content">Page 1</a></li>
         <li><a href="lst17-08a.htm" target="content">Page 2</a></li>
         <li><a href="lst17-08b.htm" target="content">Page 3</a></li>
      </ul>
   </body>
</html>
```

Listing 17-8 shows one of the content pages. As the page loads, the `checkFrameset()` function is invoked. If the window does not load inside a frameset, the script navigates to the framesetting page, passing the current content URL as a search string. Notice that for browsers that support the `location.replace()` method, the loading of this page on its own does not get recorded to the browser's history and isn't accessed if the user hits the Back button.

Listing 17-8: **A Content Page**

```
<html>
   <head>
      <title>Page 1</title>
      <script type="text/javascript">
      function checkFrameset() {
         var isNav4 = (navigator.appName == "Netscape" &&
            parseInt(navigator.appVersion) == 4);
         if (parent == window) {
            // Don't do anything if running NN4
            // so that the frame can be printed on its own
            if (isNav4 && window.innerWidth == 0) {
               return;
            }
            // Use replace() to keep current page out of history
            location.replace("lst17-06.htm?content=" + escape(location.href));
         }
      }

      // Invoke the function
      checkFrameset();
      </script>
   </head>
   <body>
      <h1>Page 1</h1>
      <hr />
   </body>
</html>
```

In practice, I recommend placing the code for the `checkFrameset()` function and call to it inside an external `.js` library and linking that library into each content document of the frameset. That's why the function assigns the generic `location.href` property to the search string — you can use it on any content page.

The code in Listings 17-6 through 17-8 establishes a frameset containing two frames. In the left frame is a Table of Contents that allows you to navigate among three different pages, the first of which is initially displayed in the right frame. The interesting thing about the example is how you can specify a new page in the `content` parameter of the search property, and then the page is opened within the frameset. For example, the following URL would result in the page hello.htm being opened in the right frame:

```
lst17-06.htm?content=hello.htm
```

In this example URL, the frameset page is first opened due to the inclusion of the file lst17-06.htm, while the hello.htm file is specified as the value of the `content` parameter.

Related Item: `location.href` property.

Methods

assign("*URL*")

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

In earlier discussions about the `location` object, I said that you navigate to another page by assigning a new URL to the `location` object or `location.href` property. The `location.assign()` method does the same thing. In fact, when you set the `location` object to a URL, JavaScript silently applies the `assign()` method. No particular penalty or benefit comes from using the `assign()` method, except perhaps to make your code more understandable to others.

Related Item: `location.href` property.

reload(*unconditionalGETBoolean*)

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

The `location.reload()` method may be named inappropriately because it makes you think of the Reload/Refresh button in the browser toolbar. The `reload()` method is actually more powerful than the Reload/Refresh button (a soft reload) in that it clears form control values that might otherwise survive the Reload/Refresh button. Note that MacIE and Safari do not preserve form control settings even with a soft reload.

Most form elements retain their screen states when you click Reload/Refresh. Text and `textarea` objects maintain whatever text is inside them; radio buttons and checkboxes maintain their checked status; `select` objects remember which item is selected. About the only items the Reload/Refresh button destroys are global variable values and any settable, but not visible, property (for example, the value of a `hidden input` object). I call this kind of reload a *soft reload*.

Browsers are frustratingly irregular about the ways they reload a document in the memory cache. In theory, an application of the `location.reload()` method should retrieve the page from the cache if the page is still available there (while the `history.go(0)` method should be even gentler, preserving form element settings). Adding a `true` parameter to the method is supposed to force an *unconditional GET* to the server, ignoring the cached version of the page. Yet when it is crucial for your application to get a page from the cache (for speed) or from the server (to guarantee a fresh copy), the browser behaves in just the opposite way you want it to behave. Meta tags supposedly designed to prevent caching of a page rarely, if ever, work. Some scripters have had success in reloading the page from the server by setting `location.href` to the URL of the page, plus a slightly different search string (for example, based on a string representation of the `Date` object) so that there is no match for the URL in the cache.

The bottom line is to be prepared to try different schemes to achieve the effect you want. And also be prepared to not get the results you need.

Listing 17-9 provides a means of testing out the different outcomes of a soft reload versus a hard reload. Open this example page in a browser and click a radio button. Then enter some new text and make a choice in the `select` object. Clicking the Soft Reload/Refresh button invokes a method that reloads the document as if you had clicked the browser's Reload/Refresh button. It also preserves the visible properties of form elements. The Hard Reload button invokes the `location.reload()` method, which resets all objects to their default settings.

Listing 17-9: **Hard versus Soft Reloading**

```
<html>
   <head>
      <title>Reload Comparisons</title>
      <script type="text/javascript">
      function hardReload() {
         location.reload(true);
      }
      function softReload() {
         history.go(0);
      }
      </script>
   </head>
   <body>
      <form name="myForm">
         <input type="radio" name="rad1" value="1" />Radio 1<br />
         <input type="radio" name="rad1" value="2" />Radio 2<br />
         <input type="radio" name="rad1" value="3" />Radio 3
         <p><input type="text" name="entry" value="Original" /></p>
         <p><select name="theList">
               <option>Red</option>
               <option>Green</option>
               <option>Blue</option>
            </select></p>
         <hr />
         <input type="button" value="Soft Reload" onclick="softReload()" />
         <input type="button" value="Hard Reload" onclick="hardReload()" />
      </form>
   </body>
</html>
```

Related Item: `history.go()` method.

`replace("`*URL*`")`

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

In a complex Web site, you may have pages that you do not want to appear in the user's history list. For example, a registration sequence may lead the user to one or more intermediate HTML documents that won't make much sense to the user later. You especially don't want users to see these pages again if they use the Back button to return to a previous URL. The `location.replace()` method navigates to another page, but it does not let the current page stay in the queue of pages accessible via the Back button.

Although you cannot prevent a document from appearing in the history list while the user views that page, you can instruct the browser to load another document into the window and replace the current history entry with the entry for the new document. This trick does not empty the history list but instead removes the current item from the list before the next URL is loaded. Removing the item from the history list prevents users from seeing the page again by clicking the Back button later.

Listing 17-10 shows how to use the `replace()` method to direct a Web browser to a new URL. Calling the `location.replace()` method navigates to another URL similarly to assigning a URL to the location. The difference is that the document doing the calling doesn't appear in the history list after the new document loads. Check the history listing (in your browser's usual spot for this information) before and after clicking Replace Me in Listing 17-10.

Listing 17-10: **Invoking the location.replace() Method**

```
<html>
   <head>
      <title>location.replace() Method</title>
      <script type="text/javascript">
      function doReplace() {
         location.replace("lst17-01.htm");
      }
      </script>
   </head>
   <body>
      <form name="myForm">
         <input type="button" value="Replace Me" onclick="doReplace()" />
      </form>
   </body>
</html>
```

Related Item: `history` object.

history Object

Property	*Method*	*Event Handler*
`current`	`back()`	(None)
`length`	`forward()`	
`next`	`go()`	
`previous`		

Syntax

Accessing `history` object properties or methods:

```
[window.]history.property | method([parameters])
```

About this object

As a user surfs the Web, the browser maintains a list of URLs for the most recent stops. This list is represented in the scriptable object model by the `history` object. A script cannot surreptitiously extract actual URLs maintained in that list unless you use signed scripts (in NN4+ — see Chapter 46 on the CD-ROM) and the user grants permission. Under unsigned conditions, a script can methodically navigate to each URL in the history (by relative number or by stepping back one URL at a time), in which case the user sees the browser navigating on its own as if possessed by a spirit. Good Netiquette dictates that you do not navigate a user outside of your Web site without the user's explicit permission.

One application for the `history` object and its `back()` or `go()` methods is to provide the equivalent of a Back button in your HTML documents. That button triggers a script that checks for any items in the history list and then goes back one page. Your document doesn't have to know anything about the URL from which the user lands at your page.

As of NN4, the behavior of the Back and Forward buttons is also available through a pair of window methods: `window.back()` and `window.forward()`. The `history` object methods are not specific to a frame that is part of the reference. When the `parent.frameName.history.back()` method reaches the end of history for that frame, further invocations of that method are ignored.

IE's history mechanism is not localized to a particular frame of a frameset. Instead, the `history.back()` and `history.forward()` methods mimic the physical act of clicking the toolbar buttons. If you want to ensure cross-browser, if not cross-generational, behavior in a frameset, address references to the `history.back()` and `history.forward()` methods to the parent window.

You should use the `history` object and its methods with extreme care. Your design must be smart enough to "watch" what the user is doing with your pages (for example, by checking the current URL before navigating with these methods). Otherwise, you run the risk of confusing your user by navigating to unexpected places. Your script can also get into trouble because it cannot detect where the current document is in the Back–Forward sequence in history.

Properties

current
next
previous

Value: String. Read-Only
Compatibility: WinIE-, MacIE-, NN4+, Moz1+, Safari-

To know where to go when you click the Back and Forward buttons, the browser maintains a list of URLs visited. To someone trying to invade your privacy and see what sites and pages you frequent, this information is valuable. That's why the three properties that expose the actual URLs in the history list are restricted to pages with signed scripts and whose visitors have given permission to read sensitive browser data (see Chapter 46 on the CD-ROM).

With signed scripts and permission, you can look through the entire array of history entries in any frame or window. Because the list is an array, you can extract individual items by index value. For example, if the array has 10 entries, you can see the fifth item by using normal array indexing methods:

```
var fifthEntry = window.history[4];
```

No property or method exists that directly reveals the index value of the currently loaded URL, but you can script an educated guess by comparing the values of the current, next, and previous properties of the `history` object against the entire list.

I personally don't like some unknown entity watching over my shoulder while I'm on the Net, so I respect that same feeling in others and therefore discourage the use of these powers unless the user is given adequate warning. The signed script permission dialog box does not offer enough detail about the consequences of revealing this level of information.

These properties were available in some form in NN3. Access to them required a short-lived security scheme called *data tainting*. That mechanism was never implemented fully and was replaced by signed scripts.

Related Item: `history.length` property.

length

Value: Number. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Use the `history.length` property to count the items in the history list. Unfortunately, this nugget of information is not particularly helpful in scripting navigation relative to the current location because your script cannot extract anything from the place in the history queue where the current document is located. If the current document is at the top of the list (the most recently loaded), you can calculate relative to that location. But users can use the Go/View menu to jump around the history list as they like. The position of a listing in the history list does not change by virtue of navigating back to that document. A `history.length` of `1`, however, indicates that the current document is the first one the user loaded since starting the browser software.

Safari 1.0 uniformly reports `history.length` as zero.

Listing 17-11 shows how to use the `length` property to notify users of how many pages they've visited.

Listing 17-11: **A Browser History Count**

```
<html>
   <head>
      <title>History Object</title>
      <script type="text/javascript">
      function showCount() {
         var histCount = window.history.length;
         if (histCount > 5) {
            alert("My, my, you\'ve been busy. You have visited " + histCount +
               " pages so far.");
         } else {
            alert("You have been to " + histCount + " Web pages this
               session.");
         }
      }
      </script>
   </head>
   <body>
      <form>
         <input type="button" name="activity" value="My Activity"
         onclick="showCount()" />
      </form>
   </body>
</html>
```

Related Items: None.

Methods

back()
forward()

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Prior to Navigator 4, the `back()` method acted identically to clicking the Back button in Navigator browsers. In Navigator 4, however, the `history.back()` method became window/frame-specific. Therefore, if you direct successive `back()` methods to a frame within a frameset, the method is ignored once it reaches the first document to be loaded into that frame. The Back button (and the `window.back()` method) unload the frameset and continue taking you back through the browser's global history.

If you deliberately lead a user to a dead end in your Web site, you should make sure that the HTML document provides a way to navigate back to a recognizable spot. Because you can easily create a new window that has no toolbar or menu bar (non-Macintosh browsers), you may end up stranding your users because they have no way of navigating out of a cul-de-sac in such a window. A button in your document should give the user a way back to the last location.

Unless you need to perform some additional processing prior to navigating to the previous location, you can simply place this method as the parameter to the event handler attribute of a button definition. To guarantee compatibility across all browsers, direct this method at the parent document when used from within a frameset.

Less likely to be scripted than the `history.back()` action is the method that performs the opposite action: navigating forward one step in the browser's history list. The only time you can confidently use the `history.forward()` method is to balance the use of the `history.back()` method in the same script — where your script closely keeps track of how many steps the script heads in either direction. Use the `history.forward()` method with extreme caution, and only after performing extensive user testing on your Web pages to make sure that you've covered all user possibilities. The same cautions about differences introduced in NN4 for `history.back()` apply equally to `history.forward()`: Forward progress extends only through the history listing for a given window or frame, not the entire browser history list.

Listings 17-12 and 17-13 provide a little workshop in which you can test the behavior of a variety of backward and forward navigation in different browsers. Some features work only in NN4+.

Listing 17-12: **Navigation Lab Frameset**

```
<html>
   <head>
      <title>Back and Forward</title>
   </head>
   <frameset cols="45%,55%">
      <frame name="controller" src="lst17-13.htm" />
      <frame name="display" src="lst17-01.htm" />
   </frameset>
</html>
```

Listing 17-13: **Navigation Lab Control Panel**

```
<html>
   <head>
      <title>Lab Controls</title>
   </head>
   <body>
      <b>Load a series of documents into the right frame by clicking some of
      these links (make a note of the sequence you click on):</b>
      <p><a href="lst17-01.htm" target="display">Listing 17-1</a><br />
         <a href="lst17-05.htm" target="display">Listing 17-5</a><br />
         <a href="lst17-09.htm" target="display">Listing 17-9</a><br /></p>
      <hr />
      <form name="input">
         <b>Click on the various buttons below to see the results in this
         frameset:</b>
         <ul>
```

Continued

Listing 17-13 *(continued)*

```
            <li><b>NN4+</b> Substitute for toolbar buttons --
               <tt>window.back()</tt> and <tt>window.forward()</tt>:<input
               type="button" value="Back" onclick="window.back()" /><input
               type="button" value="Forward" onclick="window.forward()" /></li>
            <li><tt>history.back()</tt> and <tt>history.forward()</tt> for
               righthand frame:<input type="button" value="Back"
               onclick="parent.display.history.back()" /><input type="button"
               value="Forward" onclick="parent.display.history.forward()" />
               </li>
            <li><tt>history.back()</tt> for this frame:<input type="button"
               value="Back" onclick="history.back()" /></li>
            <li><tt>history.back()</tt> for parent:<input type="button"
               value="Back" onclick="parent.history.back()" /></li>
         </ul>
      </form>
   </body>
</html>
```

Related Items: `history.go()` method.

go(*relativeNumber* | "*URLOrTitleSubstring*")

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Use the `history.go()` method to script navigation within the history list currently stored in the browser. If you elect to use a URL as a parameter, however, that precise URL must already exist in the history listing. Therefore, do not regard this method as an alternate to setting the `window.location` object to a brand-new URL.

For navigating `n` steps in either direction along the history list, use the *`relativeNumber`* parameter of the `history.go()` method. This number is an integer value that indicates which item in the list to use, relative to the current location. For example, if the current URL is at the top of the list (that is, the Forward button in the toolbar is dimmed), you need to use the following method to jump to the URL two items backward in the list:

```
history.go(-2);
```

In other words, the current URL is the equivalent of `history.go(0)` (a method that reloads the window). A positive integer indicates a jump that many items forward in the history list. Thus, `history.go(-1)` is the same as `history.back()`, whereas `history.go(1)` is the same as `history.forward()`.

Alternatively, you can specify one of the URLs or document titles stored in the browser's history list (titles appear in the Go/View menu). As security and privacy concerns increased over time, this variant of the `go()` method has been reined in. It's best not to use the string parameter in your scripting.

Like most other history methods, your script finds it difficult to manage the history list or the current URL's spot in the queue. That fact makes it even more difficult for your script to intelligently determine how far to navigate in either direction or to which specific URL or title

matches it should jump. Use this method only for situations in which your Web pages are in strict control of the user's activity (or for designing scripts for yourself that automatically crawl around sites according to a fixed regimen). Once you give the user control over navigation, you have no guarantee that the history list will be what you expect, and any scripts you write that depend on a `history` object will likely break.

In practice, this method mostly performs a soft reload of the current window using the `0` parameter.

If you are developing a page for all scriptable browsers, be aware that Internet Explorer's `go()` method behaves a little differently than Netscape's. In IE4+, the matching string must be part of the URL and not part of the document title, as in Navigator. Additionally, the reloading of a page with `history.go(0)` often returns to the server to reload the page rather than reloading from the cache.

Listing 17-14 contains sample code that demonstrates how to navigate through the history list via the `go()` method. Fill in either the number or text field of the page in Listing 17-14 and then click the associated button. The script passes the appropriate kind of data to the `go()` method. Be sure to use negative numbers for visiting a page earlier in the history.

Listing 17-14: **Navigating to an Item in History**

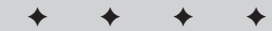
```
<html>
   <head>
      <title>history.go() Method</title>
      <script type="text/javascript">
      function doGoNum(form) {
         window.history.go(parseInt(form.histNum.value));
      }
      function doGoTxt(form) {
         window.history.go(form.histWord.value);
      }
      </script>
   </head>
   <body>
      <form>
         <b>Calling the history.go() method:</b>
         <hr />
         Enter a number (+/-):<input type="text" name="histNum" size="3"
         value="0" /> <input type="button" value="Go to Offset"
         onclick="doGoNum(this.form)" />
         <p>Enter a word in a title:<input type="text" name="histWord" />
            <input type="button" value="Go to Match"
            onclick="doGoTxt(this.form)" /></p>
      </form>
   </body>
</html>
```

Related Items: `history.back()`, `history.forward()`, `location.reload()` methods.

✦ ✦ ✦

CHAPTER 18

The Document and Body Objects

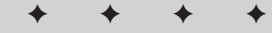


In This Chapter

Accessing arrays of objects contained by the `document` object

Writing new document content to a window or frame

Using the `body` element for IE window measurements



User interaction is a vital aspect of client-side JavaScript scripting, and most of the communication between script and user takes place by way of the `document` object and its components. Understanding the scope of the `document` object within each of the object models you support is key to implementing successful cross-browser applications.

Review the `document` object's place within the original object hierarchy. Figure 18-1 clearly shows that the `document` object is a pivotal point for a large percentage of objects. In the W3C DOM, the `document` object plays an even more important role as the container of all element objects delivered with the page: The `document` object is the root of the entire document tree.

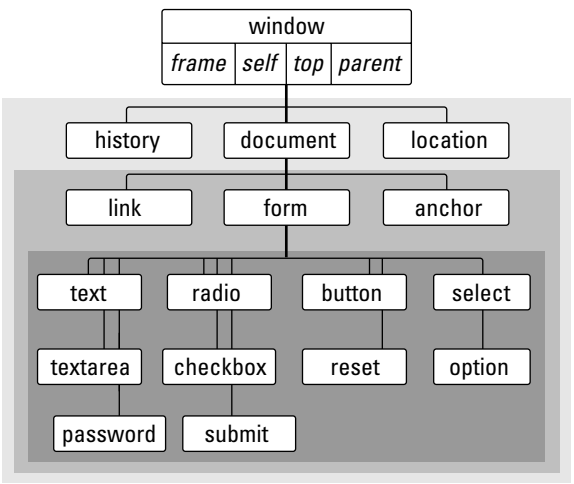


Figure 18-1: The basic document object model hierarchy.

In fact, the `document` object and all that it contains is so big that I have divided its discussion into many chapters, each focusing on related object groups. This chapter looks at the `document` object and `body` object (which have conceptual relationships), while each of the succeeding chapters in this part of the book details objects contained by the `document` object.

I must stress at the outset that many newcomers to JavaScript have the expectation that they can, on the fly, modify sections of a loaded page's content with ease: replace some text here, change a table cell there. However, understanding that these capabilities — an important part of what is called Dynamic HTML — are available only in more recent browsers, specifically IE4+/NN6+/Moz1+/Safari1+, is very important. Not only do these browsers expose every HTML element to script languages but they also automatically reflow the page when the size of content changes under script control. Pages on all previous browsers are limited to a small set of modifiable objects, such as images and form elements.

If your application requires compatibility with all scriptable browsers, you will be limited to changing only a handful of other invisible properties after the page loads. If these compatible pages need to modify their contents based on user input or timed updates, consider designing your pages so that scripts write the contents; then let the scripts rewrite the entire page with your new settings.

document Object

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Properties	*Methods*	*Event Handlers*
activeElement	attachEvent()†	onactivate†
alinkColor	captureEvents()	onbeforecut†
all†	clear()	onbeforedeactivate†
anchors[]	clearAttributes()†	onbeforeeditfocus†
applets[]	close()	onbeforepaste†
attributes†	createAttribute()	onclick†
bgColor	createComment()	oncontextmenu†
body	createDocumentFragment()	oncontrolselect†
charset	createElement()	oncut†
characterSet	createEvent()	ondblclick†
childNodes†	createEventObject()	ondrag†
compatMode	createRange()	ondragend†
cookie	createStyleSheet()	ondragenter†
defaultCharset	createTextNode()	ondragleave†
defaultView	createTreeWalker()	ondragover†
designMode	detachEvent()†	ondragstart†
doctype	elementFromPoint()	ondrop†
documentElement	execCommand()	onhelp†
domain	focus()†	onkeydown†
embeds[]	getElementById()	onkeypress†
expando	getElementsByName()	onkeyup†
fgColor	getElementsByTagName()†	onmousedown†
fileCreatedDate	getSelection()	onmousemove†
fileModifiedDate	handleEvent()	onmouseout†

Properties	*Methods*	*Event Handlers*
fileSize	hasFocus()†	onmouseover†
firstChild†	mergeAttributes()†	onmouseup†
forms[]	open()	onpaste†
frames[]	queryCommandEnabled()	onpropertychange†
height	queryCommandIndterm()	onreadyStatechange†
ids[]	queryCommandState()	onresizeend†
images[]	queryCommandSupported()	onresizestart†
implementation	queryCommandText()	onselectionchange
lastChild†	queryCommandValue()	onstop
lastModified	recalc()	
layers[]	releaseCapture()†	
linkColor	releaseEvents()	
links[]	routeEvent()	
location	setActive()†	
media	write()	
mimeType	writeln()	
nameProp		
namespaces[]		
namespaceURI†		
nextSibling†		
nodeName†		
nodeType†		
ownerDocument†		
parentNode†		
parentWindow		
plugins[]		
previousSibling†		
protocol		
readyState†		
referrer		
scripts[]		
security		
selection		
styleSheets[]		
tags[]		
title		
uniqueID†		
URL		
URLUnencoded		
vlinkColor		
width		

†See Chapter 15.

Syntax

Accessing `document` object properties or methods:

```
[window.]document.property | method([parameters])
```

About this object

A `document` object encompasses the totality of what exists inside the content region of a browser window or window frame (excluding toolbars, status lines, and so on). The document is a combination of the content and interface elements that make the Web page worth visiting. In more recent browsers, which treat HTML elements as nodes of a hierarchical tree, the `document` object is the root node — that from which all other nodes grow.

Because the `document` object isn't explicitly represented in an HTML document by tags or any other notation, the original designers of JavaScript and object models decided to make the `document` object the portal to many settings that were represented in HTML as belonging to the `body` element. That element's tag contains attributes for document-wide attributes, such as background color (`bgcolor`) and link colors in various states (`alink`, `link`, and `vlink`). The `body` element also served as an HTML container for forms, links, and anchors. The `document` object, therefore, assumed a majority of the role of the `body` element. But even then, the `document` object became the most convenient place to bind some properties that extend beyond the `body` element, such as the `title` element and the URL of the link that referred the user to the page. When viewed within the context of the HTML source code, the original `document` object is somewhat schizophrenic. Even so, the `document` object has worked well as the basis for references to original object model objects, such as forms, images, and applets.

This, of course, was before every HTML element, including the `body` element, was exposed as an object via modern object models. Amazingly, even with the IE4+ object model and W3C DOM — both of which treat the `body` element as an object separate from the `document` object — script compatibility with the original object model is quite easily accomplished. The `document` object has assumed a new schizophrenia, splitting its personality between the original object model and the one that places the `document` object at the root of the hierarchy, quite separate from the `body` element object it contains. The object knows which "face" to put on based on the rest of the script syntax that follows it. This means that quite often there are multiple ways to achieve the same reference. For example, you can use the following statement in all scriptable browsers to get the number of form objects in a document:

```
document.forms.length
```

In IE4+, you can also use

```
document.tags["FORM"].length
```

And in the W3C DOM as implemented in IE5+ and NN6+/Moz1+/Safari1+, you can use

```
document.getElementsByTagName("FORM").length
```

The more modern versions provide generic ways of accessing elements (the `tags` array in IE4+ and the `getElementsByTagName()` method in the W3C DOM) to meet the requirements of object models that expose every HTML (and XML) element as an object.

Promoting the body element to the ranks of exposed objects presented its own challenges to the new object model designers. The body element is the true "owner" of some properties that the original document object had to take on by default. Most properties that had belonged to the original document object were renamed in their transfer to the body element. For example, the original document.alinkColor property is the body.aLink property in the new model. But the bgColor property has not been renamed. For the sake of code compatibility, the current versions of browsers recognize both properties, even though the W3C DOM (in an effort to push the development world ahead) has removed the old versions as properties of what it conceives as the document object.

As confusing as all of this may sound on the surface, understanding when to refer to the original document object and when to use the new syntax doesn't take long. It all depends on what you hang off the right edge of the reference. Original properties and methods are recognized as using the original document object; new properties and methods summon the powers of the new document object. It's all quite automatic. Thankfully.

Properties

activeElement

Value: Object reference. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

In IE4+, a script can examine the document.activeElement property to see which element currently has focus. The value returned is an element object reference. You can use any of the properties and methods listed in Chapter 15 to find out more about the object. Be aware that not all elements in all operating systems receive focus. For example, buttons in IE4 for the Macintosh do not receive focus.

Although the element used to generate a mouse or keyboard event will most likely have focus (except for MacIE4 buttons), don't rely on the activeElement property to find out which element generated an event. The IE event.srcElement property is far more reliable.

Example

Use The Evaluator (Chapter 13) with IE4+ to experiment with the activeElement property. Type the following statement into the top text box:

```
document.activeElement.value
```

After you press the Enter key, the Results box shows the value of the text box you just typed into (the very same expression you just typed). But if you then click the Evaluate button, you will see the value property of that button object appear in the Results box.

Related Items: event.srcElement property.

document.alinkColor

```
alinkColor
bgColor
fgColor
linkColor
vlinkColor
```

Value: Hexadecimal triplet or color name string. Mostly Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

These five properties are the script equivalent of the `<body>` tag attributes of the same name (although the property names are case-sensitive). All five settings can be read via scripting, but the ability to change some or all of these properties varies widely with browser and client platform. Table 18-1 shows a summary of which browsers and platforms can set which of the color properties; Mozilla and Safari browsers support all of the color properties.

Table 18-1: Setting Document Colors on the Fly (Browser Versions)

	Navigator			*Internet Explorer*		
Color Property	***Windows***	***Mac***	***UNIX***	***Windows***	***Mac***	***UNIX***
bgColor	All	4+	4+	All	All	4+
All others	6+	6+	6+	All	All	4+

Values for all color properties can be either the common HTML hexadecimal triplet value (for example, `"#00FF00"`) or any of the Netscape color names. Internet Explorer recognizes these plain language color names, as well. But also be aware that some colors work only when the user has the monitor set to 16- or 24-bit color settings.

If you are scripting exclusively for IE4+ and NN6+, you should use the `document.body` object to access these properties.

Example

I select some color values at random to plug into three settings of the ugly colors group for Listing 18-1. The smaller window displays a dummy button so that you can see how its display contrasts with color settings. Notice that the script sets the colors of the smaller window by rewriting the entire window's HTML code. After changing colors, the script displays the color values in the original window's textarea. Even though some colors are set with the color constant values, properties come back in the hexadecimal triplet values. You can experiment to your heart's content by changing color values in the listing. Every time you change the values in the script, save the HTML file and reload it in the browser.

Listing 18-1: **Tweaking the Color of Page Elements**

```
<html>
   <head>
      <title>Color Me</title>
      <script type="text/javascript">
      // may be blocked at load time by browser popup blockers
      var newWindow = window.open("","","height=150,width=300");
```

```
   function defaultColors() {
      return "bgcolor='#c0c0c0' vlink='#551a8b' link='#0000ff'";
   }

   function uglyColors() {
      return "bgcolor='yellow' vlink='pink' link='lawngreen'";
   }

   function showColorValues() {
      var result = "";
      result += "bgColor: " + newWindow.document.bgColor + "\n";
      result += "vlinkColor: " + newWindow.document.vlinkColor + "\n";
      result += "linkColor: " + newWindow.document.linkColor + "\n";
      document.forms[0].results.value = result;
   }

   // dynamically writes contents of another window
   function drawPage(colorStyle) {
      // work around popup blockers
      if (!newWindow || newWindow.closed) {
         newWindow = window.open("","","height=150,width=300");
      }
      var thePage = "";
      thePage += "<html><head><title>Color Sampler<\/title><\/head><body ";
      if (colorStyle == "default") {
         thePage += defaultColors();
      } else {
         thePage += uglyColors();
      }
      thePage += ">Just so you can see the variety of items and color, <a ";
      thePage += "href='http://www.nowhere.com'>here\'s a link<\/a>, and <a
      href='http://home.netscape.com'> here is another link <\/a> you can
      use on-line to visit and see how its color differs from the standard
      link.";
      thePage += "<form>";
      thePage += "<input type='button' name='sample' value='Just a
      Button'>";
      thePage += "<\/form><\/body><\/html>";
      newWindow.document.write(thePage);
      newWindow.document.close();
      showColorValues();
   }

   // the following works properly only in Windows Navigator
   function setColors(colorStyle) {
      if (colorStyle == "default") {
         document.bgColor = "#c0c0c0";
      } else {
         document.bgColor = "yellow";
      }
   }

   var newWindow = window.open("","","height=150,width=300");
   </script>
</head>
<body>
   Try the two color schemes on the document in the small window.
   <form>
```

Continued

Listing 18-1 *(continued)*

```
        <input type="button" name="default" value='Default Colors'
        onclick="drawPage('default')" /> <input type="button" name="weird"
        value="Ugly Colors" onclick="drawPage('ugly')" />
        <p><textarea name="results" rows="3" cols="20">
           </textarea></p>
        <hr />
        These buttons change the current document, but not correctly on all
        platforms
        <p><input type="button" name="default" value='Default Colors'
           onclick="setColors('default')" /> <input type="button" name="weird"
           value="Ugly Colors" onclick="setColors('ugly')" /></p>
     </form>
  </body>
</html>
```

To satisfy the curiosity of those who want to change the color of a loaded document on the fly, the preceding example includes a pair of buttons that set the color properties of the current document. If you're running browsers and versions capable of this power (see Table 18-1), everything will look fine; but in other platforms or earlier versions, you may lose the buttons and other document content behind the color. You can still click and activate these items, but the color obscures them. Unless you know for sure that users of your Web page use only browsers and clients empowered for background color changes, do not change colors by setting properties of an existing document.

Related Items: `body.aLink`, `body.bgColor`, `body.link`, `body.text`, `body.vLink` properties.

anchors[]

Value: Array of anchor objects. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Anchor objects (described in Chapter 19) are points in an HTML document marked with `<a name="">` tags. Anchor objects are referenced in URLs by a hash value between the page URL and anchor name. Like other object properties that contain a list of nested objects, the `document.anchors` property (notice the plural) delivers an indexed array of anchors in a document. Use the array references to pinpoint a specific anchor for retrieving any anchor property.

Anchor arrays begin their index counts with 0: The first anchor in a document, then, has the reference `document.anchors[0]`. And, as is true with any built-in array object, you can find out how many entries the array has by checking the `length` property. For example

```
var anchorCount = document.anchors.length;
```

The `document.anchors` property is read-only. To script navigation to a particular anchor, assign a value to the `window.location` or `window.location.hash` object, as described in Chapter 17's `location` object discussion.

Example

In Listing 18-2, I append an extra script to Listing 17-1 to demonstrate how to extract the number of anchors in the document. The document dynamically writes the number of anchors found in the document. You will not likely ever need to reveal such information to users of your page, and the `document.anchors` property is not one that you will call frequently. The object model defines it automatically as a document property while defining actual anchor objects.

Listing 18-2: **Using Anchors to Navigate Through a Page**

```
<html>
   <head>
      <title>document.anchors Property</title>
      <script type="text/javascript">
      function goNextAnchor(where) {
         window.location.hash = where;
      }
      </script>
   </head>
   <body>
      <h1><a id="start" name="start">Top</a></h1>
      <form>
         <input type="button" name="next" value="NEXT"
         onclick="goNextAnchor('sec1')" />
      </form>
      <hr />
      <h1><a id="sec1" name="sec1">Section 1</a></h1>
      <form>
         <input type="button" name="next" value="NEXT"
         onclick="goNextAnchor('sec2')" />
      </form>
      <hr />
      <h1><a id="sec2" name="sec2">Section 2</a></h1>
      <form>
         <input type="button" name="next" value="NEXT"
         onclick="goNextAnchor('sec3')" />
      </form>
      <hr />
      <h1><a id="sec3" name="sec3">Section 3</a></h1>
      <form>
         <input type="button" name="next" value="BACK TO TOP"
         onclick="goNextAnchor('start')" />
      </form>
      <hr />
      <p>
         <script type="text/javascript">
         document.write("<i>There are " + document.anchors.length +
            " anchors defined for this document<\/i>")
         </script>
      </p>
   </body>
</html>
```

Related Items: `anchor`, `location` objects; `document.links` property.

applets[]

Value: Array of applet objects. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `applets` property refers to Java applets defined in a document by the `<applet>` tag. An applet is not officially an object in the document until the applet loads completely.

Most of the work you do with Java applets from JavaScript takes place via the methods and variables defined inside the applet. Although you can reference an applet according to its indexed array position within the `applets` array, you will more likely use the applet object's name in the reference to avoid any confusion.

Example

The `document.applets` property is defined automatically as the browser builds the object model for a document that contains applet objects. You will rarely access this property, except to determine how many applet objects a document has.

Related Items: `applet` object.

bgColor

(See `alinkColor`)

body

Value: `body` element object. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `document.body` property is a shortcut reference to the `body` element object in modern object models. As you can see in the discussion of the `body` element object later in this chapter, that object has many key properties that govern the look of the entire page. Because the `document` object is the root of all references within any window or frame, the `document.body` property is easier to use to get to the `body` properties, rather than longer references normally used to access HTML element objects in both the IE4+ and W3C object models.

Example

Use The Evaluator (Chapter 13) to examine properties of the `body` element object. First, to prove that the `document.body` is the same as the element object that comes back from longer references, enter the following statement into the top text box with either IE5+, NN6+, or some other W3C browser:

```
document.body == document.getElementsByTagName("body")[0]
```

Next, check out the `body` object's property listings later in this chapter and enter the listings into the top text box to review their results. For example:

```
document.body.bgColor
document.body.tagName
```

Related Items: `body element` object.

charset

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `charset` property reveals the character set used by the browser to render the current document (the NN6+/Moz1+ version of this property is called `characterSet`). You can find possible values for this property at

```
ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets
```

Each browser and operating system has its own default character set. Values may also be set via a <meta> tag.

Example

Use The Evaluator (Chapter 13) to experiment with the charset property. To see the default setting applied to the page, enter the following statement into the top text box:

```
document.charset
```

If you are running IE5+ for Windows and you enter the following statement, the browser will apply a different character set to the page:

```
document.charset = "iso-8859-2"
```

If your version of Windows does not have that character set installed in the system, the browser may ask permission to download and install the character set.

Related Items: characterSet, defaultCharset properties.

characterSet

Value: String. Read/Write
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The characterSet property reveals the character set used by the browser to render the current document (the IE4+ version of this property is called charset). You can find possible values for this property at

```
http://www.iana.org/assignments/character-sets
```

Each browser and operating system has its own default character set. Values may also be set via a <meta> tag.

Example

Use The Evaluator (Chapter 13) to experiment with the characterSet property in NN6+/Moz1+. To see the default setting applied to the page, enter the following statement into the top text box:

```
document.charset
```

Related Items: charset property.

compatMode

Value: String. Read-Only
Compatibility: WinIE6+, MacIE6+, NN7+, Moz1+, Safari-

The compatMode property reveals the compatibility mode for the document, as determined by the DOCTYPE element's content. The value for this property can be one of the following string constants: BackCompat or CSS1Compat. The default setting for the compatMode property is BackCompat.

Example

You may find it useful to check the compatibility mode of a document in order to carry out processing specific to one of the modes. Following is an example of how you might branch to carry out processing for backward-compatible documents:

```
if (document.compatMode == "BackCompat") {
   // perform backward compatible processing
}
```

Related Items: *Standards Compatibility Modes* (Chapter 13).

cookie

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The cookie mechanism in a Web browser lets you store small pieces of information on the client computer in a reasonably secure manner. In other words, when you need some tidbit of information to persist at the client level while either loading diverse HTML documents or moving from one session to another, the cookie mechanism saves the day. The cookie is commonly used as a means to store the username and password you enter into a password-protected Web site. The first time you enter this information into a CGI-governed form, the CGI program has Navigator write the information back to a cookie on your hard disk (usually after encrypting the password). Rather than bothering you to enter the username and password the next time you access the site, the server searches the cookie data stored for that particular server and extracts the username and password for automatic validation processing behind the scenes.

Other applications of the cookie include storing user preferences and information about the user's previous visit to the site. Preferences may include font styles or sizes and whether the user prefers viewing content inside a frameset or not. As shown in Chapter 54 on the CD-ROM, a time stamp of the previous visit can allow a coded HTML page to display highlighted images next to content that has changed since the user's last visit, even if you have updated the page several times in the interim. Rather than hard-wiring "New" flags for *your* last visit, the scripts highlight what's new for the visitor.

The cookie file

Allowing some foreign CGI program to read from and write to your hard disk may give you pause, but browser cookie mechanisms don't just open up your drive's directory for the world to see (or corrupt). Instead, the cookie mechanism provides access to just one special text file (Navigator/Mozilla/Safari) or type of text file (Internet Explorer) located in a platform-specific spot on your drive.

In Mozilla-based browsers, for example, the cookie file is named `cookies.txt` and is located in a directory (whose name ends in `.slt`) within the browser's profile area. In Windows, that location is `C:\\Windows\Application Data\Mozilla\Profiles\[profilename]\`; in Mac OSX, the location is `[user]/Library/Mozilla/Profiles/[profilename]/`. Internet Explorer for Windows uses a different filing system: all cookies for each domain are saved in a domain-specific file inside the `C:\\Windows\Temporary Internet Files\` directory. Filenames begin with `Cookie:` and include the username and domain of the server that wrote the cookie. Safari cookies are recorded in an XML file named `Cookies.plist` within the `[user]/Library/Cookies/` directory.

A cookie file is a text file. If curiosity drives you to open a cookie file, I recommend you do so only with a copy saved in another directory or folder. Any alteration to the existing file can mess up whatever valuable cookies are stored there for sites you regularly visit. The data format for cookie files differs across browsers, in line with the different methodologies used for filing cookies. Inside the Netscape/Mozilla file (after a few comment lines warning you not to manually alter the file) are lines of tab-delimited text. Each return-delimited line contains one cookie's information. The cookie file is just like a text listing of a database. In each of the IE cookie files, the same data points are stored for a cookie as for Navigator, but the items are in a return-delimited list. The structure of these files is of no importance to scripting cookies, because all browsers utilize the same syntax for reading and writing cookies through the `document.cookie` property.

As you experiment with browser's cookies, you will be tempted to look into the cookie file after a script writes some data to the cookie. The cookie file usually will not contain the newly written data, because in most browsers cookies are transferred to disk only when the user quits the browser; conversely, the cookie file is read into the browser's memory when it is launched. While you read, write, and delete cookies during a browser session, all activity is performed in memory (to speed up the process) to be saved later.

A cookie record

Among the "fields" of each cookie record are the following (not necessarily in this order):

- ✦ Domain of the server that created the cookie
- ✦ Information on whether you need a secure HTTP connection to access the cookie
- ✦ Pathname of URL(s) capable of accessing the cookie
- ✦ Expiration date of the cookie
- ✦ Name of the cookie entry
- ✦ String data associated with the cookie entry

Notice that cookies are domain-specific. In other words, if one domain creates a cookie, another domain cannot access it through the browser's cookie mechanism behind your back. That reason is why it's generally safe to store what I call *throwaway passwords* (the username/password pairs required to access some free registration-required sites) in cookies. Moreover, sites that store passwords in a cookie usually do so as encrypted strings, making it more difficult for someone to hijack the cookie file from your unattended PC and figure out what your personal password scheme may be.

Cookies also have expiration dates. Because some browsers may allow no more than a fixed number of cookies (300 in NN), the cookie file can get pretty full over the years. Therefore, if a cookie needs to persist past the current browser session, it should have an expiration date established by the cookie writer. Browsers automatically clean out any expired cookies.

Not all cookies have to last beyond the current session, however. In fact, a scenario in which you use cookies temporarily while working your way through a Web site is quite typical. Many shopping sites employ one or more temporary cookie records to behave as the shopping cart for recording items you intend to purchase. These items are copied to the order form at checkout time. But after you submit the order form to the server, that client-side data has no particular value. As it turns out, if your script does not specify an expiration date, the browser keeps the cookie fresh in memory without writing it to the cookie file. When you quit the browser, that cookie data disappears as expected.

JavaScript access

Scripted access of cookies from JavaScript is limited to setting the cookie (with a number of optional parameters) and getting the cookie data (but with none of the parameters).

The original object model defines cookies as properties of documents, but this description is somewhat misleading. If you use the default path to set a cookie (that is, the current directory of the document whose script sets the cookie in the first place), all documents in that same server directory have read and write access to the cookie. A benefit of this arrangement is that if you have a scripted application that contains multiple documents, all documents served from the same directory can share the cookie data. NN and IE, however, impose a limit of 20 named cookie entries (that is, one name/value pair) for any domain. If your cookie requirements are extensive, you need to fashion ways of concatenating cookie data (I do this in the Decision Helper application in Chapter 55 on the CD-ROM).

Saving cookies

To write cookie data to the cookie file, you use a simple JavaScript assignment operator with the `document.cookie` property. But the formatting of the data is crucial to achieving success. Here is the syntax for assigning a value to a cookie (optional items are in brackets; placeholders for data you supply are in italics):

```
document.cookie = "cookieName=cookieData
            [; expires=timeInGMTString]
            [; path=pathName]
            [; domain=domainName]
            [; secure]"
```

Let's examine each of the properties individually.

Name/Data

Each cookie must have a name and a string value (even if that value is an empty string). Such name/value pairs are fairly common in HTML, but they look odd in an assignment statement. For example, if you want to save the string "Fred" to a cookie named "userName," the JavaScript statement is

```
document.cookie = "userName=Fred";
```

If the browser sees no existing cookie in the current domain with this name, it automatically creates the cookie entry for you; if the named cookie already exists, the browser replaces the old data with the new data. Retrieving the `document.cookie` property at this point yields the following string:

```
userName=Fred
```

You can omit all the other cookie-setting properties, in which case the browser uses default values, as explained in a following section. For temporary cookies (those that don't have to persist beyond the current browser session), the name/value pair is usually all you need.

The entire name/value pair must be a single string with no semicolons, commas, or character spaces. To take care of spaces between words, preprocess the value with the JavaScript `escape()` function, which URL-encodes the spaces as `%20` (and then be sure to `unescape()` the value to restore the human-readable spaces when you retrieve the cookie later).

You cannot save a JavaScript array or object to a cookie. But with the help of the `Array.join()` method, you can convert an array to a string; use `String.split()` to re-create the array after reading the cookie at a later time. These two methods are available in NN3+/Moz1+, IE4+, and Safari1+.

Expires

Expiration dates, when supplied, must be passed as Greenwich Mean Time (GMT) strings (see Chapter 29 about time data). To calculate an expiration date based on today's date, use the JavaScript `Date` object as follows:

```
var exp = new Date();
var oneYearFromNow = exp.getTime() + (365 * 24 * 60 * 60 * 1000);
exp.setTime(oneYearFromNow);
```

Then convert the date to the accepted GMT string format:

```
document.cookie = "userName=Fred; expires=" + exp.toGMTString();
```

In the cookie file, the expiration date and time is stored as a numeric value (seconds) but, to set it, you need to supply the time in GMT format. You can delete a cookie before it expires by setting the named cookie's expiration date to a time and date earlier than the current time and date. The safest expiration parameter is

```
expires=Thu, 01-Jan-70 00:00:01 GMT
```

Omitting the expiration date signals the browser that this cookie is temporary. The browser never writes it to the cookie file and forgets it the next time you quit the browser.

Path

For client-side cookies, the default path setting (the current directory) is usually the best choice. You can, of course, create a duplicate copy of a cookie with a separate path (and domain) so that the same data is available to a document located in another area of your site (or the Web).

Domain

To help synchronize cookie data with a particular document (or group of documents), the browser matches the domain of the current document with the domain values of cookie entries in the cookie file. Therefore, if you were to display a list of all cookie data contained in a `document.cookie` property, you would get back all the name/value cookie pairs from the cookie file whose domain parameter matches that of the current document.

Unless you expect the document to be replicated in another server within your domain, you can usually omit the `domain` parameter when saving a cookie. Default behavior automatically supplies the domain of the current document to the cookie file entry. Be aware that a domain setting must have at least two periods, such as

```
.mcom.com
.hotwired.com
```

Or, you can write an entire URL to the domain, including the `http://` protocol.

SECURE

If you omit the `SECURE` parameter when saving a cookie, you imply that the cookie data is accessible to any document or CGI program from your site that meets the other domain- and path-matching properties. For client-side scripting of cookies, you should omit this parameter when saving a cookie.

Retrieving cookie data

Cookie data retrieved via JavaScript is contained in one string, including the whole name-data pair. Even though the cookie file stores other parameters for each cookie, you can retrieve only the name-data pairs via JavaScript. Moreover, when two or more (up to a maximum of 20) cookies meet the current domain criteria, these cookies are also lumped into that string, delimited by a semicolon and space. For example, a `document.cookie` string may look like this:

```
userName=Fred; password=NikL2sPacU
```

In other words, you cannot treat named cookies as objects. Instead, you must parse the entire cookie string, extracting the data from the desired name-data pair.

When you know that you're dealing with only one cookie (and that no more will ever be added to the domain), you can customize the extraction based on known data, such as the cookie name. For example, with a cookie name that is seven characters long, you can extract the data with a statement such as this:

```
var data = unescape(document.cookie.substring(7,document.cookie.length));
```

The first parameter of the `substring()` method includes the equal sign to separate the name from the data.

A better approach is to create a general-purpose function that can work with single- or multiple-entry cookies. Here is one I use in some of my pages:

```
function getCookieData(labelName) {
   var labelLen = labelName.length;
   // read cookie property only once for speed
   var cookieData = document.cookie;
   var cLen = cookieData.length;
   var i = 0;
   var cEnd;
   while (i < cLen) {
      var j = i + labelLen;
      if (cookieData.substring(i,j) == labelName) {
         cEnd = cookieData.indexOf(";",j);
         if (cEnd == -1) {
            cEnd = cookieData.length;
         }
         return unescape(cookieData.substring(j+1, cEnd));
      }
      i++;
   }
   return "";
}
```

Calls to this function pass the label name of the desired cookie as a parameter. The function parses the entire cookie string, chipping away any mismatched entries (through the semicolons) until it finds the cookie name.

If all of this cookie code still makes your head hurt, you can turn to a set of functions devised by experienced JavaScripter and Web site designer Bill Dortch of hIdaho Design. His cookie functions provide generic access to cookies that you can use in all of your cookie-related pages. Listing 18-3 shows Bill's cookie functions, which include a variety of safety nets for date calculation bugs that appeared in some older versions of Netscape Navigator. Don't be put off by the length of the listing: Most of the lines are comments.

Listing 18-3: **Bill Dortch's Cookie Functions**

```
<html>
   <head>
      <title>Cookie Functions</title>
   </head>
   <body>
      <script type="text/javascript">
      //
      //  Cookie Functions -- "Night of the Living Cookie" Version (25-Jul-96)
      //
      //  Written by:  Bill Dortch, hIdaho Design
      //  The following functions are released to the public domain.
      //
      //  This version takes a more aggressive approach to deleting
      //  cookies.  Previous versions set the expiration date to one
      //  millisecond prior to the current time; however, this method
      //  did not work in Netscape 2.02 (though it does in earlier and
      //  later versions), resulting in "zombie" cookies that would not
      //  die.  DeleteCookie now sets the expiration date to the earliest
      //  usable date (one second into 1970), and sets the cookie's value
      //  to null for good measure.
      //
      //  Also, this version adds optional path and domain parameters to
      //  the DeleteCookie function.  If you specify a path and/or domain
      //  when creating (setting) a cookie**, you must specify the same
      //  path/domain when deleting it, or deletion will not occur.
      //
      //  The FixCookieDate function must now be called explicitly to
      //  correct for the 2.x Mac date bug.  This function should be
      //  called *once* after a Date object is created and before it
      //  is passed (as an expiration date) to SetCookie.  Because the
      //  Mac date bug affects all dates, not just those passed to
      //  SetCookie, you might want to make it a habit to call
      //  FixCookieDate any time you create a new Date object:
      //
      //    var theDate = new Date();
      //    FixCookieDate (theDate);
      //
      //  Calling FixCookieDate has no effect on platforms other than
      //  the Mac, so there is no need to determine the user's platform
      //  prior to calling it.
      //
      //  This version also incorporates several minor coding improvements.
      //
      //  **Note that it is possible to set multiple cookies with the same
      //  name but different (nested) paths.  For example:
      //
      //    SetCookie ("color","red",null,"/outer");
      //    SetCookie ("color","blue",null,"/outer/inner");
      //
      //  However, GetCookie cannot distinguish between these and will return
      //  the first cookie that matches a given name.  It is therefore
      //  recommended that you *not* use the same name for cookies with
      //  different paths.  (Bear in mind that there is *always* a path
```

Continued

Listing 18-3 *(continued)*

```
//  associated with a cookie; if you don't explicitly specify one,
//  the path of the setting document is used.)
//
//  Revision History:
//
//    "Toss Your Cookies" Version (22-Mar-96)
//      - Added FixCookieDate() function to correct for Mac date bug
//
//    "Second Helping" Version (21-Jan-96)
//      - Added path, domain and secure parameters to SetCookie
//      - Replaced home-rolled encode/decode functions with Netscape's
//        new (then) escape and unescape functions
//
//    "Free Cookies" Version (December 95)
//
//
//  For information on the significance of cookie parameters,
//  and on cookies in general, please refer to the official cookie
//  spec, at:
//
//      http://www.netscape.com/newsref/std/cookie_spec.html
//
//****************************************************************
//
// "Internal" function to return the decoded value of a cookie
//
function getCookieVal (offset) {
   var endstr = document.cookie.indexOf (";", offset);
   if (endstr == -1) {
      endstr = document.cookie.length;
   }
   return unescape(document.cookie.substring(offset, endstr));
}

//
//  Function to correct for 2.x Mac date bug.  Call this function to
//  fix a date object prior to passing it to SetCookie.
//  IMPORTANT:  This function should only be called *once* for
//  any given date object!  See example at the end of this document.
//
function FixCookieDate (date) {
   var base = new Date(0);
   var skew = base.getTime(); // dawn of (Unix) time - should be 0
   if (skew > 0) {  // Except on the Mac - ahead of its time
      date.setTime (date.getTime() - skew);
   }
}

//
//  Function to return the value of the cookie specified by "name".
//    name - String object containing the cookie name.
//    returns - String object containing the cookie value, or null if
//      the cookie does not exist.
//
```

```
function GetCookie (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 getCookieVal (j);
      }
      i = document.cookie.indexOf(" ", i) + 1;
      if (i == 0) {
         break;
      }
   }
   return null;
}

//
//  Function to create or update a cookie.
//    name - String 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 place holder.  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 SetCookie (name,value,expires,path,domain,secure) {
   document.cookie = name + "=" + escape (value) +
      ((expires) ? "; expires=" + expires.toGMTString() : "") +
      ((path) ? "; path=" + path : "") +
      ((domain) ? "; domain=" + domain : "") +
      ((secure) ? "; secure" : "");
}
```

Continued

Listing 18-3 *(continued)*

```
      //  Function to delete a cookie. (Sets expiration date to start of epoch)
      //    name -   String object containing the cookie name
      //    path -   String object containing the path of the cookie to delete.
      //             This MUST be the same as the path used to create the
      //             cookie, or null/omitted if
      //             no path was specified when creating the cookie.
      //    domain - String object containing the domain of the cookie to
      //             delete.  This MUST be the same as the domain used to
      //             create the cookie, or null/omitted if no domain was
      //             specified when creating the cookie.
      //
      function DeleteCookie (name,path,domain) {
         if (GetCookie(name)) {
            document.cookie = name + "=" +
               ((path) ? "; path=" + path : "") +
               ((domain) ? "; domain=" + domain : "") +
               "; expires=Thu, 01-Jan-70 00:00:01 GMT";
         }
      }

      //
      //  Examples
      //
      var expdate = new Date ();
      FixCookieDate (expdate); // Correct for Mac date bug (call only once)
      expdate.setTime (expdate.getTime() + (24 * 60 * 60 * 1000)); // 24 hrs
      SetCookie ("ccpath", "http://www.hidaho.com/colorcenter/", expdate);
      SetCookie ("ccname", "hIdaho Design ColorCenter", expdate);
      SetCookie ("tempvar", "This is a temporary cookie.");
      SetCookie ("ubiquitous", "This cookie will work anywhere in this
         domain",null,"/");
      SetCookie ("paranoid", "This cookie requires secure
         communications",expdate,"/",null,true);
      SetCookie ("goner", "This cookie must die!");
      document.write (document.cookie + "<br>");
      DeleteCookie ("goner");
      document.write (document.cookie + "<br>");
      document.write ("ccpath = " + GetCookie("ccpath") + "<br>");
      document.write ("ccname = " + GetCookie("ccname") + "<br>");
      document.write ("tempvar = " + GetCookie("tempvar") + "<br>");
      </script>
   </body>
</html>
```

Extra batches

You may design a site that needs more than 20 cookies for a given domain. For example, in a shopping site, you never know how many items a customer may load into the shopping cart cookie.

Because each named cookie stores plain text, you can create your own text-based data structures to accommodate multiple pieces of information per cookie. (But also watch out for a

practical limit of 2,000 characters per name/value pair within the 4,000 character maximum for any domain's combined cookies.) The trick is determining a delimiter character that won't be used by any of the data in the cookie. In Decision Helper (in Chapter 55 on the CD-ROM), for example, I use a period to separate multiple integers stored in a cookie.

With the delimiter character established, you must then write functions that concatenate these "subcookies" into single cookie strings and extract them on the other side. It's a bit more work, but well worth the effort to have the power of persistent data on the client.

Example

Experiment with the last group of statements in Listing 18-3 to create, retrieve, and delete cookies. You can also experiment with The Evaluator by assigning a name/value pair string to `document.cookie`, and then examining the value of the `cookie` property.

Related Items: `String` object methods (Chapter 27).

defaultCharset

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `defaultCharset` property reveals the character set used by the browser to render the current document. You can find possible values for this property at

```
http://www.iana.org/assignments/character-sets
```

Each browser and operating system has its own default character set. Values may also be set via a `<meta>` tag. The difference between the `defaultCharset` and `charset` properties is not clear, especially because both are read/write (although modifying the `defaultCharset` property has no visual effect on the page). However, if your scripts temporarily modify the `charset` property, you can use the `defaultCharset` property to return to the original character set:

```
document.charset = document.defaultCharset;
```

Example

Use The Evaluator (Chapter 13) to experiment with the `defaultCharset` property. To see the default setting applied to the page, enter the following statement into the top text box:

```
document.defaultCharset
```

Related Items: `charset`, `characterSet` properties.

defaultView

Value: `window` or `frame` object reference. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `defaultView` property returns a reference to the object serving as the "viewer" for the document. The viewer is responsible for rendering the document, and in NN6+ the object returned in the `defaultView` property is the `window` or `frame` object that contains the document. This W3C DOM Level 2 property provides access to computed CSS values being applied to any HTML element (via the `document.defaultView.getComputedStyle()` method, or more simply, the `window.getComputedStyle()` method).

Related Items: `window` and `frame` properties; `window.getComputedStyle()` method.

designMode

Value: String. Read/Write
Compatibility: WinIE5+, MacIE-, NN7.1, Moz1.4+, Safari-

The designMode property is applicable only when WinIE5+ technology is being used as a component in another application. The property controls whether the browser module is being used for HTML editing. Modifying the property from within a typical HTML page in the IE5+ browser has no effect. But on the Mozilla side, the property can be used to turn an iframe element's document object into an HTML editable document. Visit http://www.mozilla.org/editor for current details and examples.

doctype

Value: DocumentType object reference. Read-Only
Compatibility: WinIE-, MacIE5+, NN6+, Moz1+, Safari-

The doctype property comes from the W3C Core DOM and returns a DocumentType object — a representation of the DTD information for the document. The DocumentType object (if one is explicitly defined in the source code) is the first child node of the root document node (and is thus a sibling to the HTML element).

Table 18-2 shows the typical DocumentType object property list and values for a generic HTML page. Future DOM specifications will allow these properties to be read/write.

Table 18-2: DocumentType Object in NN6+/Moz1+

Property	*Value*
entities	null
internalSubset	(empty)
name	html
notations	null
publicId	-//W3C//DTD XHTML 1.0 Transitional//EN
systemId	http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd

Related Items: Node object (Chapter 14).

documentElement

Value: HTML or XML element object reference. Read-Only
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The documentElement property returns a reference to the HTML (or XML) element object that contains all of the content of the current document. The naming of this property is a bit misleading, because the root document node is not an element, but its only child node is the HTML (or XML) element for the page. At best, you can think of this property as providing scripts with an "element face" to the document object and document node associated with the page currently loaded in the browser.

Example

Use The Evaluator (Chapter 13) to examine the behavior of the `documentElement` property. In IE5+/W3C, enter the following statement into the top text field:

```
document.documentElement.tagName
```

The result is `HTML`, as expected.

Related Items: `ownerDocument` property (Chapter 15).

domain

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

Security restrictions can get in the way of sites that have more than one server at their domain. Because some objects, especially the `location` object, prevent access to properties of other servers displayed in other frames, legitimate access to those properties are blocked. For example, it's not uncommon for popular sites to have their usual public access site on a server named something such as `www.popular.com`. If a page on that server includes a front end to a site search engine located at `search.popular.com`, visitors who use browsers with these security restrictions are denied access.

To guard against that eventuality, a script in documents from both servers can instruct the browser to think both servers are the same. In the preceding example, you would set the `document.domain` property in both documents to `popular.com`. Without specifically setting the property, the default value includes the server name as well, thus causing a mismatch between hostnames.

Before you start thinking that you can spoof your way into other servers, be aware that you can set the `document.domain` property only to servers with the same domain (following the "two-dot" rule) as the document doing the setting. Therefore, documents originating only from `xxx.popular.com` can set their `document.domain` properties to `popular.com` server.

Related Items: `window.open()` method; `window.location` object; security (Chapter 46 on the CD-ROM).

embeds[]

Value: Array of `embed` element objects. Read-Only
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

Although now supplanted by the `<object>` tag, the `<embed>` tag used to be the markup that loaded data requiring a plug-in application to play or display. The `document.embeds` property is an array of `embed` element objects within the document:

```
var count = document.embeds.length;
```

For controlling those plug-ins in Navigator and Mozilla-based browsers, you can use the LiveConnect technology, described in Chapter 44 on the CD-ROM.

Related Items: `embed` element object (Chapter 40 on the CD-ROM).

expando

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

Microsoft calls any custom property that is not a native property of the `document` object an *expando* property. By default, most objects in recent generations of browsers allow scripts to add new properties of objects as a way to temporarily store data without explicitly defining global variables. For example, if you want to maintain an independent counter of how often a function is invoked, you can create a custom property of the `document` object and use it as the storage facility:

```
document.counter = 0;
```

IE4+ lets you control whether the `document` object is capable of accepting `expando` properties. The default value of the `document.expando` property is `true`, thus allowing custom properties. But the potential downside to this permissiveness, especially during the page construction phase, is that a misspelled native property name is gladly accepted by the `document` object. You may not be aware of why the title bar of the browser window doesn't change when you assign a new string to the `document.Title` property (which, in the case-sensitive world of JavaScript, is distinct from the native `document.title` property).

Example

Use The Evaluator (Chapter 13) to experiment with the `document.expando` property in IE4+. Begin by proving that the `document` object can normally accept custom properties. Type the following statement into the top text field:

```
document.spooky = "Boo!"
```

This property is now set and stays that way until the page is either reloaded or unloaded.

Now freeze the `document` object's properties with the following statement:

```
document.expando = false
```

If you try to add a new property, such as the following, you receive an error:

```
document.happy = "tra la"
```

Interestingly, even though `document.expando` is turned off, the first custom property is still accessible and modifiable.

Related Items: `prototype` property of custom objects (Chapter 33).

fgColor

(See `alinkColor`)

fileCreatedDate
fileModifiedDate
fileSize

Value: String, Integer (fileSize). Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

These three IE-specific properties return information about the file that holds the current document. The first two properties (not implemented in MacIE) reveal the dates on which the current document's file was created and modified. For an unmodified file, its creation and modified dates are the same. The `fileSize` property reveals the number of bytes of the file.

Date values returned for the first two properties are formatted differently between IE4 and IE5+. The former provides a full readout of the day and date; the latter in a format similar to mm/dd/yyyy. Note, however, that the values contain only the date and not the time. In any case, you can use the values as the parameter to a `new Date()` constructor function. You can then use date calculations for such information as the number of days between the current day and the most recent modification.

Not all servers may provide the proper date or size information about a file or in a format that IE can interpret. Test your implementation on the deployment server to ensure compatibility.

Also, be aware that these properties can be read only for a file that is loaded in the browser. JavaScript by itself cannot get this information about files that are on the server but not loaded in the browser.

IE5.5+ exposes a property called `fileUpdatedDate`, but the property does not return any data. This property may be a phantom property left over from a prerelease version.

Example

Listing 18-4 dynamically generates several pieces of content relating to the creation and modification dates of the file, as well as its size. More importantly, the listing demonstrates how to turn a value returned by the file date properties into a genuine date object that can be used for date calculations. In the case of Listing 18-4, the calculation is the number of full days between the creation date and the day someone views the file. Notice that the dynamically generated content is added very simply via the `innerText` properties of carefully located `span` elements in the body content.

Listing 18-4: **Displaying File Information for a Web Page**

```
<html>
   <head>
      <title>fileCreatedDate and fileModifiedDate Properties</title>
      <script type="text/javascript">
      function fillInBlanks() {
         var created = document.fileCreatedDate;
         var modified = document.fileModifiedDate;
         document.getElementById("created").innerText = created;
         document.getElementById("modified").innerText = modified;
         var createdDate = new Date(created).getTime();
         var today = new Date().getTime();
         var diff = Math.floor((today - createdDate) / (1000*60*60*24));
         document.getElementById("diff").innerText = diff;
         document.getElementById("size").innerText = document.fileSize;
      }
      </script>
   </head>
   <body onload="fillInBlanks()">
```

Continued

Listing 18-4 *(continued)*

```
      <h1>fileCreatedDate and fileModifiedDate Properties</h1>
      <hr />
      <p>This file (<span id="size"> </span> bytes) was created on <span
         id="created"> </span> and most recently modified on <span
         id="modified"> </span>.</p>
      <p>It has been <span id="diff"> </span> days since this file was
         created.</p>
   </body>
</html>
```

Related Items: `lastModified` property.

forms[]

Value: Array. Read-Only

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

As I show in Chapter 21, which is dedicated to the `form` object, an HTML form (anything defined inside a `<form>...</form>` tag pair) is a JavaScript object unto itself. You can create a valid reference to a form according to its name (assigned via a form's `name` attribute). For example, if a document contains the following form definition

```
<form name="phoneData">
   input item definitions
</form>
```

your scripts can refer to the form object by name:

```
document.phoneData
```

However, a `document` object also tracks its forms in another way: as an array of `form` objects. The first item of a `document.forms` array is the form that loaded first (it was first from the top of the HTML code). If your document defines one form, the forms property is an array one entry in length; with three separate forms in the document, the array is three entries long.

Use standard array notation to reference a particular form from the `document.forms` array. For example, the first form in a document (the "zeroth" entry of the `document.forms` array) is referenced as

```
document.forms[0]
```

Any of the form object's properties or methods are available by appending the desired property or method name to the reference. For example, to retrieve the value of an input text field named `homePhone` from the second form of a document, the reference you use is

```
document.forms[1].homePhone.value
```

One advantage to using the `document.forms` property for addressing a form object or element instead of the actual form name is that you may be able to generate a library of generalizable scripts that know how to cycle through all available forms in a document and hunt for a form that has some special element and property. The following script fragment (part of a *repeat loop* described more fully in Chapter 31) uses a loop-counting variable (`i`) to help the script check all forms in a document:

```
for (var i = 0; i < document.forms.length; i++) {
   if (document.forms[i]. ... ) {
      statements
   }
}
```

One more variation on forms array references lets you substitute the name of a form (as a string) for the `forms` array index. For example, the form named `phoneData` can be referenced as

```
document.forms["phoneData"]
```

If you use a lot of care in assigning names to objects, you will likely prefer the `document.formName` style of referencing forms. In this book, you see both indexed array and form name style references. The advantage of using name references is that even if you redesign the page and change the order of forms in the document, references to the named forms will still be valid, whereas the index numbers of the forms will have changed. See also the discussion in Chapter 21 of the `form` object and how to pass a form's data to a function.

Example

The document in Listing 18-5 is set up to display an alert dialog box that simulates navigation to a particular music site, based on the checked status of the "bluish" checkbox. The user input here is divided into two forms: one form with the checkbox and the other form with the button that does the navigation. A block of copy fills the space in between. Clicking the bottom button (in the second form) triggers the function that fetches the checked property of the "bluish" checkbox by using the `document.forms[i]` array as part of the address.

Listing 18-5: **A Simple Form Example**

```
<html>
   <head>
      <title>document.forms example</title>
      <script type="text/javascript">
      function goMusic() {
         if (document.forms[0].bluish.checked) {
            alert("Now going to the Blues music area...");
         } else {
            alert("Now going to Rock music area...");
         }
      }
      </script>
   </head>
   <body>
      <form name="theBlues">
         <input type="checkbox" name="bluish" />Check here if you've got the
         blues.
      </form>
      <hr />
      M<br />
      o<br />
      r<br />
      e<br />
      <br />
      C<br />
      o<br />
```

Continued

Listing 18-5 *(continued)*

```
      p<br />
      y<br />
      <hr />
      <form name="visit">
         <input type="button" value="Visit music site" onclick="goMusic()" />
      </form>
   </body>
</html>
```

Related Items: `form` object (Chapter 21).

`frames[]`

Value: Array. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `document.frames` property is similar to the `window.frames` property, but its association with the `document` object may seem a bit illogical at times. The objects contained by the array returned from the property are window objects, which means they are the window objects of any `frame` elements (from a framesetting document) or `iframe` elements (from a plain HTML document) defined for the document. Distinguishing the window objects from the `iframe` element objects is important. Window objects have different properties and methods than the `frame` and `iframe` element objects. The latter's properties typically represent the attributes for those element's tags. If a document contains no `iframe` elements, the `document.frames` array length is zero.

While you can access an individual frame object via the typical array syntax (for example, `document.frames[0]`), you can also use alternate syntax that Microsoft provides for collections of objects. The index number can also be placed inside parentheses, as in

```
document.frames(0)
```

Moreover, if the frames have values assigned to their `name` attributes, you can use the name (in string form) as a parameter:

```
document.frames("contents")
```

And if the collection of frames has more than one frame with the same name, you must take special care. Using the duplicated name as a parameter forces the reference to return a collection of frame objects that share that name. Or, you can limit the returned value to a single instance of the duplicate-named frames by specifying an optional second parameter indicating the index. For example, if a document has two `iframe` elements with the name `contents`, a script could reference the second `window` object as

```
document.frames("contents", 1)
```

For the sake of cross-browser compatibility, my preference for referencing frame window objects is via the `window.frames` property.

Example

See Listings 16-7 and 16-8 for examples of using the `frames` property with window objects. The listings work with IE4+ if you swap references to `window` with `document`.

Related Items: `window.frames` property.

height
width

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN4+, Moz1+, Safari1+

The `height` and `width` properties provide the pixel dimensions of the content within the current window (or frame). If the document's content is smaller than the size of the browser's content region, the dimensions returned by these properties include the blank space to the right and/or bottom edges of the content area of the window. But if the content extends beyond the viewable edges of the content region, the dimensions include the unseen content as well. The corresponding measures in IE4+ are the `document.body.scrollHeight` and `document.body.scrollWidth` properties.

Example

Use The Evaluator (Chapter 13) to examine the `height` and `width` properties of that document. Enter the following statement into the top text box and click the Evaluate button:

```
"height=" + document.height + "; width=" + document.width
```

Resize the window so that you see both vertical and horizontal scroll bars in the browser window and click the Evaluate button again. If either or both numbers get smaller, the values in the Results box are the exact size of the space occupied by the document. But if you expand the window to well beyond where the scroll bars are needed, the values extend to the number of pixels in each dimension of the window's content region.

Related Items: `document.body.scrollHeight`, `document.body.scrollWidth` properties.

ids[]

Value: Array. Read-Only
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

The NN4-specific `ids` property is used in the browser's alternative, JavaScript-based stylesheet syntax. Deployment of JavaScript stylesheets is exceedingly rare. In some ways, the `document.ids` property behaves similarly to the IE4+ `document.all` property, but `document.ids` cannot be used in regular scripts to access element objects.

Related Items: `tags` property.

images[]

Value: Array. Read-Only
Compatibility: WinIE4+, MacIE3+, NN3+, Moz1+, Safari1+

With images treated as first-class objects beginning with NN3 and IE4, it's only natural for a document to maintain an array of all the image tags defined on the page (just as it does for links and anchors). The prime importance of having images as objects is that you can modify their content (the source file associated with the rectangular space of the image) on the fly. You can find details about the `image` object in Chapter 20.

Use image array references to pinpoint a specific image for retrieval of any image property or for assigning a new image file to its `src` property. Image arrays begin their index counts with 0: The first image in a document has the reference `document.images[0]`. And, as with any array object, you can find out how many images the array contains by checking the `length` property. For example:

```
var imageCount = document.images.length;
```

Images can also have names, so if you prefer, you can refer to the image object by its name, as in

```
var imageLoaded = document.imageName.complete;
```

or

```
var imageLoaded = document.images[imageName].complete;
```

The `document.images` array is a useful guide to knowing whether a browser supports swappable images. Any browser that treats an `img` element as an object always forms a `document.images` array in the page. If no images are defined in the page, the array is still there, but its length is zero. The array's existence, however, is the clue about image object compatibility. Because the `document.images` array evaluates to an array object when present, the expression can be used as a condition expression for branching to statements that involve image swapping:

```
if (document.images) {
   // image swapping or precaching here
}
```

Earlier browsers that don't have this property evaluate `document.images` as `undefined` and thus the condition is treated as a `false` value.

Example

The `document.images` property is defined automatically as the browser builds the object model for a document that contains image objects. See the discussion about the `Image` object in Chapter 20 for reference examples.

Related Items: `Image` object (Chapter 20).

implementation

Value: Object. Read-Only
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The Core W3C DOM defines the `document.implementation` property as an avenue to let scripts find out what DOM features (that is, modules of the DOM standard) are implemented for the current environment. While the object returned by the property (a `DOMImplementation` object) has no properties, it has a method, `hasFeature()`, which lets scripts find out, for example, whether the environment supports HTML or just XML. The first parameter of the `hasFeature()` method is the feature in the form of a string. The second parameter is a string form of the version number. The method returns a Boolean value.

A section of the W3C DOM specification, called "Conformance," governs the module names (the standard also allows browser-specific features to be tested via the `hasFeature()` method). Module names include strings such as `HTML`, `XML`, `MouseEvents`, and so on.

Version numbering for W3C DOM modules corresponds to the W3C DOM level. Thus, the version for the XML DOM module in DOM Level 2 is known as `2.0`. Note that versions refer to DOM modules and not, for instance, the separate HTML standard.

Example

Use The Evaluator (Chapter 13) to experiment with the `document.implementation.hasFeature()` method. Enter the following statements one at a time into the top text field and examine the results:

```
document.implementation.hasFeature("HTML","1.0")
document.implementation.hasFeature("HTML","2.0")
document.implementation.hasFeature("HTML","3.0")
document.implementation.hasFeature("CSS","2.0")
document.implementation.hasFeature("CSS2","2.0")
```

Feel free to try other values.

`lastModified`

Value: Date string. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Every disk file maintains a modified time stamp, and most (but not all) servers are configured to expose this information to a browser accessing a file. This information is available by reading the `document.lastModified` property. If your server supplies this information to the client, you can use the value of this property to present this information for readers of your Web page. The script automatically updates the value for you, rather than requiring you to hand-code the HTML line every time you modify the home page.

If the value returned to you displays itself as a date in 1969, it means that you are positioned somewhere west of GMT, or Greenwich Mean Time (some number of time zones west of GMT at 1 January 1970), and the server is not providing the proper data when it serves the file. Sometimes server configuration can fix the problem, but not always.

The returned value is not a date object (see Chapter 29) but rather a straight string consisting of time and date, as recorded by the document's file system. The format of the string varies from browser to browser and version to version. You can, however, usually convert the date string to a JavaScript date object and use the date object's methods to extract selected elements for recompilation into readable form. Listing 18-6 shows an example.

Even local file systems don't necessarily provide the correct data for every browser to interpret. But put that same file on a UNIX or Windows Web server, and the date appears correctly when accessed via the Net.

Example

Experiment with the `document.lastModified` property with Listing 18-6. But also be prepared for inaccurate readings if the file is located on some servers or local hard disks.

Listing 18-6: **Putting a Time Stamp on a Page**

```
<html>
   <head>
      <title>Time Stamper</title>
   </head>
   <body>
      <center>
         <h1>GiantCo Home Page</h1>
      </center>
      <script type="text/javascript">
      update = new Date(document.lastModified);
      theMonth = update.getMonth() + 1;
      theDate = update.getDate();
      theYear = update.getFullYear();
      document.writeln("<I>Last updated:" + theMonth + "/" + theDate + "/" +
         theYear + "<\/I>");
      </script>
      <hr />
   </body>
</html>
```

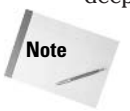
As noted at great length in the `Date` object discussion in Chapter 29, you should be aware that date formats vary greatly from country to country. Some of these formats use a different order for date elements. When you hard-code a date format, it may take a form that is unfamiliar to other users of your page.

Related Items: `Date` object (Chapter 29).

layers[]

Value: Array. Read-Only
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

The `layer` object (see Chapter 39 on the CD-ROM) is the NN4 way of exposing positioned elements to the object model. Thus, the `document.layers` property is an array of positioned elements in the document. But due to the nonstandard way that NN4 implements positioned elements, not every positioned element is represented in the `document.layers` array. More deeply nested positioned elements must be referenced through a hierarchy of layers.



The `Layer` object and `document.layers` property are orphaned in NN4, and their importance diminishes as the installed base of NN4 shrinks. The remaining discussion is included only for those Web authors who must support positioned elements in NN4. In NN6+, the layer is represented by any HTML element whose stylesheet definition includes a `position` attribute. References to such elements can be made through the `document.getElementById()` method or shortcuts described in Chapter 14.

A Netscape layer is a container for content that can be precisely positioned on the page. Layers can be defined with the NN4-specific `<layer>` tag or with W3C standard stylesheet positioning syntax, as explained in Chapter 39 on the CD-ROM. Each layer contains a `document` object — the true holder of the content displayed in that layer. Layers can be nested within each other, but a reference to `document.layers` reveals only the first level of layers defined in the document. Consider the following HTML skeleton.

```
<html>
<body>
<layer name="Europe">
   <layer name="Germany"></layer>
   <layer name="Netherlands"></layer>
</layer>
</body>
</html>
```

From the point of view of the primary document, there is one layer (Europe). Therefore, the length of the `document.layers` array is 1. But the Europe layer has a document, in which two more layers are nested. A reference to the array of those nested layers is

```
document.layers[1].document.layers
```

or

```
document.Europe.document.layers
```

The length of this nested array is two: The Germany and Netherlands layers. No property exists that reveals the entire set of nested arrays in a document, but you can create a `for` loop to crawl through all nested layers (shown in Listing 18-7).

Example

Listing 18-7 demonstrates only for NN4 how to use the `document.layers` property to crawl through the entire set of nested layers in a document. Using reflexive calls to the `crawlLayers()` function, the script builds an indented list of layers in the same hierarchy as the objects themselves and displays the results in an alert dialog box. After you load this document (the script is triggered by the `onLoad` event handler), compare the alert dialog box contents against the structure of `<layer>` tags in the document.

Listing 18-7: **Crawling Through Nested Layers in NN4**

```
<html>
   <head>
      <title></title>
      <script type="text/javascript">
      var output = "";
      function crawlLayers(layerArray, indent) {
         for (var i = 0; i < layerArray.length; i++) {
            output += indent + layerArray[i].name + "\n";
            if (layerArray[i].document.layers.length) {
               var newLayerArray = layerArray[i].document.layers;
               crawlLayers(newLayerArray, indent + "   ");
            }
         }
         return output;
      }
      function revealLayers() {
         alert(crawlLayers(document.layers, ""));
      }
      </script>
   </head>
   <body onload="revealLayers()">
      <layer name="Europe">
```

Continued

Listing 18-7 *(continued)*

```
         <layer name="Germany">
         </layer>
         <layer name="Netherlands">
            <layer name="Amsterdam"></layer>
            <layer name="Rotterdam"></layer>
         </layer>
         <layer name="France">
         </layer>
      </layer>
      <layer name="Africa">
         <layer name="South Africa">
         </layer>
         <layer name="Ivory Coast">
         </layer>
      </layer>
   </body>
</html>
```

Related Items: `layer` object (Chapter 39 on the CD-ROM).

linkColor

(See `alinkColor`)

links[]

Value: Array. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `document.links` property is similar to the `document.anchors` property, except that the objects maintained by the array are link objects — items created with `<a href="">` tags. Use the array references to pinpoint a specific link for retrieving any link property, such as the target window specified in the link's HTML definition.

Link arrays begin their index counts with 0: The first link in a document has the reference `document.links[0]`. And, as with any array object, you can find out how many entries the array has by checking the `length` property. For example:

```
var linkCount = document.links.length;
```

Entries in the `document.links` property are full-fledged `location` objects, which means you have the same properties available to each member of the `links[]` array as you do in the `location` object.

Example

The `document.links` property is defined automatically as the browser builds the object model for a document that contains link objects. You rarely access this property, except to determine the number of link objects in the document.

Related Items: `link` object; `document.anchors` property.

location
URL

Value: String. Read/Write and Read-Only (see text)
Compatibility: WinIE3+, MacIE3+, NN3+, Moz1+, Safari1+

The fact that JavaScript frequently reuses the same terms in different contexts may be confusing to the language's newcomers. Such is the case with the `document.location` property. You may wonder how this property differs from the `location` object (see Chapter 17). In practice, many scripts also get the two confused when references don't include the `window` object. As a result, a new property name, `document.URL`, was introduced in NN3 and IE4 to take the place of `document.location`. You can still use `document.location`, but the term may eventually disappear from the object model vocabulary. To help you get into the future mindset, the rest of this discussion refers to this property as `document.URL`.

The remaining question is how the `window.location` object and `document.URL` property differ. The answer lies in their respective data types.

A `location` object, you may recall from Chapter 17, consists of a number of properties about the document currently loaded in a window or frame. Assigning a new URL to the `location` object (or `location.href` property) tells the browser to load the page from that URL into the frame. The `document.URL` property, on the other hand, is simply a string (read-only in Navigator, Mozilla, and Safari) that reveals the URL of the current document. The value may be important to your script, but the property does not have the "object power" of the `window.location` object. You cannot change (assign another value to) this property value because a document has only one URL: its location on the Net (or your hard disk) where the file exists, and what protocol is required to get it.

This may seem like a fine distinction, and it is. The reference you use (`window.location` object or `document.URL` property) depends on what you are trying to accomplish specifically with the script. If the script is changing the content of a window by loading a new URL, you have no choice but to assign a value to the `window.location` object. Similarly, if the script is concerned with the component parts of a URL, the properties of the `location` object provide the simplest avenue to that information. To retrieve the URL of a document in string form (whether it is in the current window or in another frame), you can use either the `document.URL` property or the `window.location.href` property.

Example

HTML documents in Listings 18-8 through 18-10 create a test lab that enables you to experiment with viewing the `document.URL` property for different windows and frames in a multiframe environment. Results are displayed in a table, with an additional listing of the `document.title` property to help you identify documents being referred to. The same security restrictions that apply to retrieving `window.location` object properties also apply to retrieving the `document.URL` property from another window or frame.

Listing 18-8: **A Simple Frameset for the URL Example**

```
<html>
   <head>
      <title>document.URL Reader</title>
   </head>
   <frameset rows="60%,40%">
      <frame name="Frame1" src="lst18-10.htm" />
      <frame name="Frame2" src="lst18-09.htm" />
   </frameset>
</html>
```

Listing 18-9: **Showing Location Information for Different Contexts**

```
<html>
   <head>
      <title>URL Property Reader</title>
      <script type="text/javascript">
      function fillTopFrame() {
         newURL=prompt("Enter the URL of a document to show in the top
            frame:","");
         if (newURL != null && newURL != "") {
            top.frames[0].location = newURL;
         }
      }

      function showLoc(form,item) {
         var windName = item.value;
         var theRef = windName + ".document";
         form.dLoc.value = unescape(eval(theRef + ".URL"));
         form.dTitle.value = unescape(eval(theRef + ".title"));
      }
      </script>
   </head>
   <body>
      Click the "Open URL" button to enter the location of an HTML document to
      display in the upper frame of this window.
      <form>
         <input type="button" name="opener" value="Open URL..."
         onclick="fillTopFrame()" />
      </form>
      <hr />
      <form>
         Select a window or frame to view each document property values.
         <p><input type="radio" name="whichFrame" value="parent"
            onclick="showLoc(this.form,this)" />Parent window <input
            type="radio" name="whichFrame" value="top.frames[0]"
            onclick="showLoc(this.form,this)" />Upper frame <input type="radio"
            name="whichFrame" value="top.frames[1]"
```

```
            onclick="showLoc(this.form,this)" />This frame</p>
         <table border="2">
            <tr>
               <td align="RIGHT">document.URL:</td>
               <td><textarea name="dLoc" rows="3" cols="30" wrap="soft">
                  </textarea></td>
            </tr>
            <tr>
               <td align="RIGHT">document.title:</td>
               <td><textarea name="dTitle" rows="3" cols="30" wrap="soft">
                  </textarea></td>
            </tr>
         </table>
      </form>
   </body>
</html>
```

Listing 18-10: **A Placeholder Page for the URL Example**

```
<html>
   <head>
      <title>Opening Placeholder</title>
   </head>
   <body>
      Initial place holder. Experiment with other URLs for this frame (see
      below).
   </body>
</html>
```

Related Items: `location` object; `location.href`, `URLUnencoded` properties.

media

Value: String. Read/Write
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The `document.media` property indicates the output medium for which content is formatted. The property actually returns an empty string as of IE6, but the intention appears to be to provide a way to use scripting to set the equivalent of the CSS2 `@media` rule (one of the so-called "at" rules because of the at symbol). This stylesheet rule allows browsers to assign separate styles for each type of output device on which the page is rendered (for example, perhaps a different font for a printer versus the screen). In practice, however, this property is not modifiable in WinIE5.5 and WinIE6.

Related Items: None.

mimeType

Value: String. Read-Only
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

Although this property is readable in WinIE5+, its value is not strictly speaking a MIME type, or at least not in traditional MIME format. Moreover, the results are inconsistent between WinIE5 and WinIE6. Perhaps this property will be of more use in an XML, rather than HTML, document environment. In any case, this property in no way exposes supported MIME types in the current browser.

nameProp

Value: String. Read-Only
Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-

The `nameProp` property returns a string containing the title of the document, which is the same as `document.title`. If the document doesn't have a title, `nameProp` contains an empty string.

Related Items: `title` property.

namespaces[]

Value: Array of `namespace` objects. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

A `namespace` object can dynamically import an XML-based IE Element Behavior. The `namespaces` property returns an array of all `namespace` objects defined in the current document.

Related Items: None.

parentWindow

Value: `window` object reference. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `document.parentWindow` property returns a reference to the `window` object containing the current document. The value is the same as any reference to the current window.

Example

To prove the `parentWindow` property points to the document's window, you can enter the following statement into the top text field of The Evaluator (Chapter 13):

```
document.parentWindow == self
```

This expression evaluates to `true` only if both references are of the same object.

Related Items: `window` object.

plugins[]

Value: Array. Read-Only
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The document.plugins property returns the same array of embed element objects that you get from the document.embeds property. This property appears to have been deprecated in favor of document.embeds.

Related Items: document.embeds property.

protocol

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The IE-specific document.protocol property returns the plain-language version of the protocol that was used to access the current document. For example, if the file is accessed from a Web server, the property returns Hypertext Transfer Protocol. This property differs from the location.protocol property, which returns the portion of the URL that includes the often more cryptic protocol abbreviation (for example, http:). As a general rule, you want to hide all of this stuff from a Web application user.

Example

If you use The Evaluator (Chapter 13) to test the document.protocol property, you will find that it displays File Protocol in the results because you are accessing the listing from a local hard disk or CD-ROM.

Related Items: location.protocol property.

referrer

Value: String. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

When a link from one document leads to another, the second document can, under JavaScript control, reveal the URL of the document containing the link. The document.referrer property contains a string of that URL. This feature can be a useful tool for customizing the content of pages based on the previous location the user was visiting within your site. A referrer contains a value only when the user reaches the current page via a link. Any other method of navigation (such as through the history, bookmarks, or by manually entering a URL) sets this property to an empty string.

The document.referrer property usually returns an empty string unless the files are retrieved from a Web server. Also, early versions of WinIE never returned the correct referrer info.

Example

This demonstration requires two documents (and for IE, you'll also need to access the documents from a Web server). The first document, in Listing 18-11, simply contains one line of text as a link to the second document. In the second document (see Listing 18-12), a script verifies the document from which the user came via a link. If the script knows about that link, it displays a message relevant to the experience the user had at the first document. Also try opening Listing 18-12 in a new browser window from the Open File command in the File menu to see how the script won't recognize the referrer.

Listing 18-11: An Example Referrer Page

```
<html>
   <head>
      <title>document.referrer Property 1</title>
   </head>
   <body>
      <h1><a href="lst18-12.htm">Visit my sister document</a></h1>
   </body>
</html>
```

Listing 18-12: Determining the Referrer when a Page Is Visited through a Link

```
<html>
   <head>
      <title>document.referrer Property 2</title>
   </head>
   <body>
      <h1>
         <script type="text/javascript">
         alert(document.referrer.length + "    :    " + document.referrer);
         if(document.referrer.length > 0 &&
            document.referrer.indexOf("lst18-11.htm") != -1) {
            document.write("How is my brother document?");
         } else {
            document.write("Hello, and thank you for stopping by.");
         }
         </script>
      </h1>
   </body>
</html>
```

Related Items: `link` object.

scripts[]

Value: Array. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The IE-specific `document.scripts` property returns an array of all `script` element objects in the current document. You can reference an individual `script` element object to read not only the properties it shares with all HTML element objects (see Chapter 15) but also script-specific properties, such as `defer`, `src`, and `htmlFor`. The actual scripting is accessible either through the `innerText` or `text` properties for any `script` element object.

While the `document.scripts` array is read-only, many properties of individual `script` element objects are modifiable. Adding or removing `script` elements impacts the length of the

document.scripts array. Don't forget, too, that if your scripts need to access a specific script element object, you can assign an id attribute to it and reference the element directly.

This property is an IE-specific convenience property that is the same as the W3C browser expression document.getElementsByTagName("script"), which returns an array of the same objects.

Example

You can experiment with the document.scripts array in The Evaluator (Chapter 13). For example, you can see that only one script element object is in The Evaluator page if you enter the following statement into the top text field:

```
document.scripts.length
```

If you want to view all of the properties of that lone script element object, enter the following statement into the bottom text field:

```
document.scripts[0]
```

Among the properties are both innerText and text. If you assign an empty string to either property, the scripts are wiped out from the object model, but not from the browser. The scripts disappear because after the scripts loaded, they were cached outside of the object model. Therefore, if you enter the following statement into the top field:

```
document.scripts[0].text = ""
```

the script contents are gone from the object model, yet subsequent clicks of the Evaluate and List Properties buttons (which invoke functions of the script element object) still work.

Related Items: script element object (Chapter 36 on the CD-ROM).

security

Value: String. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The security property reveals information about a security certificate, if one is associated with the current document.

selection

Value: Object. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The document.selection property returns a selection object whose content is represented in the browser window as a body text selection. That selection can be explicitly performed by the user (by clicking and dragging across some text) or created under script control via the WinIE TextRange object (see Chapter 35). Because script action on a selection (for example, finding the next instance of selected text) is performed via the TextRange object, converting a selection to a TextRange object using the document.selection.createRange() method is common practice. See the selection object in Chapter 35 for more details.

Be aware that you cannot script interaction with text selections through user interface elements, such as buttons. Clicking a button gives focus to the button and deselects the selection. Use other events, such as `document.onmouseup` to trigger actions on a selection.

Example

See Listings 15-30 and 15-39 in Chapter 15 to see the `document.selection` property in action for script-controlled copying and pasting (WinIE only).

Related Items: `selection`, `TextRange` objects.

styleSheets[]

Value: Array. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `document.styleSheets` array consists of references to all `style` element objects in the document. Not included in this array are stylesheets that are assigned to elements by way of the `style` attribute inside a tag or linked in via `link` elements. See Chapter 26 for details about the `styleSheet` object.

Related Items: `styleSheet` object (Chapter 26).

tags[]

Value: Array. Read-Only
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

The NN4-specific `tags` property is used in the browser's alternate, JavaScript-based stylesheet syntax. Deployment of JavaScript stylesheets is exceedingly rare. In some ways, the `document.tags` property behaves like the W3C DOM `document.getElementsByTagName()` method, but `document.tags` cannot be used in regular scripts to access element objects.

Related Items: `ids` property.

title

Value: String. Read-Only and Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

A document's title is the text that appears between the `<title>...</title>` tag pair in an HTML document's Head portion. The title usually appears in the title bar of the browser window in a single-frame presentation. Only the title of the topmost framesetting document appears as the title of a multiframe window. Even so, the `title` property for an individual document within a frame is available via scripting. For example, if two frames are available (`UpperFrame` and `LowerFrame`), a script in the document occupying the `LowerFrame` frame can reference the `title` property of the other frame's document, such as this:

```
parent.UpperFrame.document.title
```

This property is read-only in browsers prior to IE4 and NN6+ but not all browsers allow scripts to change the text that appears in the browser window's title bar.

The `document.title` property is a holdover from the original document object model. HTML elements in recent browsers have an entirely different application of the `title` property (see Chapter 15). In modern browsers (IE4+/NN6+), you should address the document's title by way of the `title` element object directly.

Related Items: `history` object.

URL

(See `location`)

URLUnencoded

Value: String. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The `document.URL` property returns a URL-encoded string, meaning that non-alphanumeric characters in the URL are converted to URL-friendly characters (for example, a space becomes `%20`). You can always use the `unescape()` function on the value returned by the `document.URL` property, but the `URLUnencoded` property does that for you. If there are no URL-encoded characters in the URL, then both properties return identical strings.

Related Items: `document.URL` property.

vlinkColor

(See `alinkColor`)

width

(See `height`)

Methods

captureEvents(*eventTypeList*)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

In Navigator 4 only, the natural propagation of an event is downward from the `window` object, through the `document` object, and eventually reaching its target. For example, if you click a button, the `click` event first reaches the `window` object; then it goes to the `document` object; if the button is defined within a layer, the event also filters through that layer; eventually (in a split second) the event reaches the button, where an `onclick` event handler is ready to act on that click.

The NN4 mechanism allows `window`, `document`, and layer objects to intercept events and process them prior to reaching their intended targets (or preventing them from reaching their destinations entirely). But for an outer container to grab an event, your script must instruct it to capture the type of event your application is interested in preprocessing. If you want the `document` object to intercept all events of a particular type, use the `document.captureEvents()` method to turn that facility on.

Event capture with different syntax has been standardized in the W3C DOM and is implemented in W3C browsers, such as NN6+ and Mozilla. See the `addEventListener()` method in Chapter 15 for the W3C counterpart to the NN4 `captureEvents()` method. Also, see Chapter 25 for more details on the combination of event capture and event bubbling in the W3C DOM.

The `document.captureEvents()` method takes one or more event types as parameters. An event type is a constant value built inside the NN4 `Event` object. One event type exists for every kind of event handler that you see in all of the document objects of NN4. The syntax

consists of a reference to the `Event` object and the event name in all uppercase letters. For example, if you want the document to intercept all `click` events, the statement is

```
document.captureEvents(Event.CLICK);
```

For multiple events, add them as parameters, separated by the pipe (|) character:

```
document.captureEvents(Event.MOUSEDOWN | Event.KEYPRESS);
```

After the `document` object is set to capture an event type, it must have a function ready to deal with the event. For example, perhaps the function looks through all `Event.MOUSEDOWN` events and looks to see if the right mouse button is the one that triggers the event and what form element (if any) is the intended target. The goal is perhaps to display a pop-up menu (as a separate layer) for a right-click. If the click comes from the left mouse button, the event is routed to its intended target.

To associate a function with a particular event type captured by a `document` object, assign a function to the event. For example, to assign a custom `doClickEvent()` function to click events captured by the `document` object, use the following statement:

```
document.onclick = doClickEvent;
```

Notice that the function name is assigned only as a reference name, unlike an event handler within a tag. The function, itself, is like any function, but it has the added benefit of automatically receiving the event object as a parameter. To turn off event capture for one or more event types, use the `document.releaseEvent()` method. See Chapter 25 for details of working with NN4 events.

Capturing events at the `window`, `document`, or layer level in NN4 does not always work the way you may want, which is especially true if your page contains tables. For example, capturing mouse events has no effect in the Windows version of NN4 unless the cursor is atop a cell border. Event capture works most reliably when a scriptable object has an event handler defined for it (even if the handler is an empty string), and the element is the target of the event (for example, you are about to type into a text field). For all other elements, event capture may simply not be captured at the `document` or `window` level.

Example

See the example for the NN4 `window.captureEvents()` method in Listing 16-21 to see how to capture events on their way to other objects. In that example, you can substitute the `document` reference for the `window` reference to see how the document version of the method works just like the window version. If you understand the mechanism for windows, you understand it for documents. The same is true for the other NN4 event methods.

Related Items: `document.handleEvent()`, `document.releaseEvents()`, `document.routeEvent()` methods; parallel `window` object event methods.

clear()

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Ever since NN2, the `document.clear()` method was intended to clear the current document from the browser window. This method is quite impractical, because you typically need some further scripts to execute after you clear the document, but if the scripts are gone, nothing else happens.

Fixing the Sticky Wait Cursor

From time to time, various browsers fail to restore the cursor to normal after `document.write()` and `document.close()` (and some other content-modifying scripts). The cursor stubbornly remains in the wait mode or the progress bar keeps spinning when, in truth, all processing has been completed. One, albeit ugly, workaround that I have found effective is to force an extra `document.close()` via a `javascript:` pseudo-URL (just adding another `document.close()` to your script doesn't do the trick). For use within a frameset, the `javascript:` URL must be directed to the top of the frameset hierarchy, while the `document.close()` is aimed at the frame that had its content changed. For example, if the change is made to a frame named `content`, create a function, such as the following:

```
function recloseDoc() {
   top.location.href =
      "javascript:void (parent.content.document.close())";
}
```

If you place this function in the framesetting document, scripts that modify the content frame can invoke this script after any operation that prevents the normal cursor from appearing.

In practice, the `document.clear()` method never did what it was supposed to do (and in earlier browsers easily caused browser crashes). I recommend against using `document.clear()`, including in preparation for generating a new page's content with `document.write()`. The `document.write()` method clears the original document from the window before adding new content. If you truly want to empty a window or frame, then use `document.write()` to write a blank HTML document or to load an empty HTML document from the server.

Related Items: `document.close()`, `document.write()`, `document.writeln()` methods.

`close()`

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Whenever a layout stream is opened to a window via the `document.open()` method or either of the document writing methods (which also open the layout stream), you must close the stream after the document is written. This causes the `Layout:Complete` and `Done` messages to appear in the status line (although you may experience some bugs in the status message on some platforms). The document-closing step is very important to prepare the window for the next potential round of replenishment with new script-assembled HTML. If you don't close the document, subsequent writing is appended to the bottom of the document.

Some or all of the data specified for the window won't display properly until you invoke the `document.close()` method, especially when images are being drawn as part of the document stream. A common symptom is the momentary appearance and then disappearance of the document parts. If you see such behavior, look for a missing `document.close()` method after the last `document.write()` method.

Example

Before you experiment with the `document.close()` method, be sure you understand the `document.write()` method described later in this chapter. After that, make a separate set of

the three documents for that method's example (Listings 18-16 through 18-18 in a different directory or folder). In the `takePulse()` function listing, comment out the `document.close()` statement, as shown here:

```
msg += "<p>Make it a great day!</body></html>";
parent.frames[1].document.write(msg);
//parent.frames[1].document.close();
```

Now try the pages on your browser. You see that each click of the upper button appends text to the bottom frame, without first removing the previous text. The reason is that the previous layout stream was never closed. The document thinks that you're still writing to it. Also, without properly closing the stream, the last line of text may not appear in the most recently written batch.

Related Items: `document.open()`, `document.clear()`, `document.write()`, `document.writeln()` methods.

createAttribute("*attributeName*")

Returns: Attribute object reference.
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The `document.createAttribute()` method generates an attribute node object (formally known as an `Attr` object in W3C DOM terminology) and returns a reference to the newly created object. Invoking the method assigns only the name of the attribute, so it is up to your script to assign a value to the object's `nodeValue` property and then plug the new attribute into an existing element via that element's `setAttributeNode()` method (described in Chapter 15). The following sequence generates an attribute that becomes an attribute of a `table` element:

```
var newAttr = document.createAttribute("width");
newAttr.nodeValue = "80%";
document.getElementById("myTable").setAttributeNode(newAttr);
```

Attributes do not always have to be attributes known to the HTML standard, because the method also works for XML elements, which have custom attributes.

Example

To create an attribute and inspect its properties, enter the following text into the top text box of The Evaluator (Chapter 13):

```
a = document.createAttribute("author")
```

Now enter `a` into the bottom text box to inspect the properties of an `Attr` object.

Related Items: `setAttributeNode()` method (Chapter 15).

createComment("*commentText*")

Returns: Comment object reference.
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The `document.createComment()` method creates an instance of a comment node. Upon creation, the node is in memory and available to be inserted into the document via any node's `appendChild()` or `insertBefore()` method.

Related Items: `appendChild()` and `insertBefore()` methods.

createDocumentFragment()

Returns: Document fragment object reference.
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The `document.createDocumentFragment()` method creates an instance of an empty document fragment node. This node serves as a holder that can be used to assemble a sequence of nodes in memory. After creating and assembling nodes into the document fragment, the entire fragment can be inserted into the document tree.

A document fragment is particularly helpful when your scripts assemble an arbitrary sequence of element and text nodes. By providing a parent node for all content within, the fragment node supplies the necessary parent node context for W3C DOM node methods, such as `appendChild()` during the content assembly process. If you then append or insert the document fragment node to an element in the rendered document tree, the fragment wrapper disappears, leaving just its content in the desired location in the document. Therefore, a typical usage pattern for a document fragment is to begin by creating an empty fragment node (via the `createDocumentFragment()` method), populate it at will with newly created element and/or text nodes, and then use the appropriate node method on a document tree's element to append, insert, or replace using the fragment node as the source material.

Related Items: None.

createElement("*tagName*")

Returns: Element object reference.
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `document.createElement()` method generates an element object for whatever HTML (or XML) tag name you pass as the parameter. An object created in this manner is not officially part of the current document node tree because it has not yet been placed into the document. But this method is the way you begin assembling an element object that eventually gets inserted into the document.

The returned value is a reference to the object. Properties of that object include all properties (set to default values) that the browser's object model defines for that element object. Your scripts can then address the object via this reference to set the object's properties. Typically you do this before the object is inserted into the document, especially because otherwise read-only properties can be modified before the element is inserted into the document.

After the object is inserted into the document, the original reference (for example, a global variable used to store the value returned from the `createElement()` method) still points to the object, even while it is in the document and being displayed for the user. To demonstrate this effect, consider the following statements that create a simple paragraph element containing a text node:

```
var newText = document.createTextNode("Four score and seven years ago...");
var newElem = document.createElement("p");
newElem.id = "newestP";
newElem.appendChild(newText);
document.body.appendChild(newElem);
```

At this point, the new paragraph is visible in the document. But you can now modify, for example, the style of the paragraph by addressing either the element in the `document` object model or the variable that holds the reference to the object you created:

```
newElem.style.fontSize = "20pt";
```

or

```
document.getElementById("newestP").style.fontSize = "20pt";
```

The two references are inextricably connected and always point to the exact same object. Therefore, if you want to use a script to generate a series of similar elements (for example, a bunch of `li` elements), then you can use `createElement()` to make the first one and set all properties that the items have in common. Then use `cloneNode()` to make a new copy, which you can then treat as a separate element (and probably assign unique IDs to each one).

When scripting in the W3C DOM environment, you may rely on `document.createElement()` frequently to generate new content for a page or portion thereof (unless you prefer to use the convenience `innerHTML` property to add content in the form of strings of HTML). In a strict W3C DOM environment, creating new elements is not a matter of assembling HTML strings, but rather creating genuine element (and text node) objects.

You may use this method to generate any HTML element in the supported browsers except IE4, which limits you to `area`, `img`, and `option` elements.

Example

Chapter 15 contains numerous examples of the `document.createElement()` method in concert with methods that add or replace content to a document. See Listings 15-10, 15-21, 15-22,15-28, 15-29, and 15-31.

Related Items: `document.createTextNode()` method.

`createEvent("`*`eventType`*`")`

Returns: Event object reference.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `document.createEvent()` method creates an instance of a W3C DOM `Event` object of the specified event category. Upon creation, the generic event must be initialized as a particular event type, and any other relevant properties set for the event. After successfully initializing the event, you can fire it via a call to the `dispatchEvent()` method.

Event types recognized by NN6+/Moz are `KeyEvents`, `MouseEvents`, `MutationEvents`, and `UIEvents`. The process of initializing each of these event types requires its own series of parameters in the associated `initEvent()` method. See Chapter 25 for more details.

Example

Following is an example of how you might create an event, initialize it to a specific event type, and send it to a given element:

```
var evt = document.createEvent("MouseEvents");
evt.initEvent("mouseup", true, true);
document.getElementById("myButton").dispatchEvent(evt);
```

Related Items: `createEventObject()` method; W3C DOM event object (Chapter 25).

createEventObject([*eventObject*])

Returns: event object.
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The IE-specific createEventObject() method creates an event object, which can then be passed as a parameter to the fireEvent() method of any element object. The event object created by this event is just like an event object created by a user or system action.

An optional parameter lets you base the new event on an existing event object. In other words, the properties of the newly created event object pick up all the properties of the event object passed as a parameter, which lets you then modify properties of your choice. If you provide no parameter to the method, then you must fill the essential properties manually. For more about the properties of an event object, see Chapter 25.

Example

See the discussion of the fireEvent() method in Chapter 15 for an example of the sequence to follow when creating an event to fire on an element.

Related Items: createEvent() method; fireEvent() method (Chapter 15); event object (Chapter 25).

createRange()

Returns: Range object reference.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The document.createRange() method creates an empty W3C DOM Range object with the boundary points of the range collapsed to the point before the first character of the rendered body text.

Related Items: Range object.

createStyleSheet(["*URL*"[, *index*]])

Returns: styleSheet object reference.
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The IE-specific createStyleSheet() method creates a styleSheet object, a type of object that includes style element objects as well as stylesheets that are imported into a document via the link element. Thus you can dynamically load an external stylesheet even after a page has loaded.

Unlike the other "create" methods entering W3C DOM usage, the createStyleSheet() method not only creates the stylesheet, but it inserts the object into the document object model immediately. Thus, any stylesheet rules that belong (or are assigned to) that object take effect on the page right away. If you'd rather create a stylesheet and delay its deployment, you should use the createElement() method and element object assembly techniques.

If you don't specify any parameters to the method in WinIE, an empty styleSheet object is created. It is assumed that you will then use styleSheet object methods, such as addRule() to add the details to the stylesheet. To link in an external stylesheet file, assign the file's URL to the first parameter of the method. The newly imported stylesheet is appended to the end of

the document.styleSheets array of styleSheet objects. An optional second parameter lets you specify precisely where in the sequence of stylesheet elements the newly linked stylesheet should be inserted. A stylesheet rule for any given selector is overridden by a stylesheet for the same selector that appears later in the sequence of stylesheets in a document.

Example

Listing 18-13 demonstrates adding an internal and external stylesheet to a document. For the internal addition, the addStyle1() function invokes document.createStyleSheet() and adds a rule governing the p elements of the page (not available for MacIE5). In the addStyle2() function, an external file is loaded. That file contains the following two style rules:

```
h2 {font-size:20pt; color:blue}
p  {color:blue}
```

Notice that by specifying a position of zero for the imported stylesheet, the addition of the internal stylesheet always comes afterward in styleSheet object sequence. Thus, except when you deploy only the external stylesheet, the red text color of the p elements override the blue color of the external stylesheet. If you remove the second parameter of the createStyleSheet() method in addStyle2(), the external stylesheet is appended to the end of the list. If it is the last stylesheet to be added, the blue color prevails. Repeatedly clicking the buttons in this example continues to add the stylesheets to the document.

Listing 18-13: **Creating and Applying Stylesheets**

```
<html>
   <head>
      <title>document.createStyleSheet() Method</title>
      <script type="text/javascript">
      function addStyle1() {
         var newStyle = document.createStyleSheet();
         newStyle.addRule("P", "font-size:16pt; color:red");
      }

      function addStyle2() {
         var newStyle = document.createStyleSheet("lst18-13.css",0);
      }
      </script>
   </head>
   <body>
      <h1>document.createStyleSheet() Method</h1>
      <hr />
      <form>
         <input type="button" value="Add Internal"
         onclick="addStyle1()" />  <input type="button"
         value="Add External" onclick="addStyle2()" />
      </form>
      <h2>Section 1</h2>
      <p>Lorem ipsum dolor sit amet, consectetaur adipisicing elit, sed do
         eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
         adminim veniam, quis nostrud exercitation ullamco laboris nisi ut
         aliquip ex ea commodo consequat.</p>
      <h2>Section 2</h2>
```

```
      <p>Duis aute irure dolor in reprehenderit involuptate velit esse cillum
         dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
         proident, sunt in culpa qui officia deseruntmollit anim id est
         laborum.</p>
   </body>
</html>
```

Related Items: `styleSheet` object (Chapter 26).

`createTextNode("text")`

Returns: Object.
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

A text node is a W3C DOM object that contains body text without any HTML (or XML) tags, but is usually contained by (meaning, is a child of) an HTML (or XML) element. Without the IE `innerText` convenience property for modifying the text of an element, the W3C DOM relies on the node hierarchy of a document (NN6+/Moz1+ exceeds the W3C DOM by providing an `innerHTML` property, which you can use to replace text in an element). To insert or replace text inside an HTML element in the W3C DOM way, you create the text node and then use methods of the parent element (for example, `appendChild()`, `insertBefore()`, and `replaceChild()`, all described in Chapter 15) to modify the document's content. To generate a fresh text node, use `document.createTextNode()`.

The sole parameter of the `createTextNode()` method is a string whose text becomes the `nodeValue` of the text node object returned by the method. You can also create an empty text node (passing an empty string) and assign a string to the `nodeValue` of the object later. As soon as the text node is present in the document object model, scripts can simply change the `nodeValue` property to modify text of an existing element. For more details on the role of text nodes in the W3C DOM, see Chapter 14.

Example

Although Chapters 14 and 15 (Listing 15-21, for instance) provide numerous examples of the `createTextNode()` method at work, using The Evaluator (Chapter 13) is instructive to see just what the method generates in IE5+/W3C. You can use one of the built-in global variables of The Evaluator to hold a reference to a newly generated text node by entering the following statement into the top text field:

```
a = document.createTextNode("Hello")
```

The Results box shows that an object was created. Now, look at the properties of the object by typing `a` into the bottom text field. The precise listings of properties varies between IE5+ and W3C browsers, but the W3C DOM properties that they share in common indicate that the object is a node type 3 with a node name of `#text`. No parents, children, or siblings exist yet because the object created here is not part of the document hierarchy tree until it is explicitly added to the document.

To see how insertion works, enter the following statement into the top text field to append the text node to the `myP` paragraph:

```
document.getElementById("myP").appendChild(a)
```

The word "Hello" appears at the end of the simple paragraph lower on the page. Now you can modify the text of that node either via the reference from the point of view of the containing `p` element or via the global variable reference for the newly created node:

```
document.getElementById("myP").lastChild.nodeValue = "Howdy"
```

or

```
a.nodeValue = "Howdy"
```

Related Items: `document.createElement()` method.

createTreeWalker(*rootNode, whatToShow, filterFunction, entityRefExpansion*)

Returns: `TreeWalker` object reference.
Compatibility: WinIE-, MacIE-, NN7+, Moz1.4+, Safari-

The `document.createTreeWalker()` method creates an instance of a `TreeWalker` object that can be used to navigate the document tree. The first parameter to the method indicates the node in the document that is to serve as the root node of the tree. The second parameter is an integer constant that specifies one of several built-in filters for selecting nodes to be included in the tree. Following are the possible acceptable values for this parameter:

```
NodeFilter.SHOW_ALL                 NodeFilter.SHOW_ATTRIBUTE
NodeFilter.SHOW_CDATA_SECTION       NodeFilter.SHOW_COMMENT
NodeFilter.SHOW_DOCUMENT            NodeFilter.SHOW_DOCUMENT_FRAGMENT
NodeFilter.SHOW_DOCUMENT_TYPE       NodeFilter.SHOW_ELEMENT
NodeFilter.SHOW_ENTITY              NodeFilter.SHOW_ENTITY_REFERENCE
NodeFilter.SHOW_NOTATION            NodeFilter.SHOW_PROCESSING_INSTRUCTION
NodeFilter.SHOW_TEXT
```

The third parameter to the `createNodeIterator()` method is a reference to a filter function that can filter nodes even further than the *whatToShow* parameter. This function must accept a single node and return an integer value based upon one of the following constants: `NodeFilter.FILTER_ACCEPT`, `NodeFilter.FILTER_REJECT`, or `NodeFilter.FILTER_SKIP`. The idea is that you code the function to perform a test on each node and return an indicator value that lets the node iterator know whether or not to include the node in the tree. Your function doesn't loop through nodes. The `TreeWalker` object mechanism repetitively invokes the function as needed to look for the presence of whatever characteristic you wish to use as a filter.

The final parameter to the method is a Boolean value that determines whether or not the content of entity reference nodes should be treated as hierarchical nodes. This parameter applies primarily to XML documents.

Related Items: `TreeWalker` object.

elementFromPoint(*x, y*)

Returns: Element object reference.
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The IE-specific `elementFromPoint()` method returns a reference to whatever element object occupies the point whose integer coordinates are supplied as parameters to the method. The coordinate plane is that of the document, whose top-left corner is at point 0,0. This coordinate plane can be very helpful in interactive designs that need to calculate collision detection between positioned objects or mouse events.

When more than one object occupies the same point (for example, one element is positioned atop another), the element with the highest z-index value is returned. A positioned element always wins when placed atop a normal body-level element. And if multiple overlapping positioned elements have the same z-index value (or none by default), the element that comes last in the source code order is returned for the coordinate that they share in common.

Example

Listing 18-14 is a document that contains many different types of elements, each of which has an ID attribute assigned to it. The `onmouseover` event handler for the `document` object invokes a function that finds out which element the cursor is over when the event fires. Notice that the event coordinates are `event.clientX` and `event.clientY`, which use the same coordinate plane as the page for their point of reference. As you roll the mouse over every element, its ID appears on the page. Some elements, such as `br` and `tr`, occupy no space in the document, so you cannot get their IDs to appear. On a typical browser screen size, a positioned element rests atop one of the paragraph elements so that you can see how the `elementFromPoint()` method handles overlapping elements. If you scroll the page, the coordinates for the event and the page's elements stay in sync.

Listing 18-14: **Tracking the Mouse as it Passes over Elements**

```
<html>
   <head>
      <title>document.elementFromPoint() Method</title>
      <script type="text/javascript">
      function showElemUnderneath() {
         var elem = document.elementFromPoint(event.clientX, event.clientY);
         document.getElementById("mySpan").innerText = elem.id;
      }

      document.onmouseover = showElemUnderneath;
      </script>
   </head>
   <body id="myBody">
      <h1 id="header">document.elementFromPoint() Method</h1>
      <hr id="myHR" />
      <p id="instructions">Roll the mouse around the page. The coordinates
         of the mouse pointer are currently atop an element<br id="myBR" />
         whose ID is:"<span id="mySpan" style="font-weight:bold"></span>".</p>
      <form id="myForm">
         <input id="myButton" type="button" value="Sample Button"
         onclick="" /> 
      </form>
      <table border="1" id="myTable">
         <tr id="tr1">
            <td id="td_A1">Cell A1</td>
            <td id="td_B1">Cell B1</td>
```

Continued

Listing 18-14 *(continued)*

```
         </tr>
         <tr id="tr2">
            <td id="td_A2">Cell A2</td>
            <td id="td_B2">Cell B2</td>
         </tr>
      </table>
      <h2 id="sec1">Section 1</h2>
      <p id="p1">Lorem ipsum dolor sit amet, consectetaur adipisicing elit, sed
         do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
         adminim veniam, quis nostrud exercitation ullamco laboris nisi ut
         aliquip ex ea commodo consequat.</p>
      <h2 id="sec2">Section 2</h2>
      <p id="p2">Duis aute irure dolor in reprehenderit involuptate velit esse
         cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
         cupidatat non proident, sunt in culpa qui officia deseruntmollit anim
         id est laborum.</p>
      <div id="myDIV"
      style="position:absolute; top:340; left:300; background-color:yellow">
         Here is a positioned element.
      </div>
   </body>
</html>
```

Related Items: `event.clientX`, `event.clientY` properties; positioned objects (Chapter 39 on the CD-ROM).

execCommand("*commandName*"[, *UIFlag*] [, *param*])

Returns: Boolean.
Compatibility: WinIE4+, MacIE-, NN7.1, Moz1.4, Safari-

WinIE4+ includes a large set of commands that are outside of the methods defined for objects in the object model. These commands are also accessible to programmers who build an Internet Explorer ActiveX control into their applications. The `execCommand()` method is the JavaScript gateway to those commands. A series of related methods (`queryCommandEnable()` and others) also facilitate management of these commands.

The syntax for the `execCommand()` method requires at least one parameter, a string version of the command name. Command names are not case-sensitive. An optional second parameter is a Boolean flag to instruct the command to display any user interface artifacts that may be associated with the command. The default is `false`. For the third parameter, some commands require that an attribute value be passed for the command to work. For example, to set the font size of a text range, the syntax is

```
myRange.execCommand("FontSize", true, 5);
```

The `execCommand()` method returns Boolean `true` if the command is successful; `false` if not successful. Some commands can return values (for example, finding out the font name of a selection), but those are accessed through the `queryCommandValue()` method.

Most of these commands operate on body text selections that are TextRange objects. As described in Chapter 35, a TextRange object must be created under script control. But a TextRange object can be done in response to a user selecting some text in the document. Because a TextRange object is independent of the element hierarchy (indeed, a TextRange can spread across multiple nodes), it cannot respond to stylesheet specifications. Thus, many of the commands that can operate on a TextRange object have to do with formatting or modifying the text. For a list of commands that work exclusively on TextRange objects, see the TextRange.execCommand() method in Chapter 35.

While many of the commands intended for the TextRange also work when invoked from the document object, in this section the focus is on those commands that have scope over the entire document. Table 18-3 lists those few commands that work with the document. Also listed are many commands that work exclusively on text selections in the document, whether the selections are made manually by the user or with the help of the TextRange object (see Chapter 35).

Table 18-3: document.execCommand() Commands

Command	*Parameter*	*Description*
Refresh	None	Reloads the page.
SelectAll	None	Selects entire page content.
Unselect	None	Unselects any page selection.
BackColor	Color String	Encloses the current selection with a font element whose style attribute sets the background-color style to the parameter value.
CreateBookmark	Anchor String	Encloses the current selection (or text range) with an anchor element whose name attribute is set to the parameter value.
CreateLink	URL String	Encloses the current selection with an a element whose href attribute is set to the parameter value.
FontName	Font Face(s)	Encloses the current selection with a font element whose face attribute is set to the parameter value.
FontSize	Size String	Encloses the current selection with a font element whose size attribute is set to the parameter value.
FontColor	Color String	Encloses the current selection with a font element whose color attribute is set to the parameter value.
Indent	None	Indents the current selection.
JustifyCenter	None	Centers the current selection.
JustifyFull	None	Full-justifies the current selection.
JustifyLeft	None	Left-justifies the current selection.
JustifyRight	None	Right-justifies the current selection.
Outdent	None	Outdents the current selection.
RemoveFormat	None	Removes formatting for the current selection.
SelectAll	None	Selects all text of the document.
UnBookmark	None	Removes anchor tags that surround the current selection.
Unlink	None	Removes link tags that surround the current selection.
Unselect	None	Deselects the current selection anywhere in the document.

Mozilla 1.4 includes a feature that allows scripts to turn an `iframe` element's document object into an HTML editable document. Part of the scripting incorporates the `document.execCommand()` method. Visit `http://www.mozilla.org/editor` for current details and examples.

Example

You can find many examples of the `execCommand()` method for the `TextRange` object in Chapter 35. But you can try out the document-specific commands in The Evaluator (Chapter 13) if you like. Try each of the following statements in the top text box and click the Evaluate button:

```
document.execCommand("Refresh")
document.execCommand("SelectAll")
document.execCommand("Unselect")
```

All methods return `true` in the Results box.

Because any way you can evaluate a statement in The Evaluator forces a body selection to become deselected before the evaluation takes place, you can't experiment this way with the selection-oriented commands.

Related Items: `queryCommandEnabled()`, `queryCommandIndterm()`, `queryCommandState()`, `queryCommandSupported()`, `queryCommandText()`, `queryCommandValue()` methods.

`getElementById("elementID")`

Returns: Element object reference.
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The `document.getElementById()` method is the W3C DOM syntax for retrieving a reference to any element in a document that has a unique identifier assigned to its `id` attribute. If the document contains more than one instance of an ID, the method returns a reference to the first element in source code order with that ID. Because this method is such an important avenue to writing references to objects that are to be modified under script control, you can see how important it is to assign unique IDs to elements.

This method's name is quite a finger twister for scripters, especially compared to the IE4+ convention of letting a reference to any element begin simply with the object's ID. However, the `getElementById()` method is the cross-browser way of acquiring an element's reference for W3C DOM-compatible browsers. When you type this method, be sure to use a lowercase "d" as the last character of the method name.

Unlike some other element-oriented methods (for example, `getElementsByTagName()`), which can be invoked on any element in a document, the `getElementById()` method works exclusively with the `document` object.

Example

You can find many examples of this method in use throughout this book, but you can take a closer look at how it works by experimenting in The Evaluator (Chapter 13). A number of elements in The Evaluator have IDs assigned to them, so that you can use the method to inspect the objects and their properties. Enter the following statements into both the top and bottom text fields of The Evaluator. Results from the top field are references to the objects; results from the bottom field are lists of properties for the particular object.

```
document.getElementById("myP")
document.getElementById("myEM")
document.getElementById("myTitle")
document.getElementById("myScript")
```

Related Items: `getElementsByTagName()` method (Chapter 15).

`getElementsByName("elementName")`

Returns: Array.
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The `document.getElementsByName()` method returns an array of references to objects whose `name` attribute is assigned the element name passed as the method's attribute. Although NN6+/Moz1+ recognizes `name` attributes even for elements that don't have them by default, IE5+ does not. Therefore, for maximum cross-browser compatibility, use this method only to locate elements that have `name` attributes defined for them by default, such as form control elements. If the element does not exist in the document, the method returns an array of zero length.

For the most part, you are best served by using IDs on elements and the `getElementById()` method to unearth references to individual objects. But some elements, especially the `input` element of type radio, use the `name` attribute to group elements together. In that case, a call to `getElementsByName()` returns an array of all elements that share the name — facilitating perhaps a `for` loop that inspects the `checked` property of a radio button group. Thus, instead of using the old-fashioned (although entirely backward-compatible) approach by way of the containing form object

```
var buttonGroup = document.forms[0].radioGroupName;
```

you can go more directly:

```
var buttonGroup = document.getElementsByName(radioGroupName);
```

In the latter case, you operate independently of the containing form object's index number or name. This assumes, of course, that a group name is not shared elsewhere on the page.

Example

Use The Evaluator (Chapter 13) to test out the `getElementsByName()` method. All form elements in the upper part of the page have names associated with them. Enter the following statements into the top text field and observe the results:

```
document.getElementsByName("output")
document.getElementsByName("speed").length
document.getElementsByName("speed")[0].value
```

You can also explore all of the properties of the text field by typing the following expression into the bottom field:

```
document.getElementsByName("speed")[0]
```

Related Items: `document.getElementsById()`, `getElementsByTagName()` methods.

getSelection()

Returns: String.
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

Many Web browser users aren't aware that they can select and copy body text in a document for pasting into other application documents. Even so, NN4+ offers a scripted way of capturing the text selected by a user in a page. The `document.getSelection()` method returns the string of text selected by the user. If nothing is selected, an empty string is the result. Returned values consist only of the visible text on the page and not the underlying HTML or style of the text. This method generates JavaScript Console warning messages in Mozilla-based browsers, urging you to use the newer, more sophisticated `window.getSelection()` method instead.

The WinIE4+ equivalent involves the `document.selection` property, which returns an IE `selection` object. To derive the text from this object, you must create a `TextRange` object from it and then inspect the `text` property:

```
var selectedText = document.selection.createRange().text;
```

Example

The document in Listing 18-15 provides a cross-browser (but not MacIE5) solution to capturing text that a user selects in the page. Selected text is displayed in the textarea. The script uses browser detection and branching to accommodate the diverse ways of recognizing the event and reading the selected text.

Listing 18-15: Retrieving Selected Text

```
<html>
   <head>
      <title>Getting Selected Text</title>
      <script type="text/javascript">
      var isNav4 = (navigator.appName == "Netscape" &&
         parseInt(navigator.appVersion) == 4);
      var isNav4Min = (navigator.appName == "Netscape" &&
         parseInt(navigator.appVersion) >= 4);
      var isIE4Min = (navigator.appName.indexOf("Microsoft") != -1 &&
         parseInt(navigator.appVersion) >= 4);

      function showSelection() {
         if (isNav4Min) {
            document.forms[0].selectedText.value = document.getSelection();
         } else if (isIE4Min) {
            if (document.selection) {
               document.forms[0].selectedText.value =
                  document.selection.createRange().text;
               event.cancelBubble = true;
            }
         }
      }

      if (isNav4) {
         document.captureEvents(Event.MOUSEUP);
      }
```

```
      document.onmouseup = showSelection;
      </script>
   </head>
   <body>
      <h1>Getting Selected Text</h1>
      <hr />
      <p>Select some text and see how JavaScript can capture the selection:</p>
      <h2>ARTICLE I</h2>
      <p>Congress shall make no law respecting an establishment of religion, or
         prohibiting the free exercise thereof; or abridging the freedom of
         speech, or of the press; or the right of the people peaceably to
         assemble, and to petition the government for a redress of
         grievances.</p>
      <form>
         <textarea name="selectedText" rows="3" cols="40" wrap="virtual">
         </textarea>
      </form>
   </body>
</html>
```

Related Items: `document.selection` property.

handleEvent(event)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

When you explicitly capture NN4 events in the `window`, `document`, or layer object (by invoking the `captureEvents()` method for that object), you can control where the events go after their initial capture. To let an event continue to its original target (for example, a button that is clicked by a user), you use the `routeEvent()` method. But if you want to redirect an event (or class of events) to a particular event handler elsewhere in the document, use the `handleEvent()` method.

See the discussion of the `handleEvent()` method for the `window` object in Chapter 16. The behavior of the `handleEvent()` method for all objects is the same.

Related Items: `document.captureEvents()`, `document.releaseEvents()`, `document.routeEvent()` methods; `event` object (Chapter 25).

open(["*mimeType*"] [,"*replace*"])

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Opening a document is different from opening a window. In the case of a window, you're creating a new object, both on the screen and in the browser's memory. Opening a document, on the other hand, tells the browser to get ready to accept some data for display in the window named or implied in the reference to the `document.open()` method. (For example, `parent.frames[1].document.open()` may refer to a different frame in a frameset, whereas `document.open()` implies the current window or frame.) Therefore, the method name may mislead newcomers because the `document.open()` method has nothing to do with loading documents from the Web server or hard disk. Rather, this method is a prelude to sending data to a window via the `document.write()` or `document.writeln()` methods. In a sense,

the document.open() method merely opens the valve of a pipe; the other methods send the data down the pipe like a stream, and the document.close() method closes that valve as soon as the page's data has been sent in full.

The document.open() method is optional because a document.write() method that attempts to write to a closed document automatically clears the old document and opens the stream for a new one. Whether or not you use the document.open() method, be sure to use the document.close() method after all the writing has taken place.

An optional parameter to the document.open() method lets you specify the nature of the data being sent to the window. A MIME (Multipurpose Internet Mail Extension) type is a specification for transferring and representing multimedia data on the Internet (originally for mail transmission, but now applicable to all Internet data exchanges). You've seen MIME depictions in the list of helper applications in your browser's preferences settings. A pair of data type names separated by a slash represents a MIME type (such as text/html and image/gif). When you specify a MIME type as a parameter to the document.open() method, you're instructing the browser about the kind of data it is about to receive, so that it knows how to render the data. Common values that most browsers accept are

```
text/html
text/plain
image/gif
image/jpeg
image/xbm
```

If you omit the parameter, JavaScript assumes the most popular type, text/html — the kind of data you typically assemble in a script prior to writing to the window. The text/html type includes any images that the HTML references. Specifying any of the image types means that you have the raw binary representation of the image that you want to appear in the new document — possible, but unlikely.

Another possibility is to direct the output of a write() method to a plug-in. For the mimeType parameter, specify the plug-in's MIME type (for example, application/x-director for Shockwave). Again, the data you write to a plug-in must be in a form that it knows how to handle. The same mechanism also works for writing data directly to a helper application.

IE accepts only the text/html MIME type parameter.

NN4+/Moz1+ and IE5+ include a second, optional parameter to the method: replace. This parameter does for the document.open() method what the replace() method does for the location object. For document.open(), it means that the new document you are about to write replaces the previous document in the window or frame from being recorded to that window or frame's history.

Example

You can see an example of where the document.open() method fits in the scheme of dynamically creating content for another frame in the discussion of the document.write() method later in this chapter.

Related Items: document.close(), document.clear(), document.write(), document.writeln() methods.

```
queryCommandEnabled("commandName")
queryCommandIndterm("commandName")
queryCommandCommandState("commandName")
queryCommandSupported("commandName")
queryCommandText("commandName")
queryCommandValue("commandName")
```

Returns: Various values.
Compatibility: WinIE4+, MacIE-, NN7.1, Moz1.4, Safari-

These six methods lend further support to the `execCommand()` method for `document` and `TextRange` objects in WinIE. If you choose to use the `execCommand()` method to achieve some stylistic change on a text selection, you can use some of these query methods to make sure the browser supports the desired command and to retrieve any returned values. Table 18-4 summarizes the purpose and returned values for each of the methods.

Table 18-4: IE Query Commands

queryCommand	*Returns*	*Description*
`Enabled`	Boolean	Reveals whether the `document` or `TextRange` object is in a suitable state to be invoked.
`Indterm`	Boolean	Reveals whether the command is in an indeterminate state.
`CommandState`	Boolean | null	Reveals whether the command has been completed (`true`), is still working (`false`), or is in an indeterminate state (`null`).
`Supported`	Boolean	Reveals whether the command is supported in the current browser.
`Text`	String	Returns any text that may be returned by a command.
`Value`	Varies	Returns whatever value (if any) is returned by a command.

Because the `execCommand()` method cannot be invoked on a page while it is still loading, any such invocations that may collide with the loading of a page should check with `queryCommandEnabled()` prior to invoking the command. Validating that the browser version running the script supports the desired command (especially for commands that have been introduced after IE4) is also a good idea. Therefore, you may want to wrap any command call with the following conditional structure:

```
if (document.queryCommandEnabled(commandName) &&
   document.queryCommandSupported(commandName)) {
   ...
}
```

When using a command to read information about a selection, use the `queryCommandText()` or `queryCommandValue()` methods to catch that information (recall that the `execCommand()` method itself returns a Boolean value regardless of the specific command invoked).

Example

See the examples for these methods covered under the `TextRange` object in Chapter 35.

Related Items: `TextRange` object (Chapter 35); `execCommand()` method.

`recalc([allFlag])`

Returns: Nothing.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

IE5 introduced the concept of dynamic properties. With the help of the `setExpression()` method of all elements and the `expression()` stylesheet value, you can establish dependencies between object properties and potentially dynamic properties, such as a window's size or a draggable element's location. After those dependencies are established, the `document.recalc()` method causes those dependencies to be recalculated — usually in response to some user action, such as resizing a window or dragging an element.

The optional parameter is a Boolean value. The default value, `false`, means that the recalculations are performed only on expressions for which the browser has detected any change since the last recalculation. If you specify `true`, however, all expressions are recalculated whether they have changed or not.

Mozilla 1.4 includes a feature that allows scripts to turn an `iframe` element's document object into an HTML editable document. Part of the scripting incorporates the `document.execCommand()` and related methods. Visit `http://www.mozilla.org/editor` for current details and examples.

Example

You can see an example of `recalc()` in Listing 15-32 for the `setExpression()` method. In that example, the dependencies are between the current time and properties of standard element objects.

Related Items: `getExpression()`, `removeExpression()`, `setExpression()` methods (Chapter 15).

`releaseEvents(eventTypeList)`

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

If your NN4 scripts have enabled event capture for the `document` object (or `window` or layer, for that matter), you can turn off that capture with the `releaseEvents()` method. This method does not inhibit events from reaching their intended target. In fact, by releasing capture from a higher object, released events don't bother stopping at those higher objects anymore.

See the discussion of the `releaseEvents()` method for the `window` object in Chapter 16. The behavior of the `releaseEvents()` method for all objects is the same.

Related Items: `document.captureEvents()`, `document.routeEvent()` methods.

routeEvent(*event*)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

If you turn on NN4 event capturing in the `window`, `document`, or layer object (via their respective `captureEvents()` methods), the event handler you assign to those events really captures those events, preventing them from ever reaching their intended targets. For some page designs, this capturing is intentional, because it allows the higher-level object to handle all events of a particular type. But if your goal is to perform some preprocessing of events before they reach their destination, you need a way to pass that event along its regular path, which is what the `routeEvent()` method is for.

See the discussion of the `routeEvent()` method for the `window` object in Chapter 16. The behavior of the `routeEvent()` method for all objects is the same.

Related Items: `document.captureEvents()`, `document.releaseEvents()` methods.

write("*string1*" [,"*string2*" ... [, "*stringn*"]])
writeln("*string1*" [,"*string2*" ... [, "*stringn*"]])

Returns: Boolean `true` if successful.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Both of these methods send text to a document for display in its window. The only difference between the two methods is that `document.writeln()` appends a carriage return to the end of the string it sends to the document. This carriage return is helpful for formatting source code when viewed through the browser's source view window. For new lines in rendered HTML that is generated by these methods, you must still write a `<br>` to insert a line break.

A common, incorrect conclusion that many JavaScript newcomers make is that these methods enable a script to modify the contents of an existing document, which is not true. As soon as a document has loaded into a window (or frame), the only fully backward-compatible text that you can modify without reloading or rewriting the entire page is the content of text and `textarea` objects. In IE4+, you can modify HTML and text via the `innerHTML`, `innerText`, `outerHTML`, and `outerText` properties of any element. For W3C DOM–compatible browsers, you can modify an element's text by setting its `nodeValue` or `innerHTML` properties; strict adherence to the W3C DOM requires creating and inserting or replacing new elements, as described in Chapter 15.

The two safest ways to use the `document.write()` and `document.writeln()` methods are to

- ✦ Write some or all of the page's content by way of scripts embedded in the document
- ✦ Send HTML code either to a new window or to a separate frame in a multiframe window

For the first case, you essentially interlace script segments within your HTML. The scripts run as the document loads, writing whatever scripted HTML content you like. This task is exactly what you did in `script1.htm` in Chapter 3. This task is also how you can have one page generate browser-specific HTML when a particular class of browser requires unique syntax.

In the latter case, a script can gather input from the user in one frame and then algorithmically determine the layout and content destined for another frame. The script assembles the HTML code for the other frame as a string variable (including all necessary HTML tags). Before the script can write anything to the frame, it can optionally open the layout stream

(to close the current document in that frame) with the `parent.frameName.document.open()` method. In the next step, a `parent.frameName.document.write()` method pours the entire string into the other frame. Finally, a `parent.frameName.document.close()` method ensures that the total data stream is written to the window. Such a frame looks just the same as if it were created by a source document on the server rather than on the fly in memory. The `document` object of that window or frame is a full citizen as a standard `document` object. You can, therefore, even include scripts as part of the HTML specification for one of these temporary HTML pages.

After an HTML document (containing a script that is going to write via the `write()` or `writeln()` methods) loads completely, the page's incoming stream closes automatically. If you then attempt to apply a series of `document.write()` statements, the first `document.write()` method completely removes all vestiges of the original document. That includes all of its objects and scripted variable values. Therefore, if you try to assemble a new page with a series of `document.write()` statements, the script and variables from the original page will be gone before the second `document.write()` statement executes. To get around this potential problem, assemble the content for the new screen of content as one string variable and then pass that variable as the parameter to a single `document.write()` statement. Also be sure to include a `document.close()` statement in the next line of script.

Assembling HTML in a script to be written via the `document.write()` method often requires skill in concatenating string values and nesting strings. A number of JavaScript `String` object shortcuts facilitate the formatting of text with HTML tags (see Chapter 27 for details).

If you are writing to a different frame or window, you are free to use multiple `document.write()` statements if you like. Whether your script sends lots of small strings via multiple `document.write()` methods or assembles a larger string to be sent via one `document.write()` method depends partly on the situation and partly on your own scripting style. From a performance standpoint, a fairly standard procedure is to do more preliminary work in memory and place as few I/O (input/output) calls as possible. On the other hand, making a difficult-to-track mistake is easier in string concatenation when you assemble longer strings. My personal preference is to assemble longer strings, but you should use the system that's most comfortable for you.

You may see another little-known way of passing parameters to these methods. Instead of concatenating string values with the plus (+) operator, you can also bring string values together by separating them with commas. For example, the following two statements produce the same results:

```
document.write("Today is " + new Date());
document.write("Today is ",new Date());
```

Neither form is better than the other, so use the one that feels more comfortable to your existing programming style.

Dynamically generating scripts requires an extra trick, especially in NN. The root of the problem is that if you try code, such as `document.write("<script></script>")`, the browser interprets the end script tag as the end of the script that is doing the writing. You have to trick the browser by separating the end tag into a couple of components. Escaping the forward slash also helps. For example, if you want to load a different `.js` file for each class of browser, the code looks similar to the following:

```
// variable 'browserVer' is a browser-specific string
// and 'page' is the HTML your script is accumulating
// for document.write()
page += "<script type='text/javascript' src='" +
   browseVer + ".js'><" + "\/script>";
```

Using the `document.open()`, `document.write()`, and `document.close()` methods to display images in a document requires some small extra steps. First, any URL assignments that you write via `document.write()` must be complete (not relative) URL references. Alternatively, you can write the `<base>` tag for the dynamically generated page so that its `href` attribute value matches that of the file that is writing the page.

The other image trick is to be sure to specify `height` and `width` attributes for every image, scripted or otherwise. Document-rendering performance is improved on all platforms, because the values help the browser lay out elements even before their details are loaded.

In addition to the `document.write()` example that follows (see Listings 18-16 through 18-18), you can find fuller implementations that use this method to assemble images and bar charts in many of the applications in Chapters 48 through 57 on the CD-ROM. Because you can assemble any valid HTML as a string to be written to a window or frame, a customized, on-the-fly document can be as elaborate as the most complex HTML document that you can imagine.

Example

The example in Listings 18-16 through 18-18 demonstrates several important points about using the `document.write()` or `document.writeln()` methods for writing to another frame. First is the fact that you can write any HTML code to a frame, and the browser accepts it as if the source code came from an HTML file somewhere. In the example, I assemble a complete HTML document, including basic HTML tags for completeness.

Listing 18-16: **A Frameset for the Document Writing Example**

```
<html>
   <head>
      <title>Writin' to the doc</title>
   </head>
   <frameset rows="50%,50%">
      <frame name="Frame1" src="lst18-17.htm" />
      <frame name="Frame2" src="lst18-18.htm" />
   </frameset>
</html>
```

Listing 18-17: **Writing a Document Based upon User Input**

```
<html>
   <head>
      <title>Document Write Controller</title>
      <script type="text/javascript">
      function takePulse(form) {
         var msg = "<html><head><title>On The Fly with " +
            form.yourName.value + "<\/title><\/head>";
         msg += "<body bgcolor='salmon'><h1>Good Day " + form.yourName.value +
            "!<\/h1><hr />";
         for (var i = 0; i < form.how.length; i++) {
            if (form.how[i].checked) {
               msg += form.how[i].value;
               break;
            }
```

Continued

Listing 18-17 *(continued)*

```
         }
  msg += "<br />Make it a great day!<\/body><\/html>";
         parent.Frame2.document.write(msg);
         parent.Frame2.document.close();
      }

      function getTitle() {
         alert("Lower frame document.title is now:" +
parent.Frame2.document.title);
      }
      </script>
   </head>
   <body>
      Fill in a name, and select how that person feels today. Then click "Write
      To Below" to see the results in the bottom frame.
      <form>
         Enter your first name:<input type="text" name="yourName"
         value="Dave" />
         <p>How are you today? <input type="radio" name="how"
            value="I hope that feeling continues forever."
            checked="checked" />Swell <input type="radio" name="how"
            value="You may be on your way to feeling Swell" />Pretty Good
            <input type="radio" name="how"
            value="Things can only get better from here." />So-So</p>
         <p><input type="button" name="enter" value="Write To Below"
            onclick="takePulse(this.form)" /></p>
         <hr />
         <input type="button" name="peek" value="Check Lower Frame Title"
         onclick="getTitle()" />
      </form>
   </body>
</html>
```

Listing 18-18: **A Placeholder Page for the Document Writing Example**

```
<html>
   <head>
      <title>Placeholder</title>
   </head>
   <body>
   </body>
</html>
```

It is important to note that this example customizes the content of the document based on user input. This customization makes the experience of working with your Web page feel far more interactive to the user — yet you're doing it without any CGI programs running on the server.

The second point I want to bring home is that the document created in the separate frame by the `document.write()` method is a genuine `document` object. In this example, for instance, the `<title>` tag of the written document changes if you redraw the lower frame after changing the entry of the name field in the upper frame. If you click the lower button after updating the bottom frame, you see that the `document.title` property has, indeed, changed to reflect the `<title>` tag written to the browser in the course of displaying the frame's page. The fact that you can artificially create full-fledged, JavaScript `document` objects on the fly represents one of the most important powers of serverless CGI scripting (for information delivery to the user) with JavaScript. You have much to take advantage of here if your imagination is up to the task.

Notice that you can easily modify Listing 18-17 to write the results to the same frame as the document containing the field and buttons. Instead of specifying the lower frame

```
parent.frames[1].document.open()
parent.frames[1].document.write(msg)
parent.frames[1].document.close()
```

the code simply can use

```
document.open()
document.write(msg)
document.close()
```

This code would replace the form document with the results and not require any frames in the first place. Because the code assembles all of the content for the new document into one variable value, that data survive the one `document.write()` method.

The frameset document (Listing 18-16) creates a blank frame by loading a blank document (Listing 18-18). An alternative I highly recommend is to have the framesetting document fill the frame with a blank document of its own creation. See the section titled "Blank frames" in Chapter 16 for further details about this technique for NN3+ and IE3+.

Related Items: `document.open()`; `document.close()`; `document.clear()` methods.

Event handlers

onselectionchange

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The `onselectionchange` event can be triggered by numerous user actions, although all of those actions occur on elements that are under the influence of the WinIE5.5 edit mode.

Related Items: `oncontrolselect` event handler.

onstop

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `onstop` event fires in WinIE5+ when the user clicks the browser's Stop button. Use this event handler to stop potentially runaway script execution on a page, because the Stop button does not otherwise control scripts after a page has loaded. If you are having a problem with a runaway repeat loop during development, you can temporarily use this event handler to let you stop the script for debugging.

Example

Listing 18-19 provides a simple example of an intentional infinitely looping script. In case you load this page into a browser other than IE5+, you can click the Halt Counter button to stop the looping. The Halt Counter button as well as the `onstop` event handler invokes the same function.

Listing 18-19: **Stopping a Script Using the onstop Event Handler**

```
<html>
   <head>
      <title>onStop Event Handler</title>
      <script type="text/javascript">
      var counter = 0;
      var timerID;
      function startCounter() {
         document.forms[0].display.value = ++counter;
         //clearTimeout(timerID)
         timerID = setTimeout("startCounter()", 10);
      }
      function haltCounter() {
         clearTimeout(timerID);
         counter = 0;
      }

      document.onstop = haltCounter;
      </script>
   </head>
   <body>
      <h1>onStop Event Handler</h1>
      <hr />
      <p>Click the browser's Stop button (in IE) to stop the script
      counter.</p>
      <form>
         <p><input type="text" name="display" /></p>
         <input type="button" value="Start Counter" onclick="startCounter()" />
         <input type="button" value="Halt Counter" onclick="haltCounter()" />
      </form>
   </body>
</html>
```

Related Items: Repeat loops (Chapter 31).

body Element Object

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	Methods	Event Handlers
alink	createControlRange()	onafterprint
background	createTextRange()	onbeforeprint
bgColor	doScroll()	onscroll
bgProperties		
bottomMargin		
leftMargin		
link		
noWrap		
rightMargin		
scroll		
scrollLeft		
scrollTop		
text		
topMargin		
vLink		

Syntax

Accessing `body` element object properties or methods:

```
[window.] document.body.property | method([parameters])
```

About this object

In object models that reveal HTML element objects, the `body` element object is the primary container of the content that visitors see on the page. The `body` contains all rendered HTML. This special place in the node hierarchy gives the `body` object some special powers, especially in the IE object model.

As if to signify the special relationship, both the IE and W3C object models provide the same shortcut reference to the `body` element: `document.body`. As a first-class HTML element object (as evidenced by the long lists of properties, methods, and event handlers covered in Chapter 15), you are also free to use other syntaxes to reach the `body` element.

You are certainly familiar with several `body` element attributes that govern body-wide content appearance, such as link colors (in three states) and background (color or image). But IE and NN/Mozilla (and the W3C so far) have some very different ideas about the `body` element's role in scripting documents. Many methods and properties that NN/Mozilla considers to be the domain of the window (for example, scrolling, inside window dimensions, and so forth), IE puts into the hands of the `body` element object. Therefore, while NN/Mozilla scrolls the window (and whatever it may contain), IE scrolls the body (inside whatever window it lives). And because the `body` element fills the entire viewable area of a browser window or frame, that

viewable rectangle is determined in IE by the body's `scrollHeight` and `scrollWidth` properties, whereas NN4+ features `window.innerHeight` and `window.innerWidth` properties. This distinction is important to point out because when you are scripting window- or document-wide appearance factors, you may have to look for properties and methods for the `window` or `body` element object, depending on your target browser(s).

Use caution when referencing the `document.body` object while the page is loading. The object may not officially exist until the page has completely loaded. If you need to set some initial properties via scripting, do so in response to the `onload` event handler located in the `<body>` tag. Attempts at setting `body` element object properties in immediate scripts inside the `head` element may result in error messages about the object not being found.

Properties

aLink
bgColor
link
text

vLink

Value: Hexadecimal triplet or color name string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `aLink`, `link`, and `vLink` properties are the new versions of the `document` properties `alinkColor`, `linkColor`, and `vlinkColor`. The `bgColor` is the same as the old `document.bgColor` property, while the `text` property is the new version of the `document.fgColor` property. These new properties are the scripted equivalents of the HTML attributes for the `body` element — the property names more closely align themselves with the HTML attributes than the old property names.

Link colors that are set via pseudo-class selectors in stylesheets (as `style` attributes of the `body` element) must be accessed via the `style` property for the `body` object. Over time, these properties will likely fall into disuse as stylesheets become more common.

Example

You can modify Listing 18-1 for use with IE4+ and NN6+/Moz1+/W3C only by using the new property names instead. Replace all references to the `document` properties with their `document.body` equivalents. For example, the function would be reworked as the following (changes in boldface):

```
function showColorValues() {
   var result = "";
   result += "bgColor: " + newWindow.document.body.bgColor + "\n";
   result += "vLink: " + newWindow.document.body.vLink + "\n";
   result += "link: " + newWindow.document.body.link + "\n";
   document.forms[0].results.value = result;
}
```

Related Items: `document.alinkColor`, `document.bgColor`, `document.fgColor`, `document.linkColor`, `document.vlinkColor` properties.

background

Value: URL string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The background property lets you set or get the URL for the background image (if any) assigned to the body element. A body element's background image overlays the background color in case both attributes or properties are set. To remove an image from the document's background, set the document.body.background property to an empty string.

Example

If you have a background image file named images/logoBG.gif, a script can set the background via the following statement:

```
document.body.background = "images/logoBG.gif";
```

To clear the background image:

```
document.body.background = "";
```

If a background color has been previously set, the color becomes visible after the image disappears.

Related Items: body.bgColor, body.bgProperties properties.

bgColor

See aLink

bgProperties

Value: String constant. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The IE-specific bgProperties property is an alternative way of adjusting whether the background image should remain fixed when the user scrolls the document or if it should scroll with the document. Initial settings for this behavior should be done via the background-attachment CSS attribute and modified under script control by way of the body element's style.backgroundAttachment property.

No matter which way you reference this property, the only allowable values are string constants scroll (the default) or fixed.

Example

Both of the following statements change the default behavior of background image scrolling in IE4+:

```
document.body.bgProperties = "fixed";
```

or

```
document.body.style.backgroundAttachment = "fixed";
```

The added benefit of using the stylesheet version is that it also works in NN6+.

Related Items: body.background property.

bottomMargin
leftMargin
rightMargin
topMargin

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The four IE-specific margin properties are alternatives to setting the corresponding four margin stylesheet attributes for the `body` element (`body.style.marginBottom`, and so on). Stylesheet margins represent blank space between the edge of an element's content and its next outermost container. In the case of the `body` element, that container is an invisible document container.

Of the four properties, only the one for the bottom margin may be confusing if the content does not fill the vertical space of a window or frame. The margin value is not automatically increased to accommodate the extra blank space.

Different versions and operating system implementations of IE4+ offer a variety of default integer values for these properties. But be aware that their values are not necessarily returned by these properties unless they are explicitly set in the IE-proprietary `body` element attributes of the same name. Therefore, even though a default `body` has a visible margin, the property does not return that default value except in MacIE, where the left and top margins may need to be factored into determining an inline element's position.

Example

Both of the following statements change the default left margin in IE4+:

```
document.body.leftMargin = 30;
```

or

```
document.body.style.marginLeft = 30;
```

Related Items: `style` object.

leftMargin

(See `bottomMargin`)

link

(See `aLink`)

noWrap

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `noWrap` property lets you modify the `body` element behavior normally set via the `nowrap` attribute. Because the property name is a negative, the Boolean logic needed to control it can get confusing.

The default behavior for a `body` element is for text to wrap within the width of the window or frame. This behavior occurs when the value of `noWrap` is its default value of `false`. By turning `noWrap` to `true`, a line of text continues to render past the right edge of the window or frame

until the HTML contains a line break (or end of paragraph). If the text continues on past the right edge of the window, the window (or frame) gains a horizontal scroll bar (of course, not if a frame is set to not scroll).

By and large, users don't like to scroll in any direction if they don't have to. Unless you have a special need to keep single lines intact, let the default behavior rule the day.

Example

To change the word-wrapping behavior from the default, the statement is:

```
document.body.noWrap = true;
```

Related Items: None.

rightMargin

(See `bottomMargin`)

scroll

Value: Constant string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The IE-specific `scroll` property provides scripted access to the IE-specific `scroll` attribute of a `body` element. By default, an IE `body` element displays a vertical scroll bar even if the height of the content does not warrant it; a horizontal scroll bar appears only when the content is forced to be wider than the window or frame. You can make sure that both scroll bars are hidden by setting the `scroll` attribute to "no" or changing it via script. Possible values for this property are the constant strings `yes` and `no`.

Other than `frame` attributes and NN4+ signed scripts, other browsers do not provide facilities for turning off scroll bars under script control. You can generate a new window (via the `window.open()` method) and specify that its scroll bars be hidden.

Example

To change the scroll bar appearance from the default, the statement is:

```
document.body.scroll = "no";
```

Related Items: `window.scrollbars` property; `window.open()` method.

scrollLeft

scrollTop

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN7+, Moz1+, Safari-

Even though the `scrollLeft` and `scrollTop` properties of the `body` object are the same as those for generic HTML element objects, they play an important roll in determining the position of positioned elements (described more fully in Chapter 39 on the CD-ROM). Because the mouse event and element position properties tend to be relative to the visible content region of the browser window, you must take the scrolling values of the `document.body` object into account when assigning an absolute position. Values for both of these properties are integers representing pixels.

Example

Listing 18-20 is the IE4+ (and NN7+/Moz1+) version of the original NN example for `pageXOffset` and `pageYOffset` properties (see Listing 16-13). Everything about these two examples is the same except for the syntax that retrieves the values indicating how much the document is scrolled in a window.

Listing 18-20: **Determining Scroll Values**

```
<html>
   <head>
      <title>Master of all Windows</title>
      <script type="text/javascript">
      function leftFrame() {
         var output = "<html><body><h3>Body Scroll Values<\/h3><hr />\n";
         output += "<form>body.scrollLeft:<input type='text' name='xOffset'
            size=4 /><br />\n";
         output += "body.scrollTop:<input type='text' name='yOffset'
            size=4 /><br />\n";
         output += "<\/form><\/body><\/html>";
         return output;
      }

      function rightFrame() {
         var output = "<html><head><script type='text/javascript'>\n";
         output += "function showOffsets() {\n";
         output += "parent.readout.document.forms[0].xOffset.value =
            document.body.scrollLeft\n";
         output += "parent.readout.document.forms[0].yOffset.value =
            document.body.scrollTop\n}\n";
         output += "document.onclick = showOffsets\n";
         output += "<\/script><\/head><body><h3>Content Page<\/h3>\n";
         output += "Scroll this frame and click on a table border to view page
            offset values.<br /><hr />\n";
         output += "<table border=5 width=800>";
         var oneRow = "<td>Cell 1<\/td><td>Cell 2<\/td><td>Cell 3<\/td><td>Cell
            4<\/td><td>Cell 5<\/td>";
         for (var i = 1; i <= 30; i++) {
            output += "<tr><td><b>Row " + i + "<\/b><\/td>" + oneRow +
               "<\/tr>";
         }
         output += "<\/table><\/body><\/html>";
         return output;
      }
      </script>
   </head>
   <frameset cols="30%,70%">
      <frame name="readout" src="javascript:parent.leftFrame()" />
      <frame name="display" src="javascript:parent.rightFrame()" />
   </frameset>
</html>
```

Related Items: `window.pageXOffset`, `window.pageYOffset` properties.

text

(See aLink)

topMargin

(See bottomMargin)

vLink

(See aLink)

Methods

createControlRange()

Returns: Array.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

This method creates a control range in WinIE5+ browsers. Control ranges are used for control-based selection, as opposed to text-based selection made possible by text ranges. The method only applies to documents in edit mode. In regular document view mode, the createControlRange() method returns an empty array.

createTextRange()

Returns: Object.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The body element object is the most common object to use to generate a TextRange object in IE4+, especially when the text you are about to manipulate is part of the document's body text. The initial TextRange object returned from the createTextRange() method encompasses the entire body element's HTML and body text. Further action on the returned object is required to set the start and end point of the range. See Chapter 35's discussion of the TextRange object for more details.

Example

See Listing 35-8 for an example of the createTextRange() method in action.

Related Items: TextRange object (Chapter 35).

doScroll(["*scrollAction*"])

Returns: Nothing.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

Use the doScroll() method to simulate user action on the scroll bars inside a window or frame that holds the current document. This method comes in handy if you are creating your own scroll bars in place of the standard system scroll bars. Scrolling is instantaneous, however, rather than with animation even if the Display control panel is set for animated scrolling. The parameter for this method is one of the string constant values shown in Table 18-5. In practice, occasionally the longer scroll action names more closely simulate an actual click on the scroll bar component, whereas the shortcut versions may scroll at a slightly different increment.

Table 18-5: document.body.doScroll() Parameters

Long Parameter	*Short Parameter*	*Scroll Action Simulates*
`scrollbarDown`	`down`	Clicking the down arrow.
`scrollbarHThumb`	n/a	Clicking the horizontal scroll bar thumb (no scrolling action).
`scrollbarLeft`	`left`	Clicking the left arrow.
`scrollbarPageDown`	`pageDown`	Clicking the page down area or pressing PgDn (default).
`scrollbarPageLeft`	`pageLeft`	Clicking the page left area.
`scrollbarPageRight`	`pageRight`	Clicking the page right area.
`scrollbarPageUp`	`pageUp`	Clicking the page up area or pressing PgUp.
`scrollbarVThumb`	n/a	Clicking the vertical scroll bar thumb (no scrolling action).

Unlike scrolling to a specific pixel location (by setting the `body` element's `scrollTop` and `scrollLeft` properties), the `doScroll()` method depends entirely on the spatial relationship between the body content and the window or frame size. Also, the `doScroll()` method triggers the `onscroll` event handler for the `body` element object.

Be aware that scripted modifications to body content can alter these spatial relationships. IE is prone to being sluggish in updating all of its internal dimensions after content has been altered. Should you attempt to invoke the `doScroll()` method after such a layout modification, the scroll may not be performed as expected. You may find the common trick of using `setTimeout()` to delay the invocation of the `doScroll()` method by a fraction of a second.

Example

Use The Evaluator (Chapter 13) to experiment with the `doScroll()` method in IE5+. Size the browser window so that at least the vertical scroll bar is active (meaning it has a thumb region). Enter the following statement into the top text field and press Enter a few times to simulate clicking the PgDn key:

```
document.body.doScroll()
```

Return to the top of the page and now do the same for scrolling by the increment of the scroll bar down arrow:

```
document.body.doScroll("down")
```

You can also experiment with upward scrolling. Enter the desired statement in the top text field and leave the text cursor in the field. Manually scroll to the bottom of the page and then press Enter to activate the command.

Related Items: `body.scroll`, `body.scrollTop`, `body.scrollLeft` properties; `window.scroll()`, `window.scrollBy()`, `window.scrollTo()` methods.

Event handlers

`onafterprint`
`onbeforeprint`

(See the `onafterprint` event handler for the `window` object, Chapter 16)

onscroll

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `onscroll` event handler fires for the `body` element object as the result of manual scrolling of the document (via scroll bars or navigation keyboard keys) and scripted scrolling via the `doScroll()` method, `scrollIntoView()` method, or adjusting the `scrollTop` and/or `scrollLeft` properties of the `body` element object. For manual scrolling and scrolling by `doScroll()`, the event seems to fire twice in succession. Moreover, the `event.srcElement` property is `null`, even when the `body` element is handling the `onscroll` event handler.

Example

Listing 18-21 is a highly artificial demonstration of what can be a useful tool for some page designs. Consider a document that occupies a window or frame, but one that you don't want scrolled, even by accident with one of the newer mouse wheels that are popular with Wintel PCs. If scrolling of the content would destroy the appearance or value of the content, then you want to make sure that the page always zips back to the top. The `onscroll` event handler in Listing 18-21 does just that. Notice that the event handler is set as a property of the `document.body` object after the page has loaded. Although the event handler can also be set as an attribute of the `<body>` tag, to assign it as a property requires the page to load first. Until then, the `document.body` object does not yet officially exist in the object model for this page.

Listing 18-21: Preventing a Page from Scrolling

```
<html>
   <head>
      <title>onscroll Event Handler</title>
      <script type="text/javascript">
      function zipBack() {
         window.scroll(0,0);
      }
      function init() {
         document.body.onscroll = zipBack;
      }
      </script>
   </head>
   <body onload="init()">
      <h1>onscroll Event Handler</h1>
      <hr />
      This page always zips back to the top if you try to scroll it.
      <p><iframe frameborder="0" scrolling="no" height="1000"
         src="bofright.htm"></iframe></p>
   </body>
</html>
```

Related Items: `body.scrollTop`, `body.scrollLeft` properties; `scrollIntoView()`, `body.doScroll()` methods.

TreeWalker Object

Property	Method	Event Handler
currentNode	firstChild()	(None)
expandEntityReference	lastChild()	
filter	nextNode()	
root	nextSibling()	
whatToShow	parentNode()	
	previousNode()	
	previousSibling()	

Syntax

Creating a TreeWalker object:

```
var treewalk = document.createTreeWalker(document, whatToShow, filterFunction,
entityRefExpansion);
```

Accessing TreeWalker object properties and methods:

```
TreeWalker.property | method([parameters])
```

Compatibility: WinIE-, MacIE-, NN7+, Moz1+, Safari-

About this object

The TreeWalker object serves as a container for a list of nodes that meet the criteria defined by the document.createTreeWalker() method, which is used to create the object. The list of nodes contained by a TreeWalker object conforms to the same hierarchical structure of the document from which they are referenced. The TreeWalker object provides a means of navigating through this list of nodes based upon their inherent tree-like structure.

You can think of the TreeWalker object as somewhat of an iterator object since its main purpose is to provide a means of stepping through nodes in a list. However, in this case the list is a hierarchical tree, as opposed to a linear list. The TreeWalker object maintains a pointer inside the list of nodes that always points to the "current node." Whenever you navigate through the list using the TreeWalker object, the navigation is always relative to the pointer. For example, referencing the previous or next node via calls to the previousNode() or nextNode() methods depends upon the current position of the node pointer in the tree.

Use the document.createTreeWalker() method to create a TreeWalker object for a particular document. This method requires a user function that serves as a filter for nodes selected to be part of the tree. A reference to the function is the third parameter of the method call. The return value of this user function can be one of three constant values, which indicate the status of the current node: NodeFilter.FILTER_ACCEPT, NodeFilter.FILTER_REJECT, or NodeFilter.FILTER_SKIP. The difference between NodeFilter.FILTER_REJECT and NodeFilter.FILTER_SKIP is that descendents of skipped nodes may still qualify as part of

the tree, while rejected nodes and their descendents are excluded altogether. Following is an example of a user function you could use to create a `TreeWalker` object:

```
function ratingAttrFilter(node) {
   if (node.hasAttribute("rating")) {
      return NodeFilter.FILTER_ACCEPT;
   }
   return NodeFilter.FILTER_REJECT;
}
```

In this example function, only nodes containing an attribute named `rating` are allowed through the filter, which means only those nodes will get added to the list (tree). With this function in place, you then call the `document.createTreeWalker()` method to create the `TreeWalker` object:

```
var myTreeWalker = document.createTreeWalker(document, NodeFilter.SHOW_ELEMENT,
   ratingAttrFilter, false);
```

Now that the `TreeWalker` object is created, you can use its properties and methods to access individual nodes and navigate through the list.

Properties

currentNode

Value: Node reference. Read/Write
Compatibility: WinIE-, MacIE-, NN7+, Moz1+, Safari-

The `currentNode` property returns a reference to the current node, which sits at the location of the tree's node pointer. Although you can use the `currentNode` property to access the current node, you can also use it to set the current node.

Example

To assign a node to the current position in the tree, just create an assignment statement using the `currentNode` property:

```
myTreeWalker.currentNode = document.getElementById("info");
```

Related Item: `root` property.

expandEntityReference
filter
root
whatToShow

Value: See text. Read-Only
Compatibility: WinIE-, MacIE-, NN7+, Moz1+, Safari-

These properties reflect the parameter values passed into the `document.createTreeWalker()` method upon the creation of the `TreeWalker` object.

Related Item: `document.createTreeWalker()` method.

Methods

firstChild()
lastChild()
nextSibling()
parentNode()
previousSibling()

Returns: Node reference.
Compatibility: WinIE-, MacIE-, NN7+, Moz1+, Safari-

These methods return references to nodes within the hierarchy of the tree-like list of nodes contained by the `TreeWalker` object. There is a parent-child relationship among all of the nodes in the tree, and these functions are used to obtain node references based upon this relationship. The node pointer within the tree moves to the new node whenever you use one of these methods to navigate to a given node. This means you can access the new node as the current node after calling one of these navigation methods.

Example

The following code shows how to obtain the tag name of the parent node of the current node in the `TreeWalker` object:

```
if (myTreeWalker.parentNode()) {
  var parentTag = myTreeWalker.currentNode.tagName;
}
```

Related Items: `nextNode()`, `previousNode()` methods.

nextNode()
previousNode()

Returns: Node reference.
Compatibility: WinIE-, MacIE-, NN7+, Moz1+, Safari-

The `nextNode()` and `previousNode()` methods navigate back and forth in the list of nodes contained by the `TreeWalker` object. It's important to note that these methods operate on the node list as if it has been flattened from a tree into a linear sequence of nodes. Both methods move the internal node pointer to the next or previous node, respectively.

Example

The following code demonstrates both the node filter function and a typical function you could use to display (in a series of alert windows, perhaps for debugging purposes) the IDs of all elements inside the body that have `id` attributes assigned. The `nextNode()` method is called first to advance the `TreeWalker`'s node pointer to the first node of the collection, and then iteratively (inside a `do-while` construction) to obtain the next node that passes the node filter's test.

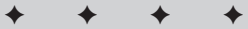
```
function idFilter(node) {
   if (node.hasAttribute("id")) {
      return NodeFilter.FILTER_ACCEPT;
   }
   return NodeFilter.FILTER_SKIP;
}

function showIds() {
   var tw =
   document.createTreeWalker(document.body, NodeFilter.SHOW_ELEMENT, idFilter,
false);
   // make sure TreeWalker contains at least one node, and go to it if true
   if (tw.nextNode()) {
      do {
         alert(tw.currentNode.id);
      } while (tw.nextNode());
   }
}
```

Related Items: `parentNode()` method.

✦ ✦ ✦

CHAPTER 19

Link and Anchor Objects

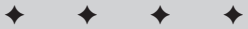


In This Chapter

Differences between link, anchor, and a element objects

Scripting a link to invoke a script function

Scripting a link to swap an image on mouse rollovers



The Web is based on the notion that the world's information can be strung together by way of the *hyperlink* — the clickable hunk of text or image that enables an inquisitive reader to navigate to a further explanation or related material. Of all the document objects you work with in JavaScript, the link is the one that makes that connection. Anchors also provide guideposts to specific locations within documents.

As scriptable objects going back to the first scriptable browsers, links and anchors are comparatively simple devices. But this simplicity belies their significance in the entire scheme of the Web. Under script control, links can be far more powerful than mere tethers to locations on the Web.

In modern browsers (IE4+ and W3C-compatible browsers), the notion of separating links and anchors as similar yet distinctly different objects begins to fade. The association of the word "link" with objects is potentially confused by the newer browsers' recognition of the `link` element (see Chapter 36 on the CD-ROM), which has an entirely different purpose, as a scriptable object. Taking the place of the anchor and link objects is an HTML element object representing the element created by the `<a>` tag. As an element object, the `a` element assumes all of the properties, methods, and event handlers that accrue to all HTML element objects in modern object models. To begin making that transition, this chapter treats all three types of objects at once. If you develop pages that must be compatible with early scriptable browsers, pay special attention to the comments about properties and event handler compatibility.

a

Anchor, Link, and a Element Objects

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
charset		
coords		
hash		
host		
hostname		
href		
hreflang		
Methods		
mimeType		
name		
nameProp		
pathname		
port		
protocol		
protocolLong		
rel		
rev		
search		
shape		
target		
text		
type		
urn		
x		
y		

Syntax

Accessing link object properties:

```
(all)      [window.]document.links[index].property
```

Accessing `a` element object properties:

```
 (IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

a

About this object

A little scripting history can help you to understand where the link and anchor objects came from and how the a element object evolved from them.

Using the terminology of the original object model, the anchor and link objects are both created in the object model from the <a> tag. What distinguishes a link from an anchor is the presence of the href attribute in the tag. Without an href attribute, the element is an anchor object, which (prior to version 4 browsers) has no properties, events, or event handlers associated with it. And even in NN4, the anchor object gains only four properties, all but one of which (name) disappear in NN6+/Moz1+.

A link, on the other hand, is much more alive as an object — all just because of the inclusion of an href attribute, which usually points to a URL to load into a window or frame. In fact, the majority of early object model properties for the link object are the same as those of the early location object — properties that reveal information about the URL assigned to the href attribute. The other vital part of the original link object — especially as distinct from an anchor object — is that a link can respond to events. Initially, event handlers were limited to just onclick and onmouseover. By NN4, additional mouse events and an ondblclick event joined the repertoire.

When object models treat HTML elements as objects, both the anchor and link objects are subsumed by the a element object. Even so, one important characteristic from the original object still holds true: all a element objects that behave as link objects (by virtue of the presence of an href attribute) are members of the document.links property array. Therefore, if your scripts need to inspect or modify properties of all link objects on a page, they can do so by way of a for loop through the array of link objects. This is true even if you script solely for modern browsers and want to, say, change a style attribute of all links (for example, changing their style.textDecoration property from none to underline). The fact that the same element can have different behaviors depending on the existence of one attribute makes me think of the a element object as potentially two different animals. Thus, you see references to link and anchor objects throughout this book when the distinction between the two is important.

Scripting newcomers are often confused about the purpose of the target attribute of an a element when they want a scripted link to act on a different frame or window. Under plain HTML, the target attribute points to the frame or window into which the new document (the one assigned to the href attribute) is to load, leaving the current window or frame intact. But if you intend to use event handlers to navigate (by setting the location.href property), the target attribute does not apply to the scripted action. Instead, assign the new URL to the location.href property of the desired frame or window. For example, if one frame contains a table of contents consisting entirely of links, the onclick event handlers of those links can load other pages into the main frame by assigning the URL to the parent.main.location.href property. You must also cancel the default behavior of any link, as described in the discussion of the generic onclick event handler in Chapter 15.

When you want a click of the link (whether the link consists of text or an image) to initiate an action without actually navigating to another URL, you can use a special technique — the javascript: pseudo-URL — to direct the URL to a JavaScript function. The URL javascript:*functionName*() is a valid parameter for the href attribute (and not just in the link object). Browsers that don't have JavaScript enabled do not respond to clicks on such a link.

a

If you don't want the link to do anything other than change the status bar in the `onmouseover` event handler, define an empty function and set the URL to that empty JavaScript function (such as `href="javascript:doNothing()"`). Starting with NN3 and IE4, you can also add a special `void` operator that guarantees that the called function does not trigger any true linking action (`href="javascript: void someFunction()"`). Specifying an empty string for the `href` attribute yields an FTP-like file listing for the client computer — an undesirable artifact. Don't forget, too, that if the URL leads to a type of file that initiates a browser helper application (for example, to play a RealAudio sound file) the helper app or plug-in loads and plays without changing the page in the browser window.

A single link can change the content of more than one frame at once with the help of JavaScript. If you want only JavaScript-enabled browsers to act on such links, use a `javascript:` pseudo-URL to invoke a function that changes the `location.href` properties of multiple frames. For example, consider the following function, which changes the content of two frames:

```
function navFrames(url1, url2) {
   parent.product.location.href = url1;
   parent.accessories.location.href = url2;
}
```

You can then have a `javascript:` pseudo-URL invoke this multipurpose function and pass the specifics for the link as parameters:

```
<a href="javascript: void navFrames('products/gizmo344.html',
'access/access344.html')">Deluxe Super Gizmo</a>
```

Or if you want one link to do something for everyone, but something extra for JavaScript-enabled browsers, you can combine the standard link behavior with an `onclick` event handler to take care of both situations:

```
function setAccessFrame(url) {
   parent.accessories.location.href = url;
}
...
<a href="products/gizmo344.html" target="product"
onclick="setAccessFrame('access/access344.html')">Deluxe Super Gizmo</a>
```

Notice here that the `target` attribute is necessary for the standard link behavior, while the script assigns a URL to a frame's `location.href` property.

One additional technique allows a single link tag to operate for both scriptable and nonscriptable browsers. For nonscriptable browsers, establish a genuine URL to navigate from the link. Then make sure that the link's `onclick` event handler evaluates to `return false`. At click time, a scriptable browser executes the event handler and ignores the `href` attribute; a nonscriptable browser ignores the event handler and follows the link. See the discussion of the generic `onclick` event handler in Chapter 15 for more details.

As you design your links, consider building `onmouseover` and `onmouseout` event handlers into your link definitions. The most common applications for these event handlers are as a means of adjusting the `window.status` property or swapping images. Thus, as a user rolls the mouse pointer atop a link, a descriptive label (perhaps more detailed or friendly than what the link text or image may indicate) appears in the status line at the bottom of the window. Whether a user notices the change down there is another issue, so don't rely on the status line as a medium for mission-critical communication. Image swaps, however, are more dramatic and enable a user to receive visual feedback that the mouse pointer is atop a particular button image. Thanks to the `onmousedown` event handler, you can even swap the image when the user presses down with the mouse button atop the link.

Properties

charset

Value: String. Read/Write
Compatibility: WinIE6+, MacIE6+, NN6+, Moz1+, Safari1+

The `charset` property represents the HTML 4.0 `charset` attribute of an `a` element. It advises the browser of the character set used by the document to which the `href` attribute points. The value is a string of one of the character set codes from the registry found at `http://www.iana.org/assignments/character-sets`.

coords
shape

Value: Strings. Read/Write
Compatibility: WinIE6+, MacIE6+, NN6+, Moz1+, Safari1+

HTML 4.0 provides specifications for `a` elements that accommodate different shapes (rect, circle, and poly) and coordinates when the link surrounds an image. Although the `coords` and `shape` properties are present for `a` element objects in all W3C DOM–compatible browsers, active support for the feature is not present in NN6.

hash
host
hostname
pathname
port
protocol
search

Value: Strings. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

This large set of properties is identical to the same-named properties of the `location` object (see Chapter 17). All properties are components of the URL that is assigned to the link object's `href` attribute. Although none of these properties appear in the W3C DOM specification for the `a` element object, they survive in modern browsers for backward compatibility. If you want to script the change of the destination for a link, try modifying the value of the object's `href` property rather than individual components of the URL.

Related Item: `location` object.

href

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `href` property (included in the W3C DOM) is the URL of the destination of an `a` element equipped to act as a link. URLs can be relative or absolute.

In W3C DOM–compatible browsers, you can turn an anchor object into a link object by assigning a value to the `href` property even if the `a` element has no `href` attribute in the HTML that loads from the server. Naturally, this conversion is temporary, and it lasts only as long as the page is loaded in the browser. When you assign a value to the `href` property of an `a` element that surrounds text, the text assumes the appearance of a link (either the default appearance or whatever style you assign to links).

Related Item: `location` object.

hreflang

Value: String. Read/Write
Compatibility: WinIE6+, MacIE6+, NN6+, Moz1+, Safari1+

The `hreflang` property advises the browser (if the browser takes advantage of it) about the written language used for the content to which the `a` element's `href` attribute points. Values for this property must be in the form of the standard language codes (for example, `en-us` for U.S. English).

Methods

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `Methods` property (note the uppercase "M") represents the HTML 4.0 `methods` attribute for an `a` element. Values for this attribute and property serve as advisory instructions to the browser about which HTTP method(s) to use for accessing the destination document. This is a rare case in which an HTML 4.0 attribute is not echoed in the W3C DOM. In any case, while IE4+ supports the property, the IE browsers do nothing special with the information.

mimeType

Value: String. Read-Only
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

This property is used to obtain the MIME type of the document linked to by the `a` element. The HTML 4.0 and W3C DOM specifications define a `type` attribute and `type` property instead of `mimeType`. The property is a read-only property, and therefore has no control over the MIME type of the destination document.

Related Item: `a.type` property.

name

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Although a `name` attribute is optional for an `a` element serving solely as a link object, it is required for an anchor object. This value is exposed to scripting via the `name` property. While it is unlikely you will need to change the value by scripting, you can use this property as a way to identify a link object from among the `document.links` arrays in a repeat loop. For example:

```
for (var i = 0; i < document.links.length; i++) {
   if (document.links[i].name == "bottom" {
      // statements dealing with the link named "bottom"
   }
}
```

nameProp

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The IE-specific `nameProp` property is a convenience property that retrieves the segment of the `href` to the right of the rightmost forward slash character of the URL. Most typically, this value is the name of the file from a URL. But if the URL also includes a port number, that number is returned as part of the `nameProp` value.

protocolLong

Value: String. Read-Only
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The IE-specific `protocolLong` property returns a verbose rendition of the `protocol` property as indicated in the `a` element's `href` attribute. For example, if the `href` attribute points to an `http:` protocol, the `protocolLong` property returns `HyperText Transfer Protocol`. Introduced in IE4, the `protocolLong` property is still present in IE5 but apparently is no longer supported.

rel
rev

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `rel` and `rev` properties define relationships in the forward and back directions with respect to the destination document of the `a` element. Browsers have yet to exploit most of the potential of these attributes and properties.

A long list of values are predefined for these properties, based on the corresponding attribute values specified in HTML 4.0. If the browser does nothing with a particular value, the value is ignored. You can string together multiple values in a space-delimited list inside a single string. Accepted values are as follows:

`alternate`	`contents`	`index`	`start`
`appendix`	`copyright`	`next`	`stylesheet`
`bookmark`	`glossary`	`prev`	`subsection`
`chapter`	`help`	`section`	

target

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

An important property of the link object is the `target`. This value reflects the window name supplied to the `target` attribute in the `a` element.

You can temporarily change the target for a link. But, as with most transient object properties, the setting does not survive soft reloads. Rather than altering the target this way, you can safely force the target change by letting the `href` attribute call a `javascript:`*`functionName`*`()` psuedo-URL in which the function assigns a document to the desired `window.location`. If you have done extensive HTML authoring before, you will find it hard to break the habit of relying on the `target` attribute.

Further, if you develop XHTML pages that must validate with the strict DTD, you will not be able to include a `target` attribute in your `<a>` tags. Instead, use the page's `onload` event handler or the `a` element's `onclick` event handler to invoke a function that assigns the desired value to the `target` property.

Related Item: `document.links` property.

text

Value: String. Read-Only
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

Between the start and end tags of a link goes the text (or image) that is highlighted in the distinguishing link color of the document. Navigator 4 enables you to read that text with the `link.text` property. This property is read-only. For later browsers, use the IE4+ and/or W3C DOM syntax for reading the text node (`innerText`, `innerHTML`, or `nodeValue`) property of the `a` element.

This property was not implemented in releases of Navigator 4 prior to version 4.02.

type

Value: String. Read/Write
Compatibility: WinIE6+, MacIE6+, NN6+, Moz1+, Safari1+

The `type` property represents the HTML 4.0 `type` attribute, which specifies the MIME type for the content of the destination document to which the element's `href` attribute points. This is primarily an advisory property for browsers that wish to, say, display different cursor styles based on the anticipated type of content at the other end of the link. Thus far, browsers do not take advantage of this feature. However, you can assign MIME type values to the attribute (for example, `video/mpeg`) and let scripts read those values for making style changes to the link text after the page loads. IE4+ also implements a similar property in the `mimeType` property.

Related Item: `a.mimeType` property.

urn

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `urn` property represents the IE-specific `URN` attribute, which enables authors to use a URN (Uniform Resource Name) for the destination of the `a` element. (See `http://www.ietf.org/rfc/rfc2141.txt` for information about URNs.) This property is not in common use.

x
y

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

Your Navigator 4 script can retrieve the x and y coordinates of a link object (the top-left corner of the rectangular space occupied by the linked text or image) via the `link.x` and `link.y` properties. With IE4+ and W3C DOM-compatible browsers, you can get the coordinates of a typical link via the `a` element's `offsetLeft` and `offsetTop` properties.

✦ ✦ ✦

CHAPTER 20

Image, Area, and Map Objects

In This Chapter

How to precache images

Swapping images after a document loads

Creating interactive, client-side image maps

For modern Web browsers, images and areas — those items created by the `<img>` and `<area>` tags — are first-class objects that you can script for enhanced interactivity. You can swap the image displayed in an `<img>` tag with other images, perhaps to show the highlighting of an icon button when the cursor rolls atop it. And with scriptable client-side area maps, pages can be smarter about how they respond to users' clicks on image regions.

One further benefit afforded scripters is that they can preload images into the browser's image cache as the page loads. With cached images, the user experiences no delay when the first swap occurs.

Image and img Element Objects

For HTML element properties, methods, and event handlers, see Chapter 15.

img

Properties	*Methods*	*Event Handlers*
align		onabort
alt		onerror
border		onload
complete		
dynsrc		
fileCreatedDate		
fileModifiedDate		
fileSize		
fileUpdatedDate		
height		
href		
hspace		
isMap		
loop		
longDesc		
lowsrc		
mimeType		
name		
nameProp		
naturalHeight		
naturalWidth		
protocol		
src		
start		
useMap		
vspace		
width		
x		
y		

Syntax

Creating an `Image` object:

```
imageName = new Image([pixelWidth, pixelHeight]);
```

Accessing `img` element and `image` object properties or methods:

```
(NN3+/IE4+)   [window.]document.imageName. property | method([parameters])
(NN3+/IE4+)   [window.]document.images[index]. property | method([parameters])
(NN3+/IE4+)   [window.]document.images["imageName"]. property |
method([parameters])
(IE4+)        [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)    [window.]document.getElementById("elemID").property |
              method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

About this object

Before getting into detail about images as objects, it's important to understand the distinction between instances of the static `Image` object and `img` element objects. The former exist only in the browser's memory without showing anything to the user; the latter are the elements on the page generated via the `<img>` (or nonsanctioned, but accepted, `<image>`) tag. Scripts use `Image` objects to precache images for a page, but `Image` objects obviously have fewer applicable properties, methods, and event handlers because they are neither visible on the page nor influenced by tag attributes.

`img` elements have been in the HTML vocabulary since the earliest days, but Netscape Navigator 3 was the first to treat them like first-class objects along with the companion `Image` object for precaching images. All flavors of IE4+ treat `img` elements as true document objects. The primary advantage of treating `img` elements as objects is that scripts can change the image that occupies the `img` object's space on the page, even after the document has loaded and displayed an initial image. The key to this scriptability is the `src` property of an image.

In a typical scenario, a page loads with an initial image. That image's tags specify any of the extra attributes, such as `height` and `width` (which help speed the rendering of the page), and specify whether the image uses a client-side image map to make it interactive. (See the `area` object later in this chapter.) As the user spends time on the page, the image can then change (perhaps in response to user action or some timed event in the script), replacing the original image with a new one in the same space. In browsers prior to IE4 and NN6 that support the `img` element object, the height and width of the initial image that loads into the element establishes a fixed-sized rectangular space for the image. Attempts to fit an image of another size into that space forces the image to scale (up or down, as the case may be) to fit the rectangle. But in IE4+ and W3C-compatible browsers, a change in the image's size is reflected by an automatic reflow of the page content around the different size.

The benefit of the separate `Image` object is that a script can create a virtual image to hold a preloaded image. (The image is loaded into the image cache but the browser does not display the image.) The hope is that one or more unseen images will load into memory while the user is busy reading the page or waiting for the page to download. Then, in response to user action on the page, an image can change instantaneously rather than forcing the user to wait for the image to load on demand.

To preload an image, begin by assigning a new, empty image object to a global variable. The new image is created via the constructor function available to the `Image` object:

```
var imageVariable = new Image(width, height);
```

You help the browser allocate memory for the image if you provide the pixel height and width of the precached image as parameters to the constructor function. All that this statement does is create an object in memory whose properties are all empty. To force the browser to load the image into the cache, assign an image file URL to the object's `src` property:

```
var oneImage = new Image(55,68);
oneImage.src = "neatImage.gif";
```

As this image loads, you see the progress in the status bar just like any image. Later, assign the `src` property of this stored image to the `src` property of the `img` element object that appears on the page:

```
document.images["someImage"].src = oneImage.src;
```

Depending on the type and size of image, you will be amazed at the speedy response of this kind of loading. With small-palette graphics, the image displays instantaneously.

A popular user-interface technique is to change the appearance of an image that represents a clickable button when the user rolls the mouse pointer atop that art. This action assumes that a mouse event fires on an element associated with the object. Prior to IE4 and NN6, `img` element objects did not respond to mouse events on their own. The required technique was to encase the `img` element inside an `a` element. This allowed the events associated with rollovers (`onmouseover` and `onmouseout`) and a user click on the image to effect some change (usually to navigate to another page). Although IE4+, NN6+, and other W3C browsers provide these events directly for `img` element objects, you can guarantee your pages to be backward compatible if you continue to surround your images with `a` elements. You can see examples of these kinds of actions in Chapters 12 and 20.

Image rollovers are most commonly accomplished in two different image states: normal and highlighted. But you may want to increase the number of states to more closely simulate the way clickable buttons work in application programs. In some instances, a third state signifies that the button is switched on. For example, if you use rollovers in a frame for navigational purposes and the user clicks a button to navigate to the Products area, that button stays selected but in a different style than the rollover highlights. Some designers go one step further by providing a fourth state that appears briefly when the user mouses down an image. Each one of these states requires the download of yet another image, so you have to gauge the effect of the results against the delay in loading the page.

The speed with which image swapping takes place may lead you to consider using this approach for animation. Though this approach may be practical for brief bursts of animation, the many other ways of introducing animation to your Web page (such as via GIF89a-standard images, Flash animations, Java applets, and a variety of plug-ins) produce animation that offers better speed control. In fact, swapping preloaded JavaScript image objects for some cartoon-like animations may be too fast. You can build a delay mechanism around the `setInterval()` method, but the precise timing between frames varies with client processor performance.

All browsers that implement the `img` element object also implement the `document.images` array. You can (and should) use the availability of this array as a conditional switch before any script statements that work with the `img` element or `Image` object. The construction to use is as follows:

```
if (document.images) {
   // statements working with images as objects
}
```

Earlier browsers treat the absence of this array as the equivalent of `false` in the `if` clause's conditional statement.

Most of the properties discussed here mirror attributes of the `img` HTML element. For more details on the meanings and implications of attribute values on the rendered content, consult the HTML 4.01 specification (`http://www.w3.org/TR/REC-html401`) and Microsoft's extensions for IE (`http://msdn.microsoft.com/workshop/author/dhtml/reference/objects/img.asp`).

Properties

`align`

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `align` property defines how the image is oriented in relation to surrounding text content. It is a double-duty property because you can use it to control the vertical or horizontal alignment

depending on the value (and whether the image is influenced by a `float` style attribute). Values are string constants, as follows:

```
absbottom    middle
absmiddle    right
baseline     texttop
bottom       top
left
```

The default alignment for an image is `bottom`. Safari 1.0 responds only to `left` and `right`. Increasingly, element alignment is handed over to stylesheet control.

Listing 20-1 enables you to choose from the different `align` property values as they influence the layout of an image whose HTML is embedded inline with some other text. Resize the window to see different perspectives on word-wrapping on a page and their effects on the alignment choices. Not all browsers provide distinctive alignments for each choice, so experiment in multiple supported browsers.

Listing 20-1: **Testing an Image's align Property**

```
<html>
   <head>
      <title>img align Property</title>
      <script type="text/javascript">
      function setAlignment(sel) {
         document.getElementById("myIMG").align =
            sel.options[sel.selectedIndex].value;
      }
      </script>
   </head>
   <body>
      <h1>img align Property</h1>
      <hr />
      <form>
         Choose the image alignment: <select onchange="setAlignment(this)">
            <option value="absbottom">absbottom</option>
            <option value="absmiddle">absmiddle</option>
            <option value="baseline">baseline</option>
            <option value="bottom" selected="selected">bottom</option>
            <option value="left">left</option>
            <option value="middle">middle</option>
            <option value="right">right</option>
            <option value="texttop">texttop</option>
            <option value="top">top</option>
         </select>
      </form>
      <hr />
      <p>Lorem ipsum dolor sit amet, consectetaur adipisicing elit, sed do
         eiusmod tempor incididunt ut labore et dolore magna aliqua. <img
         alt="image" id="myIMG" src="desk1.gif" height="90" width="120" /> Ut
         enim adminim veniam, quis nostrud exercitation ullamco laboris nisi ut
         aliquip ex ea commodo consequat.</p>
   </body>
</html>
```

Related Items: `text-align`, `float` stylesheet attributes.

alt

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `alt` property enables you to set or modify the text that the browser displays in the image's rectangular space (if height and width are specified in the tag) before the image downloads to the client. Also, if a browser has images turned off (or is incapable of displaying images), the `alt` text helps users identify what is normally displayed in that space. You can modify this `alt` text even after the page loads.

Example

Use The Evaluator (Chapter 13) to assign a string to the `alt` property of the `document.myIMG` image on the page. First, assign a nonexistent image to the `src` property to remove the existing image:

```
document.myIMG.src = "fred.gif"
```

Scroll down to the image, and you can see a space for the image. Now, assign a string to the `alt` property:

```
document.myIMG.src = "Fred\'s face"
```

The extra backslash is required to escape the apostrophe inside the string. Scroll down to see the new `alt` text in the image space.

Related Item: `title` property.

border

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

The `border` property defines the thickness in pixels of a border around an image. Remember that if you wrap an image inside an `a` element to make use of the mouse events (for rollovers and such), be sure to set the `border=0` attribute of the `<img>` tag to prevent the browser from generating the usual link kind of border around the image. Even though the default value of the attribute is zero, surrounding the image with an `a` element or attaching the image to a client-side image map puts a border around the image.

Example

Feel free to experiment with the `document.myIMG.border` property for the image in The Evaluator (Chapter 13) by assigning different integer values to the property.

Related Items: `isMap`, `useMap` properties.

complete

Value: Boolean. Read-Only
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari-

Sometimes you may want to make sure that an image is not still in the process of loading before allowing another process to take place. This situation is different from waiting for an image to load before triggering some other process (which you can do via the image object's `onload` event handler). To verify that the `img` object displays a completed image, check for the Boolean value of the `complete` property. To verify that a particular image file has loaded, first find out whether the `complete` property is `true`; then compare the `src` property against the desired filename.

An image's `complete` property switches to `true` even if only the specified `lowsrc` image has finished loading. Do not rely on this property alone for determining whether the `src` image has loaded if both `src` and `lowsrc` attributes are specified in the `<img>` tag.

One of the best ways to use this property is in an `if` construction's conditional statement:

```
if (document.myImage.complete) {
    // statements that work with document.myImage
}
```

The `complete` property is not reliable in Navigator 4 and some versions of Internet Explorer 4. For those browsers, the value returns `true` in all instances.

To experiment with the `image.complete` property, quit and relaunch your browser before loading Listing 20-2 (in case the images are in memory cache). As each image loads, click the "Is it loaded yet?" button to see the status of the `complete` property for the `image` object. The value is `false` until the loading finishes; then, the value becomes `true`. The arch image is the bigger of the two image files. You may have to quit and relaunch your browser between trials to clear the arch image from the cache (or empty the browser's memory cache). If you experience difficulty with this property in your scripts, try adding an `onload` event handler (even if it is empty, as in Listing 20-2) to your `<img>` tag.

Listing 20-2: **Scripting image.complete**

```
<html>
   <head>
      <title></title>
      <script type="text/javascript">
      function loadIt(theImage,form) {
         form.result.value = "";
         document.images[0].src = theImage;
      }
      function checkLoad(form) {
         form.result.value = document.images[0].complete;
      }
      </script>
   </head>
   <body>
      <img alt="image" src="cpu2.gif" width="120" height="90" onload="" />
      <form>
         <input type="button" value="Load keyboard"
         onclick="loadIt('cpu2.gif',this.form)" /> <input type="button"
         value="Load arch" onclick="loadIt('arch.gif',this.form)" />
         <p><input type="button" value="Is it loaded yet?"
            onclick="checkLoad(this.form)" /> <input type="text"
            name="result" /></p>
      </form>
   </body>
</html>
```

Related Items: `img.src`, `img.lowsrc`, `img.readyState` properties; `onload` event handler.

img.dynsrc

dynsrc

Value: URL string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `dynsrc` property is a URL to a video source file, which (in IE) you can play through an `img` element. You can turn a space devoted to a static image into a video viewer by assigning a URL of a valid video source (for example, an `.avi` or `.mpg` file) to the `dynsrc` property of the image element object. Unlike the `src` property of image objects, assigning a URL to the `dynsrc` property does not precache the video.

You may experience buggy behavior in various IE versions when you assign a value to an image's `dynsrc` property after the `img` element renders a `.gif` or `.jpg` image. In WinIE5, the status bar indicates that the video file is still downloading, even though the download is complete. Clicking the Stop button has no effect. WinIE5.5+ may not even load the video file, leaving a blank space on the page. MacIE5 changes between static and motion images with no problems, but playing the video file multiple times causes the `img` element to display black space beyond the element's rectangle. You can experience all this behavior in the example provided in Listing 20-3. None of these bugs is fatal, but they should discourage you from using the `img` element as a vehicle for video content.

To swap between still and video sources, simply empty the opposite property. Listing 20-3 shows a simplified example that swaps between one fixed image and one video image. This listing exhibits most of the bugs associated with changing between static image and video sources described a moment ago.

Listing 20-3: **Changing Between Still and Motion Images**

```
<html>
   <head>
      <title>img dynsrc Property</title>
      <script type="text/javascript">
      var trainImg = new Image(160,120);
      trainImg.src = "amtrak.jpg";
      trainImg.dynsrc = "amtrak.mpg";

      function setLoop() {
         var selector = document.forms[0].looper;
         document.getElementById("myIMG").loop =
            selector.options[selector.selectedIndex].value;
      }

      function setImage(type) {
         if (type == "jpg") {
            document.getElementById("myIMG").dynsrc = "";
            document.getElementById("myIMG").src = trainImg.src;
         } else {
            document.getElementById("myIMG").src = "";
            document.getElementById("myIMG").start = "fileopen";
            setLoop();
            document.getElementById("myIMG").dynsrc = trainImg.dynsrc;
         }
      }
      </script>
   </head>
```

```
   <body>
      <h1>img dynsrc Property</h1>
      <hr />
      <form>
         Choose image type: <input type="radio" name="imgGroup"
         checked="checked" onclick="setImage('jpg')" />Still <input
         type="radio" name="imgGroup" onclick="setImage('mpg')" />Video
         <p>Play video how many times after loading: <select name="looper"
            onchange="setLoop()">
               <option value="1" selected="selected">Once</option>
               <option value="2">Twice</option>
               <option value="-1">Continuously</option>
            </select></p>
      </form>
      <hr />
      <img alt="image" id="myIMG" src="amtrak.jpg" height="120"
      width="160" />
   </body>
</html>
```

Related Items: `img.loop`, `img.start` properties.

fileCreatedDate
fileModifiedDate
fileUpdatedDate
fileSize

Value: String, Integer (`fileSize`). Read-Only
Compatibility: WinIE4+, MacIE5+, NN-, Moz-, Safari-

These four IE-specific properties return information about the file displayed in the `img` element (whether still or motion image). Three of the properties reveal the dates on which the current image's file was created, modified, and updated. For an unmodified file, its creation and modified dates are the same. The updated date of an image is the date on which the image file was last uploaded to the server; the `fileUpdatedDate` property is only supported on WinIE5.5+ and MacIE5. The `fileSize` property reveals the number of bytes of the file.

Date values returned for the first two properties are formatted differently between IE4 and IE5. The former provides a full readout of the day and date; the latter returns a format similar to mm/dd/yyyy. Note, however, that the values contain only the date and not the time. In any case, you can use the values as the parameter to a `new Date()` constructor function. This enables you to then use date calculations for such information as the number of days between the current day and the most recent modification.

Not all servers provide the proper date or size information about a file or in a format that IE can interpret. Test your implementation on the deployment server to ensure compatibility.

Also, be aware that these properties can be read-only for a file that is loaded in the browser. JavaScript by itself cannot get this information about files on the server that are not loaded in the browser.

All of these file-related properties are present in the Mac version of IE, but the values are empty.

Example

These properties are similar to the same-named properties of the `document` object. You can see these properties in action in Listing 18-4. Make a copy of that listing, and supply an image before modifying the references from the `document` object to the image object to see how these properties work with the `img` element object.

Related Items: None.

height
width

Value: Integer. Read/Write (see text)
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

The `height` and `width` properties return and (in later browsers) control the pixel height and width of an image object. The property is read-only in NN3 and NN4, but it is read/write in all newer browsers that support the `img` element object.

If you adjust the `height` property of an image, the browser automatically scales the image within the same proportions as the original. But adjusting the `width` property has no effect on the `height` property in most browser versions. Scaling of an image may cause unwanted pixelation in the image, so modify an image's size with extreme care.

Example

Use The Evaluator (Chapter 13) to experiment with the `height` and `width` properties. Begin retrieving the default values by entering the following two statements into the top text box:

```
document.myIMG.height
document.myIMG.width
```

Increase the height of the image from its default 90 to 180:

```
document.myIMG.height = 180
```

If you scroll down to the image, you see that the image has scaled in proportion. Next, exaggerate the width:

```
document.myIMG.width = 400
```

View the resulting image.

Related Items: `hspace`, `vspace` properties.

href

(See `src` property)

hspace
vspace

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

The `hspace` and `vspace` properties control the pixel width of a transparent margin surrounding an image. Specifically, `hspace` controls the margins at the left and right of the image; `vspace` controls the top and bottom margins. Images, by default, have margins of zero pixels.

Example

Use The Evaluator (Chapter 13) to experiment with the `hspace` and `vspace` properties. Begin by noticing that the image near the bottom of the page has no margins specified for it and is flush left with the page. Now assign a horizontal margin spacing of 30 pixels:

```
document.myIMG.hspace = 30
```

The image has shifted to the right by 30 pixels. An invisible margin also exists to the right of the image.

Related Items: `height`, `width` properties.

isMap

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `isMap` property enables you to set whether the image should act as a server-side image map. When set as a server-side image map, pixel coordinates of the click are passed as parameters to whatever link `href` surrounds the image. For client-side image maps, see the `useMap` property later in this chapter.

Example

The image in The Evaluator page is not defined as an image map. Thus, if you type the following statement into the top text box, the property returns `false`:

```
document.myIMG.isMap
```

Related Item: `img.useMap` property.

longDesc

Value: URL string. Read/Write
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The `longDesc` property is a URL of a file that is intended to provide a detailed description of the image associated with the `img` element. Version 6 browsers recognize this property, but do not do anything special with the information — whether specified by script or the `longdesc` attribute.

Related Item: `alt` property.

loop

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `loop` property represents the number of times a video clip playing through the `img` element object should run. After the video plays that number of times, only the first frame of the video appears in the image area. The default value is `1`; but if you set the value to `-1`, the video plays continuously. Unfortunately, setting the property to `0` prior to assigning a URL to the `dynsrc` property does not prevent the movie from playing at least once (except on the Mac, as noted in the `dynsrc` property discussion earlier in this chapter).

Example

See Listing 20-3 for the `dynsrc` property to see the `loop` property in action.

Related Item: `dynsrc` property.

lowsrc
lowSrc

Value: URL string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari-

For image files that take several seconds to load, recent browsers enable you to specify a lower-resolution image or some other quick-loading placeholder to stand in while the big image crawls to the browser. You assign this alternate image via the `lowsrc` attribute in the `<img>` tag. The attribute is reflected in the `lowsrc` property of an image object.

All compatible browsers recognize the all-lowercase version of this property. NN6 also recognizes an interCap "S" version of the property, `lowSrc`.

Be aware that if you assign a URL to the `lowsrc` attribute, the `complete` property switches to `true` and the `onLoad` event handler fires when the alternate file finishes loading: The browser does not wait for the main `src` file to load.

Example

See Listing 20-5 for the image object's `onload` event handler to see how the source-related properties affect event processing.

Related Items: `img.src`, `img.complete` properties.

mimeType

Value: String. Read-Only
Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-

The `mimeType` property returns a plain-language description of the MIME type for the image, such as `JPEG Image` or `GIF Image`.

Example

You can use the `mimeType` property to determine the format of an image, as the following example demonstrates:

```
if (document.myIMG.mimeType.indexOf("JPEG") != -1) {
   // Carry out JPEG-specific processing
}
```

In this example, the `indexOf()` method is used to check for the presence of the phrase `"JPEG"` anywhere in the MIME type string. This works because the string returned in the `mimeType` property for JPEG images is `"JPEG Image"`.

Related Items: None.

name

Value: Identifier string. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `name` property returns the value assigned to the `name` attribute of an `img` element. Starting with IE4 and NN6, you can use the ID of the element (`id` attribute) to reference the `img` element object via `document.all` and `document.getElementById()`. But references in the form of `document.`*`imageName`* and `document.images[`*`imageName`*`]` must use only the value assigned to the `name` attribute.

In some designs, it may be convenient to assign numerically sequenced names to `img` elements, such as `img1`, `img2`, and so on. As with any scriptable identifier, the name cannot begin with a numeric character. Rarely, if ever, will you need to change the name of an `img` element object.

Example

You can use The Evaluator (Chapter 13) to examine the value returned by the `name` property of the image on that page. Enter the following statement into the top text box:

```
document.myIMG.name
```

Of course, this is redundant because the name is part of the reference to the object.

Related Item: `id` property.

nameProp

Value: Filename string. Read-Only
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

Unlike the `src` property, which returns a complete URL in IE, the IE `nameProp` property returns only the filename exclusive of protocol and path. If your image-swapping script needs to read the name of the file currently assigned to the image (to determine which image to show next), the `nameProp` property makes it easier to get the actual filename without having to perform extensive parsing of the URL.

Example

You can use The Evaluator Sr. (Chapter 13) to compare the results of the `src` and `nameProp` properties in WinIE5+. Enter each of the following statements into the top text box:

```
document.myIMG.src
document.myIMG.nameProp
```

Related Item: `img.src` property.

naturalHeight
naturalWidth

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `naturalHeight` and `naturalWidth` properties return the unscaled height and width of the image, in pixels. These properties are useful in situations where script code or `img` element attributes have scaled an image and you wish to know the image's original size.

Example

Use The Evaluator (Chapter 13) to experiment with the `naturalHeight` and `naturalWidth` properties. Begin retrieving the default values by entering the following statement into the top text box:

```
document.myIMG.width
```

Increase the width of the image from its default 120 to 200:

```
document.myIMG.width = 200
```

If you scroll down to the image, you see that the image has scaled in proportion. You can now find out the natural width of the original image by taking a look at the `naturalWidth` property:

```
document.myIMG.naturalWidth
```

The Evaluator will reveal 120 as the natural image width even though the image is currently scaled to 200.

Related Items: `img.height`, `img.width` properties.

protocol

Value: String. Read-Only
Compatibility: WinIE4+, MacIE5+, NN-, Moz-, Safari-

The IE `protocol` property returns only the protocol portion of the complete URL returned by the `src` property. This allows your script, for example, to see if the image is sourced from a local hard drive or a Web server. Values returned are not the actual protocol strings; rather, they are descriptions thereof: `HyperText Transfer Protocol` or `File Protocol`.

Example

You can use The Evaluator Sr. (Chapter 13) to examine the `protocol` property of the image on the page. Enter the following statement into the top text box:

```
document.myIMG.protocol
```

Related Items: `img.src`, `img.nameProp` properties.

src

Value: URL string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

The `src` property is the gateway to precaching images (in instances of the `Image` object that are stored in memory) and performing image swapping (in `img` element objects). Assigning a URL to the `src` property of an image object in memory causes the browser to load the image into the browser's cache (provided the user has the cache turned on). Assigning a URL to the `src` property of an `img` element object causes the element to display the new image. To take advantage of this powerful combination, you preload alternate versions of swappable images into image objects in memory and then assign the `src` property of the image object to the `src` property of the desired `img` element object.

In NN3 and NN4 (all OS platforms) and MacIE3, the size of the image defined by the `img` element's attributes (or, if not specified, then calculated by the browser from the size of the incoming image) governs the rectangular space devoted to that image. An attempt to assign an image of a different size to that `img` element object causes the image to rescale to fit the rectangle (usually resulting in a distorted image). In all later browsers, however, the `img` element object resizes itself to accommodate the image, and the page content reflows around the new size.

Note that when you read the `src` property, it returns a fully formed URL of the image file including protocol and path. This often makes it inconvenient to let the name of the file guide your script to swap images with another image in a sequence of your choice. Some other mechanism (such as storing the current filename in a global variable) may be easier to work with (and see the WinIE5+ `nameProp` property).

Example

In the following example (see Listing 20-4), you see a few applications of image objects. Of prime importance is a comparison of how precached and regular images feel to the user. As a bonus, you see an example of how to set a timer to automatically change the images displayed in an image object. This feature is a popular request among sites that display advertising banners or slide shows.

As the page loads, a global variable is handed an array of image objects. Entries of the array are assigned string names as index values (`"desk1"`, `"desk2"`, and so on). The intention is that these names ultimately will be used as addresses to the array entries. Each image object in the array has a URL assigned to it, which precaches the image.

The page (see Figure 20-1) includes two `img` elements: one that displays noncached images and one that displays cached images. Under each image is a `select` element that you can use to select one of four possible image files for each element. The `onchange` event handler for each `select` list invokes a different function to change the noncached (`loadIndividual()`) or cached (`loadCached()`) images. Both of these functions take as their single parameter a reference to the form that contains the `select` elements.

To cycle through images at five-second intervals, the `checkTimer()` function looks to see if the timer checkbox is checked. If so, the `selectedIndex` property of the cached image `select` control is copied and incremented (or reset to zero if the index is at the maximum value). The `select` element is adjusted, so you can now invoke the `loadCached()` function to read the currently selected item and set the image accordingly.

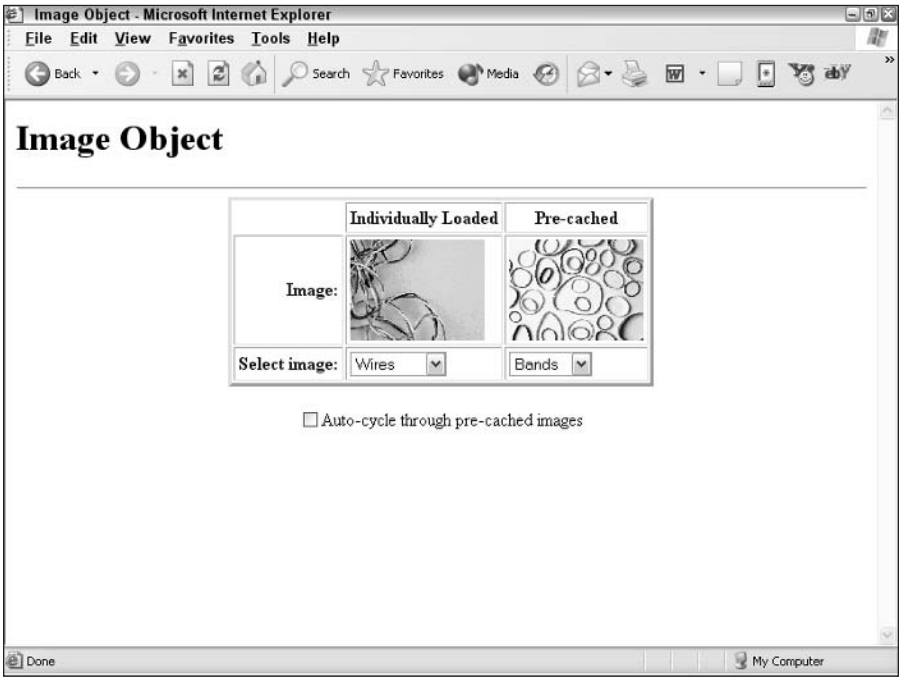


Figure 20-1: The image object demonstration page.

For some extra style points, the <body> tag includes an onunload event handler that invokes the resetSelects() function. This general-purpose function loops through all forms on the page and all elements within each form. For every select element, the selectedIndex property is reset to zero. Thus, if a user reloads the page, or returns to the page via the Back button, the images start in their original sequence. An onload event handler makes sure that the images are in sync with the select choices and the checkTimer() function is invoked with a five-second delay. Unless the timer checkbox is checked, however, the cached images don't cycle.

Listing 20-4: A Scripted Image Object and Rotating Images

```
<html>
   <head>
      <title>Image Object</title>
      <script type="text/javascript">
      // global declaration for 'desk' images array
      var imageDB;
      // pre-cache the 'desk' images
      if (document.images) {
         // list array index names for convenience
         var deskImages = new Array("desk1", "desk2", "desk3", "desk4");
         // build image array and pre-cache them
         imageDB = new Array(4);
         for (var i = 0; i < imageDB.length ; i++) {
            imageDB[deskImages[i]] = new Image(120,90);
            imageDB[deskImages[i]].src = deskImages[i] + ".gif";
         }
      }
      // change image of 'individual' image
      function loadIndividual(form) {
         if (document.images) {
            var gifName =
               form.individual.options[form.individual.selectedIndex].value;
            document.getElementById("thumbnail1").src = gifName + ".gif";
         }
      }
      // change image of 'cached' image
      function loadCached(form) {
         if (document.images) {
            var gifIndex =
               form.cached.options[form.cached.selectedIndex].value;
            document.getElementById("thumbnail2").src = imageDB[gifIndex].src;
         }
      }
      // if switched on, cycle 'cached' image to next in queue
      function checkTimer() {
         if (document.images && document.Timer.timerBox.checked) {
            var gifIndex = document.selections.cached.selectedIndex;
            if (++gifIndex > imageDB.length - 1) {
               gifIndex = 0;
            }
            document.selections.cached.selectedIndex = gifIndex;
            loadCached(document.selections);
            var timeoutID = setTimeout("checkTimer()",5000);
         }
      }
      // reset form controls to defaults on unload
      function resetSelects() {
```

```
            for (var i = 0; i < document.forms.length; i++) {
               for (var j = 0; j < document.forms[i].elements.length; j++) {
                  if (document.forms[i].elements[j].type == "select-one") {
                     document.forms[i].elements[j].selectedIndex = 0;
                  }
               }
            }
         }
         // get things rolling
         function init() {
            loadIndividual(document.selections);
            loadCached(document.selections);
            setTimeout("checkTimer()",5000);
         }
         </script>
      </head>
      <body onload="init()" onunload="resetSelects ()">
         <h1>Image Object</h1>
         <hr />
         <center>
            <table border="3" cellpadding="3">
               <tr>
                  <th></th>
                  <th>Individually Loaded</th>
                  <th>Pre-cached</th>
               </tr>
               <tr>
                  <td align="right"><b>Image:</b></td>
                  <td><img alt="image" src="cpu1.gif" id="thumbnail1"
                     height="90" width="120" /></td>
                  <td><img alt="image" src="desk1.gif" id="thumbnail2"
                     height="90" width="120" /></td>
               </tr>
               <tr>
                  <td align="right"><b>Select image:</b></td>
                  <form name="selections">
                  <td><select name="individual"
                     onchange="loadIndividual(this.form)">
                        <option value="cpu1">Wires</option>
                        <option value="cpu2">Keyboard</option>
                        <option value="cpu3">Disks</option>
                        <option value="cpu4">Cables</option>
                     </select></td>
                  <td><select name="cached" onchange="loadCached(this.form)">
                     <option value="desk1">Bands</option>
                     <option value="desk2">Clips</option>
                     <option value="desk3">Lamp</option>
                     <option value="desk4">Erasers</option>
                  </select></td>
                  </form>
               </tr>
            </table>
            <form name="Timer">
               <input type="checkbox" name="timerBox"
               onclick="checkTimer()" />Auto-cycle through pre-cached images
            </form>
         </center>
      </body>
   </html>
```

Related Items: `img.lowsrc`, `img.nameProp` properties.

start

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `start` property works in conjunction with video clips viewed through the `img` element in IE4+. By default, a clip starts playing (except on the Macintosh) when the image file opens. This follows the default setting of the `start` property: `"fileopen"`. Another recognized value is `"mouseover"`, which prevents the clip from running until the user rolls the mouse pointer atop the image.

Related Items: `img.dynsrc`, `img.loop` properties.

useMap

Value: Identifier string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `useMap` property represents the `usemap` attribute of an `img` element, pointing to the name assigned to the `area` element in the page (see Listing 20-7). This `area` element contains the details about the client-side image map (described later in this chapter). The value for the `useMap` property must include the hash mark that defines an internal HTML reference on the page. If you need to switch among two or more image maps for the same `img` element (for example, you swap images or the user is in a different mode), you can define multiple `map` elements each with a different name. Then change the value of the `useMap` property for the `img` element object to associate a different map with the image.

Related Item: `isMap` property.

vspace

(See `hspace`)

width

(See `height`)

x

y

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

An NN4 script can retrieve the x and y coordinates of an `img` element (the top-left corner of the rectangular space occupied by the image) via the `x` and `y` properties. These properties are read-only. They were supplanted in NN6 via the `offsetLeft` and `offsetTop` properties of any element.

Even without Dynamic HTML, you can use the information from these properties to help scroll a NN4 document to a precise position (with the `window.scrollTo()` method) as a navigational aid in your page. Due to the different ways each operating system platform renders pages and the different sizes of browser windows, you can dynamically locate the position of an image (in other words, scroll the document) given the current client conditions.

Example

If you want to scroll the document so that the link is a few pixels below the top of the window, use a statement such as this:

```
window.scrollTo(document.images[0].x, (document.images[0].y - 3));
```

Related Items: `img.offsetLeft`, `img.offsetTop` properties; `img.scrollIntoView()`, `window.scrollTo()` methods.

Event handlers

onabort
onerror

Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

Your scripts may need to be proactive when a user clicks the Stop button while an image loads or when a network or server problem causes the image transfer to fail. Use the `onabort` event handler to activate a function in the event of a user clicking the Stop button; use the `onerror` event handler for the unexpected transfer snafu.

In practice, these event handlers don't supply all the information you may like to have in a script, such as the filename of the image loading at the time. If such information is critical to your scripts, the scripts need to store the name of a currently loading image to a variable before they set the image's `src` property. You also don't know the nature of the error that triggers an error event. You can treat such problems by forcing a scripted page to reload or by navigating to an entirely different spot in your Web site.

Example

Listing 20-5 includes an `onabort` event handler. If the images already exist in the cache, you must quit and relaunch the browser to try to stop the image from loading. In that example, I provide a reload option for the entire page. How you handle the exception depends a great deal on your page design. Do your best to smooth over any difficulties that users may encounter.

onload

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

An `img` object's `onload` event handler fires when one of three actions occurs: an image's `lowsrc` image finishes loading; in the absence of a `lowsrc` image specification, the `src` image finishes loading; or when each frame of an animated GIF (GIF89a format) appears.

It's important to understand that if you define a `lowsrc` file inside an `<img>` tag, the `img` object receives no further word about the `src` image having completed its loading. If this information is critical to your script, verify the current image file by checking the `src` property of the image object.

Be aware, too, that an `img` element's `onload` event handler may fire before the other elements on the page have completed loading. If the event handler function refers to other elements on the page, the function should verify the existence of other elements prior to addressing them.

img.onload

Quit and restart your browser to get the most from Listing 20-5. As the document first loads, the `lowsrc` image file (the picture of pencil erasers) loads ahead of the computer keyboard image. When the erasers are loaded, the `onload` event handler writes "done" to the text field even though the main image is not loaded yet. You can experiment further by loading the arch image. This image takes longer to load, so the `lowsrc` image (set on the fly, in this case) loads way ahead of it.

Listing 20-5: **The Image onload Event Handler**

```
<html>
   <head>
      <title></title>
      <script type="text/javascript">
      function loadIt(theImage,form) {
         if (document.images) {
            form.result.value = "";
            document.images[0].lowsrc = "desk1.gif";
            document.images[0].src = theImage;
         }
      }
      function checkLoad(form) {
         if (document.images) {
            form.result.value = document.images[0].complete;
         }
      }
      function signal() {
         if(confirm("You have stopped the image from loading. Do you want to
            try again?")) {
            location.reload();
         }
      }
      </script>
   </head>
   <body>
      <img alt="image" src="cpu2.gif" lowsrc="desk4.gif" width="120"
      height="90" onload="if (document.forms[0].result)
      document.forms[0].result.value='done'" onabort="signal()" />
      <form>
         <input type="button" value="Load keyboard"
         onclick="loadIt('cpu2.gif',this.form)" /> <input type="button"
         value="Load arch" onclick="loadIt('arch.gif',this.form)" />
         <p><input type="button" value="Is it loaded yet?"
            onclick="checkLoad(this.form)" /> <input type="text"
            name="result" /> <input type="hidden" /></p>
      </form>
   </body>
</html>
```

Related Items: `img.src`, `img.lowsrc` properties.

area Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
alt		
coords		
hash		
host		
hostname		
href		
noHref		
pathname		
port		
protocol		
search		
shape		
target		

Syntax

Accessing `area` element object properties:

```
(NN3+/IE4+)    [window.]document.links[index].property
(IE4+)         [window.]document.all.elemID.property | method([parameters])
(IE4+)         [window.]document.all.MAPElemID.areas[index].property |
               method([parameters])
(IE5+/W3C)   [window.]document.getElementById("MAPElemID).areas[index].property
|          method([parameters])
(IE5+/W3C)     [window.]document.getElementById("elemID").property |
               method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

About this object

Document object models treat an image map area object as one of the link (`a` element) objects in a document (see the anchor object in Chapter 19). When you think about it, such treatment is not illogical at all because clicking a map area generally leads the user to another document or anchor location in the same document — a hyperlinked reference.

Although the HTML definitions of links and map areas differ greatly, the earliest scriptable implementations of both kinds of objects had nearly the same properties and event handlers. Therefore, to read about the details for these items, refer to the discussion about the link object in Chapter 19. The one difference is that in NN3 and NN4, a map area object does not have the same full array of mouse event handlers — you can count upon having only the `onclick` (NN4+), `onmouseover`, and `onmouseout` event handlers for those browsers.

area

Starting with IE4, NN6, and W3C-compatible browsers, all `area` element attributes are accessible as scriptable properties. Moreover, you can change the makeup of client-side image map areas by way of the `map` element object. The `map` element object contains an array of `area` element objects nested inside. You can remove, modify, or add to the `area` elements inside the `map` element.

Client-side image maps are fun to work with, and they have been well documented in HTML references since Netscape Navigator 2 introduced the feature. Essentially, you define any number of areas within the image based on shape and coordinates. Many graphics tools can help you capture the coordinates of images that you need to enter into the `coords` attribute of the `<area>` tag.

Tip

If one gotcha exists that trips up most HTML authors, it's the tricky link between the `<img>` and `<map>` tags. You must assign a name to the `<map>`; in the `<img>` tag, the `usemap` attribute requires a hash symbol (#) and the map name. If you forget the hash symbol, you can't create a connection between the image and its map.

Listing 20-6 contains an example of a client-side image map that allows you to navigate through different geographical features of the Middle East. As you drag the mouse around an aerial image, certain regions cause the mouse pointer to change, indicating that there is a link associated with the region. Clicking a region results in an alert box indicating which region you clicked.

Listing 20-6: **A Simple Client-Side Image Map**

```
<html>
   <head>
      <title></title>
      <script type="text/javascript">
      function show(msg) {
         window.status = msg;
         return true;
      }
      function go(where) {
         alert("We're going to " + where + "!");
      }
      function clearIt() {
         window.status = "";
         return true;
      }
      </script>
   </head>
   <body>
      <h1>Sinai and Vicinity</h1>
      <img alt="image" src="nile.gif" width="320" height="240"
      usemap="#sinai" />
      <map id="sinai" name="sinai">
         <area href="javascript:go('Cairo')" coords="12,152,26,161"
         shape="rect" onmouseover="return show('Cairo')"
         onmouseout="return clearIt()" />
         <area href="javascript:go('the Nile River')"
         coords="1,155,6,162,0,175,3,201,61,232,109,227,167,238,274,239,292,
         220,307,220,319,230,319,217,298,213,282,217,267,233,198,228,154,227,
         107,221,71,225,21,199,19,165,0,149"
```

```
            shape="poly" onmouseover="return show('Nile River')"
            onmouseout="return clearIt()" />
            <area href="javascript:go('Israel')" coords="95,69,201,91"
            shape="rect" onmouseover="return show('Israel')"
            onmouseout="return clearIt()" />
            <area href="javascript:go('Saudi Arabia')" coords="256,57,319,121"
            shape="rect" onmouseover="return show('Saudi Arabia')"
            onmouseout="return clearIt()" />
            <area href="javascript:go('the Mediterranean Sea')"
            coords="1,55,26,123" shape="rect"
            onmouseover="return show('Mediterranean Sea')"
            onmouseout="return clearIt()" />
            <area href="javascript:go('the Mediterranean Sea')"
            coords="27,56,104,103" shape="rect"
            onmouseover="return show('Mediterranean Sea')"
            onmouseout="return clearIt()" />
         </map>
      </body>
   </html>
```

Properties

alt

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `alt` property represents the `alt` attribute of an `area`. Future browsers may implement this attribute to provide additional information about the link associated with the `area` element.

Related Item: `title` property.

coords
shape

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `coords` and `shape` properties control the location, size, and shape of the image hot spot governed by the `area` element. Shape values that you can use for this property control the format of the `coords` property values, as follows:

Shape	*Coordinates*	*Example*
`circ`	center-x, center-y, radius	`"30, 30, 20"`
`circle`	center-x, center-y, radius	`"30, 30, 20"`
`poly`	x1, y1, x2, y2,...	`"0, 0, 0, 30, 15, 30, 0, 0"`
`polygon`	x1, y1, x2, y2,...	`"0, 0, 0, 30, 15, 30, 0, 0"`
`rect`	left, top, right, bottom	`"10, 20, 60, 40"`
`rectangle`	left, top, right, bottom	`"10, 20, 60, 40"`

The default shape for an area is a rectangle.

Related Items: None.

hash
host
hostname
href
pathname
port
protocol
search
target

(See corresponding properties of the link object in Chapter 19)

shape

(See coords)

map Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
areas[]		onscroll
name		

Syntax

Accessing map element object properties:

```
(IE4+)        [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)    [window.]document.getElementById("elemID").property |
              method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About this object

The map element object is an invisible HTML container for all area elements, each of which defines a "hot" region for an image. Client-side image maps associate links (and targets) to rectangular, circular, or polygonal regions of the image.

By far, the most important properties of a map element object are the areas array and, to a lesser extent, its name. It is unlikely that you will change the name of a map. (It is better to define multiple map elements with different names, and then assign the desired name to an img element object's useMap property.) But you can use the areas array to change the makeup of the area objects inside a given client-side map.

Properties

areas[]

Value: Array of `area` element objects. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Use the `areas` array to iterate through all `area` element objects within a `map` element. While NN6 adheres closely to the document node structure of the W3C DOM, IE4+ provides more direct access to the `area` element objects nested inside a `map`. If you want to rewrite the `area` elements inside a `map`, you can clear out the old ones by setting the `length` property of the `areas` array to zero. Then assign `area` element objects to slots in the array to build that array.

Listing 20-7 demonstrates how to use scripting to replace the `area` element objects inside a `map` element. The scenario is that the page loads with one image of a computer keyboard. This image is linked to the `keyboardMap` client-side image map, which specifies details for three hot spots on the image. If you then switch the image displayed in that `img` element, scripts change the `useMap` property of the `img` element object to point to a second `map` that has specifications more suited to the desk lamp in the second image. Roll the mouse pointer atop the images, and view the URLs associated with each area in the status bar (for this example, the URLs do not lead to other pages).

Another button on the page, however, invokes the `makeAreas()` function (not working in MacIE5), which creates four new `area` element objects and (through DOM-specific pathways) adds those new area specifications to the image. If you roll the mouse atop the image after the function executes, you can see that the URLs now reflect those of the new areas. Also note the addition of a fourth area, whose status bar message appears in Figure 20-2.

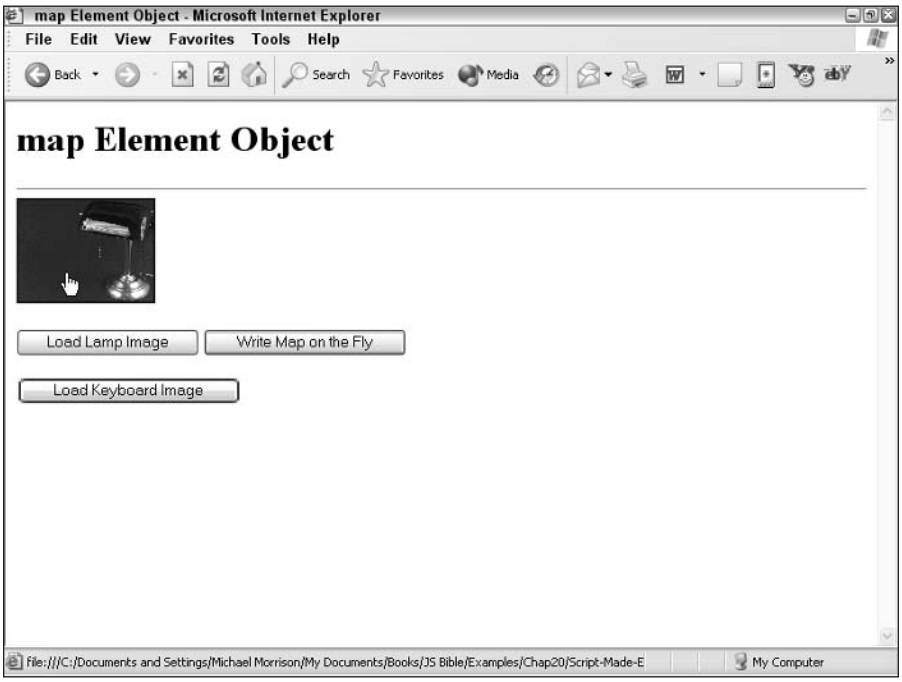


Figure 20-2: Scripts created a special client-side image map for the image.

Listing 20-7: **Modifying area Elements on the Fly**

```
<html>
   <head>
      <title>map Element Object</title>
      <script type="text/javascript">
      // generate area elements on the fly
      function makeAreas() {
         document.getElementById("myIMG").src = "desk3.gif";
         // build area element objects
         var area1 = document.createElement("area");
         area1.href = "Script-Made-Shade.html";
         area1.shape = "polygon";
         area1.coords = "52,28,108,35,119,29,119,8,63,0,52,28";
         var area2 = document.createElement("area");
         area2.href = "Script-Made-Base.html";
         area2.shape = "rect";
         area2.coords = "75,65,117,87";
         var area3 = document.createElement("area");
         area3.href = "Script-Made-Chain.html";
         area3.shape = "polygon";
         area3.coords = "68,51,73,51,69,32,68,51";
         var area4 = document.createElement("area");
         area4.href = "Script-Made-Emptyness.html";
         area4.shape = "rect";
         area4.coords = "0,0,50,120";
         // stuff new elements into MAP child nodes
         var mapObj = document.getElementById("lamp_map");
         while (mapObj.childNodes.length) {
            mapObj.removeChild(mapObj.firstChild);
         }
         mapObj.appendChild(area1);
         mapObj.appendChild(area2);
         mapObj.appendChild(area3);
         mapObj.appendChild(area4);
         // workaround NN6 display bug
         document.getElementById("myIMG").style.display = "inline";
      }

      function changeToKeyboard() {
         document.getElementById("myIMG").src = "cpu2.gif";
         document.getElementById("myIMG").useMap = "#keyboardMap";
      }

      function changeToLamp() {
         document.getElementById("myIMG").src = "desk3.gif";
         document.getElementById("myIMG").useMap = "#lampMap";
      }
      </script>
   </head>
   <body>
      <h1>map Element Object</h1>
      <hr />
      <img alt="image" id="myIMG" src="cpu2.gif" width="120" height="90"
      usemap="#keyboardMap" />
```

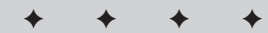
```
      <map id="keyboardMap" name="keyboardMap">
         <area href="AlpaKeys.htm" shape="rect" coords="0,0,26,42" />
         <area href="ArrowKeys.htm" shape="polygon"
         coords="48,89,57,77,69,82,77,70,89,78,84,89,48,89" />
         <area href="PageKeys.htm" shape="circle" coords="104,51,14" />
      </map>
      <map name="lampMap" id="lamp_map">
         <area href="Shade.htm" shape="polygon"
         coords="52,28,108,35,119,29,119,8,63,0,52,28" />
         <area href="Base.htm" shape="rect" coords="75,65,117,87" />
         <area href="Chain.htm" shape="polygon"
         coords="68,51,73,51,69,32,68,51" />
      </map>
      <form>
         <p><input type="button" value="Load Lamp Image"
            onclick="changeToLamp()" /> <input type="button"
            value="Write Map on the Fly" onclick="makeAreas()" /></p>
         <p><input type="button" value="Load Keyboard Image"
            onclick="changeToKeyboard()" /></p>
      </form>
   </body>
</html>
```

Related Items: `area` element object.

✦ ✦ ✦

CHAPTER 21

The Form and Related Objects

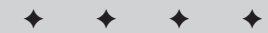


In This Chapter

The `form` object as a container of form controls

Processing form validations

`label`, `fieldset`, and `legend` element objects



Prior to the advent of dynamic object models and automatic page reflow, the majority of scripting in an HTML document took place in and around forms. Even with all the new DHTML powers, forms remain the primary user interface elements of HTML documents because they enable users to input information and make choices in very familiar user interface elements, such as buttons, option lists, and so on. The challenge of scripting forms and form elements often involves getting object references just right. The references can get pretty long by the time you start pointing to the property of a form control element (which is part of a form, which is part of a document, which is part of a window or frame).

Expanded object models of W3C-compatible browsers include scriptable access to form-related elements that are part of the HTML 4.0 specification. One pair of elements, `fieldset` and `legend`, provides both contextual and visual containment of form controls in a document. Another element, `label`, provides context for text labels that usually appear adjacent to form controls. Although there is generally little need to script these objects, the browsers give you access to them just as they do for virtually every HTML element supported by the browser.

The Form in the Object Hierarchy

Take another look at the JavaScript object hierarchy in the lowest common denominator object model (refer back to Figure 14-1). The `form` element object can contain a wide variety of form element objects (sometimes called *form controls*), which I cover in Chapters 22 through 24. In this chapter, however, I focus primarily on the container.

The good news on the compatibility front is that much of the client-side scripting works on all scriptable browsers. While you are free to use newer ways of addressing forms and their nested elements when your audience exclusively uses the newer browsers, it can serve you well to be comfortable with the "old-fashioned" reference syntax. Therefore, almost all example code in this and the next three chapters uses syntax that is compatible with the earliest scriptable browsers. Besides, the only significant additions to the defining points of the `form` object in newer browsers are those characteristics that all other HTML elements share.

form Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
acceptCharset	handleEvent()	onreset
action	reset()	onsubmit
autocomplete	submit()	
elements[]		
encoding		
enctype		
length		
method		
name		
target		

Syntax

Accessing `form` object properties or methods:

```
(All)       [window.]document.formName. property | method([parameters])
(All)       [window.]document.forms[index]. property | method([parameters])
(All)       [window.]document.forms["formName"]. property | method([parameters])
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

About this object

Forms and their elements are the most common two-way gateways between users and JavaScript scripts. A form control element provides the only way that users can enter textual information. Form controls also provide somewhat standardized and recognizable user interface elements for the user to make a selection from a predetermined set of choices. Sometimes those choices appear in the form of an on/off checkbox, in a set of mutually exclusive radio buttons, or as a selection from a list.

As you have seen in many Web sites, the form is the avenue for the user to enter information that is sent to the server housing the Web files. Just what the server does with this information depends on the CGI (Common Gateway Interface) programs running on the server. If your Web site runs on a server directly under your control (that is, it is *in-house* or *hosted* by a service), you have the freedom to set up all kinds of data-gathering or database search programs to interact with the user. But with some of the more consumer-oriented Internet service providers (ISPs), you may have no CGI support available — or, at best, a limited set of

popular but inflexible CGI programs available to all customers of the service. Custom databases or transactional services are rarely provided for this kind of Internet service.

Regardless of your Internet server status, you can find plenty of uses for JavaScript scripts in forms. For instance, rather than using data exchanges (and Internet bandwidth) to gather raw user input and report any input errors, a JavaScript-enhanced document can preprocess the information to make sure that it employs the format that your back-end database or other programs most easily process. All corrective interaction takes place in the browser, without one extra bit flowing across the Net. I devote all of Chapter 43 (on the CD-ROM) to these kinds of form data-validation techniques.

How you define a `form` element (independent of the user interface elements described in subsequent chapters) depends a great deal on how you plan to use the information from the form's controls. If you intend to use the form exclusively for JavaScript purposes (that is, no queries or postings going to the server), you do not need to use the `action`, `target`, and `method` attributes. But if your Web page will be feeding information or queries back to a server, you need to specify at least the `action` and `method` attributes. You need to also specify the `target` attribute if the resulting data from the server is to be displayed in a window other than the calling window and the `enctype` attribute if your form's scripts fashion the server-bound data in a MIME type other than in a plain ASCII stream.

References to form control elements

For most client-side scripting, user interaction comes from the elements within a form; the `form` element object is merely a container for the various control elements. If your scripts perform any data validation checks on user entries prior to submission or other calculations, many statements have the `form` object as part of the reference to the element.

A complex HTML document can have multiple `form` objects. Each `<form>...</form>` tag pair defines one form. You don't receive any penalties (except for potential confusion on the part of someone reading your script) if you reuse a name for an element in each of a document's forms. For example, if each of three forms has a grouping of radio buttons with the name "choice," the object reference to each button ensures that JavaScript doesn't confuse them. The reference to the first button of each of those button groups is as follows:

```
document.forms[0].choice[0]
document.forms[1].choice[0]
document.forms[2].choice[0]
```

If you assign identifiers to `id` attributes, however, you should not reuse an identifier on the same page.

Passing forms and elements to functions

When a form or form element contains an event handler that calls a function defined elsewhere in the document, you can use a couple of shortcuts to simplify the task of addressing the objects while the function does its work. Failure to grasp this concept not only causes you to write more code than you have to, but it also hopelessly loses you when you try to trace somebody else's code in his or her JavaScripted document. The watchword in event handler parameters is

```
this
```

which represents a reference to the current object that contains the event handler attribute. For example, consider the function and form definition in Listing 21-1. The entire user interface for this listing consists of form elements, as shown in Figure 21-1.

Listing 21-1: **Passing the form Object as a Parameter**

```
<html>
  <head>
     <title>Beatle Picker</title>
     <script type="text/javascript">
     function processData(form) {
        for (var i = 0; i < form.Beatles.length; i++) {
           if (form.Beatles[i].checked) {
              break;
           }
        }
        var chosenBeatle = form.Beatles[i].value;
        var chosenSong = form.song.value;
        alert("Looking to see if " + chosenSong + " was written by " +
           chosenBeatle + "...");
     }

     function checkSong(songTitle) {
        var enteredSong = songTitle.value;
        alert("Making sure that " + enteredSong + " was recorded by the
           Beatles.");
     }
     </script>
  </head>
  <body>
     <form name="Abbey Road">
        Choose your favorite Beatle: <input type="radio" name="Beatles"
        value="John Lennon" checked="true" />John <input type="radio"
        name="Beatles" value="Paul McCartney" />Paul <input type="radio"
        name="Beatles" value="George Harrison" />George <input type="radio"
        name="Beatles" value="Ringo Starr" />Ringo
        <p>Enter the name of your favorite Beatles song:<br />
           <input type="text" name="song" value="Eleanor Rigby"
           onchange="checkSong(this)" /></p>
        <p><input type="button" name="process" value="Process Request..."
           onclick="processData(this.form)" /></p>
     </form>
  </body>
</html>
```

The `processData()` function, which needs to read and write properties of multiple form control elements, can reference the controls in two ways. One way is to have the `onclick` event handler (in the button element at the bottom of the document) call the `processData()` function and not pass any parameters. Inside the function, all references to objects (such as the radio buttons or the song field) must be complete references, such as

```
document.forms[0].song.value
```

to retrieve the value entered into the `song` field.

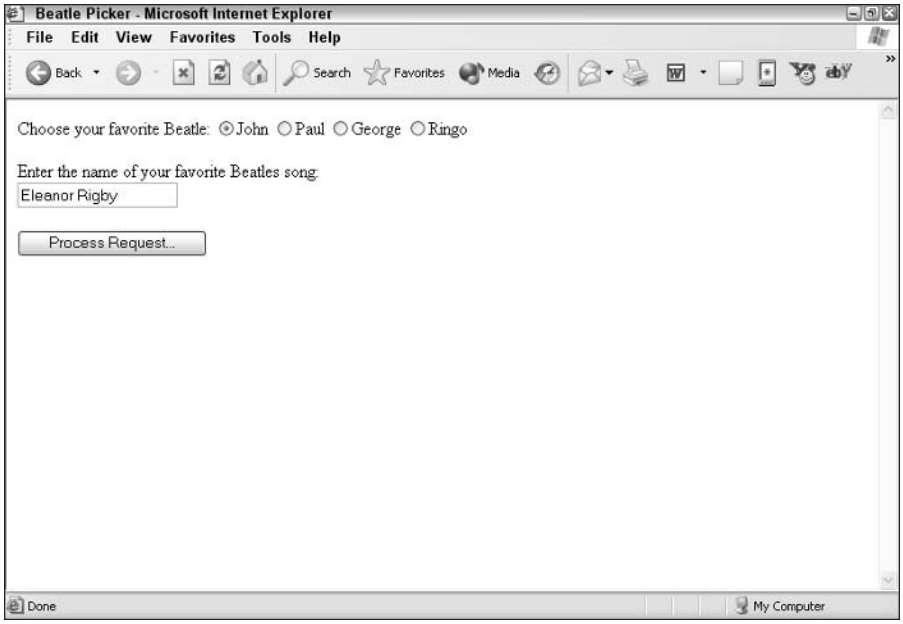


Figure 21-1: Controls pass different object references to functions in Listing 21-1.

A more efficient way is to send a reference to the `form` object as a parameter with the call to the function (as shown in Listing 21-1). By specifying `this.form` as the parameter, you tell JavaScript to send along everything it knows about the form from which the function is called. This works because `form` is a property of every form control element; the property is a reference to the form that contains the control. Therefore, `this.form` passes the value of the `form` property of the control.

At the function, the reference to the `form` object is assigned to a variable name (arbitrarily set to `form` here) that appears in parentheses after the function name. I use the parameter variable name `form` here because it represents an entire form. But you can use any valid variable name you like.

The reference to the form contains everything the browser needs to know to find that form within the document. Any statements in the function can therefore use the parameter value in place of the longer, more cumbersome reference to the form. Thus, here I can use `form` to take the place of `document.forms[0]` in any address. To get the value of the `song` field, the reference is:

```
form.song.value
```

Had I assigned the `form` object to a parameter variable called `sylvester`, the reference would have been:

```
sylvester.song.value
```

When a function parameter is a reference to an object, statements in the function can retrieve or set properties of that object as well as invoke the object's methods.

Another version of the `this` parameter passing style simply uses the word `this` as the parameter. Unlike `this.form`, which passes a reference to the entire form connected to a particular element, `this` passes a reference only to that one element. In Listing 21-1, you can add an event handler to the `song` field to do some validation of the entry (to make sure that the entry appears in a database array of Beatles' songs created elsewhere in the document). Therefore, you want to send only the `field` object to the function for analysis:

```
<input type="text" name="song" onchange="checkSong(this)" />
```

You then have to create a function to catch this call:

```
function checkSong(songTitle) {
   var enteredSong = songTitle.value;
   alert("Making sure that " + enteredSong + " was recorded by the Beatles.");
}
```

Within this function, you can go straight to the heart — the `value` property of the field element — without a long reference.

One further extension of this methodology passes only a single property of a form control element as a parameter. In the last example, the `checkSong()` function needs only the `value` property of the field, so the event handler can pass `this.value` as a parameter. Because `this` refers to the very object in which the event handler appears, the `this.`*`propertyName`* syntax enables you to extract and pass along a single property:

```
<input type="text" name="song" onchange="checkSong(this.value)" />
```

A benefit of this way of passing form element data is that the function doesn't have to do as much work:

```
function checkSong(songTitle) {
   alert("Making sure that " + songTitle + " was recorded by the Beatles.");
}
```

Unlike passing object references (like the form and text field objects above), when you pass a property value (for example, `this.value`), the property's value is passed with no reference to the object from which it came. This suffices when the function just needs the value to do its job. However, if part of that job is to modify the object's property (for example, converting all text from a field to uppercase and redisplaying the converted text), the value passed to the function does not maintain a "live" connection with its object. To modify a property of the object that invokes an event handler function, you need to pass some object reference so that the function knows where to go to work on the object.

Many programmers with experience in other languages expect parameters to be passed either by reference or by value, but not both ways. The rule of thumb in JavaScript, however, is fairly simple: object references are passed by reference; property values are passed by value.

Here are some guidelines to follow when deciding what kind of value to pass to an event handler function:

- ✦ Pass the entire form control object (`this`) when the function needs to make subsequent access to that same element (perhaps reading an object's `value` property, converting the value to all uppercase letters, and then writing the result back to the same object's `value` property).
- ✦ Pass only one property (`this.`*`propertyName`*) when the function needs read-only access to that property.

- ✦ Pass the entire `form` element object (`this.form`) for the function to access multiple elements inside a form (for example, a button click means that the function must retrieve a field's content).

Also be aware that you can submit multiple parameters (for example, `onclick="someFunction(this.form, this.name)"`) or even an entirely different object from the same form (for example, `onclick="someFunction(this.form.emailAddr.value)"`). Simply adjust your function's incoming parameters accordingly. (See Chapter 33 for more details about custom functions.)

E-mailing forms

A common request among scripters is how to send a form via e-mail to the page's author. This includes the occasional desire to send "secret" e-mail to the author whenever someone visits the Web site. Let me address the privacy issue first.

A site visitor's e-mail address is valuable personal information that you should not retrieve without the visitor's permission or knowledge. That's one reason why Netscape plugged a privacy hole in Navigator 2 that allowed submitting a form to a `mailto:` URL without requesting permission from the user. You can use some workarounds for this in Navigator 3, but I do not condone surreptitiously lifting e-mail addresses and therefore choose not to publicize those workarounds here. Besides, the workarounds fail in newer browsers anyway.

Microsoft, on the other hand, went too far in preventing forms e-mailing in the earliest browser versions. While Netscape's browsers reveal to the user in an alert that an e-mail message bearing the user's e-mail address (as stored in the browser's preferences) will be sent upon approval, Internet Explorer 3 does not send form content via e-mail at all. Internet Explorer 4 sends form content as an attachment through Microsoft Outlook, but only after displaying a mail composition window to the user. Starting with IE5, the process is somewhat more fluid, but if the user doesn't have a compatible e-mail application installed as the default, the job fails. Due to the unreliable nature and occasionally awkward user interface of mailing a form via the `mailto:` URL, I do not recommend its use.

Many ISPs that host Web sites provide standard CGIs for forwarding forms to an e-mail address of your choice. Search the Web for "formmail service" to locate third-party suppliers of this feature if you don't have access to server programming for yourself.

The remaining discussion about mailing forms focuses primarily on NN2+ and IE5+ browsers and assumes an ideally configured e-mail program is installed. You should be aware that mailing forms in the following ways is controversial in some Web standards circles. As of this writing, the W3C HTML specification does not endorse these techniques specifically. Use these facilities judiciously and only after extensive testing on the client browsers you intend to support.

If you want to have forms submitted as e-mail messages, you must attend to three `<form>` tag attributes. The first is the `method` attribute. You must set it to `POST`. Next comes `enctype`. If you omit this attribute, the e-mail client sends the form data as an attachment consisting of escaped name-value pairs, as in this example:

```
name=Danny+Goodman&rank=Scripter+First+Class&serialNumber=042
```

But if you set the `enctype` attribute to text/plain, the form name-value pairs are placed in the body of the mail message in a more human-readable format:

```
name=Danny Goodman
rank=Scripter First Class
serialNumber=042
```

The last attribute of note is the `action` attribute, which is normally the spot to place a URL to another file or server CGI. Substitute the URL with the special `mailto:` URL followed by an optional parameter for the subject. Here is an example:

```
action="mailto:prez@whitehouse.gov?subject=Opinion%20Poll"
```

To sum up, the following example shows the complete `<form>` tag for e-mailing the form in Navigator:

```
<form name="entry"
   method="POST"
   enctype="text/plain"
   action="mailto:prez@whitehouse.gov?subject=Opinion Poll">
```

None of this requires any JavaScript at all. But seeing how you can use the attributes — and the fact that these attributes are exposed as properties of the `form` element object — you might see some extended possibilities for script control over forms.

Changing form attributes

With the exception of IE3 (whose `form` object properties are read-only), all scriptable browsers expose `form` element attributes as modifiable properties. Therefore, you can change, say, the action of a form via a script in response to user interaction on your page. For example, you can have two different CGI programs invoked on your server depending on whether a form's checkbox is checked.

The best opportunity to change the properties of a `form` element object is in a function invoked by the form's `onsubmit` event handler. The modifications are performed at the last instant prior to actual submission, leaving no room for user-induced glitches to get in the way.

Buttons in forms

A common mistake that newcomers to scripting make is defining all clickable buttons as the submit type of `input` object (`<input type="submit">`). The Submit button does exactly what it says — it submits the form. If you don't set any `method` or `action` attributes of the `<form>` tag, the browser inserts its default values for you: `method=GET` and `action=`*`pageURL`*. When you submit a form with these attributes, the page reloads itself and resets all field values to their initial values.

Use a Submit button only when you want the button to actually submit the form. If you want a button for other types of action, use the button style (`<input type="button">`). A regular button can invoke a function that performs some internal actions and then invokes the `form` element object's `submit()` method to submit the form under script control.

Redirection after submission

Undoubtedly, you have submitted a form to a site and seen a "Thank You" page come back from the server to verify that your submission was accepted. This is warm and fuzzy, if not logical, feedback for the submission action. It is not surprising that you would want to re-create that effect even if the submission is to a `mailto:` URL. Unfortunately, a problem gets in the way.

A common sense approach to the situation calls for a script to perform the submission (via the `form.submit()` method) and then navigate to another page that does the "Thank You." Here is such a scenario from inside a function triggered by a click of a link surrounding a nice, graphical Submit button:

```
function doSubmit() {
   document.forms[0].submit();
   location.href = "thanks.html";
}
```

The problem is that when another statement executes immediately after the `form.submit()` method, the submission is canceled. In other words, the script does not wait for the submission to complete itself and verify to the browser that all is well (even though the browser appears to know how to track that information given the status bar feedback during submission). The point is, because JavaScript does not provide an event that is triggered by a successful submission, there is no sure-fire way to display your own "Thank You" page.

Don't be tempted by the `window.setTimeout()` method to change the location after some number of milliseconds following the `form.submit()` method. You cannot predict how fast the network and/or server is for every visitor. If the submission does not complete before the timeout ends, the submission is still canceled — even if it is partially complete.

Form element arrays

Starting with NN2 and IE4, document object models provide a feature that is beneficial to a lot of scripters. If you create a series of like-named objects, they automatically become an array of objects accessible via array syntax (see Chapter 7). This is particularly helpful when you create forms with columns and rows of fields, such as in an order form. By assigning the same name to all fields in a column, you can employ `for` loops to cycle through each row using the loop index as an array index.

As an example, the following code shows a typical function that calculates the total for an order form row (and calls another custom function to format the value):

```
function extendRows(form) {
   for (var i = 0; i < Qty.length; i++) {
      var rowSum = form.Qty[i].value * form.Price[i].value;
      form.Total[i].value = formatNum(rowSum,2);
   }
}
```

All fields in the Qty column are named `Qty`. The item in the first row has an array index value of zero and is addressed as `form.Qty[i]`.

About <input> element objects

While this chapter focuses strictly on the `form` element as a container of controls, the next three chapters discuss different types of controls that nest inside a form. Many of these controls share the same HTML tag: `<input>`. Only the `type` attribute of the `<input>` tag determines whether the browser shows you a clickable button, a checkbox, a text field, or so on. The fact that one element has so many guises makes the system seem illogical at times to scripters.

An `input` element has some attributes (and corresponding scriptable object properties) that simply don't apply to every type of form control. For example, while the `maxLength` property

of a text box makes perfect sense in limiting the number of characters that a user can type into it, the property has no bearing whatsoever on form controls that act as clickable buttons. Similarly, you can switch a radio button or checkbox on or off by adjusting the `checked` property; however, that property simply doesn't apply to a text box.

As the document object models have evolved, they have done so in an increasingly object-oriented way. The result in this form-oriented corner of the model is that all elements created via the `<input>` tag have a long list of characteristics that they all share by virtue of being types of `input` elements — they inherit the properties and methods that are defined for any `input` element. To try to limit the confusion, I divide the chapters in this book that deal with `input` elements along functional lines (clickable buttons in one chapter, text fields in the other), and only list and discuss those `input` element properties and methods that apply to the specific control type.

In the meantime, this chapter continues with details of the `form` element object.

Properties

acceptCharset

Value: String. Read/Write
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The `acceptCharset` property represents the `acceptcharset` attribute of the `form` element in HTML 4.0. The value is a list of one or more recognized character sets that the server receiving the form must support. For a list of registered character set names, see `http://www.iana.org/assignments/character-sets`.

Related Items: None.

action

Value: URL string. Read/Write (see text)
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `action` property (along with the `method` and `target` properties) primarily functions for HTML authors whose pages communicate with server-based CGI scripts. This property is the same as the value you assign to the `action` attribute of a `<form>` tag. The value is typically a URL on the server where queries or postings are sent for submission.

User input may affect how you want your page to access a server. For example, a checked box in your document may set a form's `action` property so that a CGI script on one server handles all the input, whereas an unchecked box means the form data goes to a different CGI script or a CGI script on an entirely different server. Or, one setting may direct the action to one `mailto:` address, whereas another setting sets the `action` property to a different `mailto:` address.

Although the specifications for all three related properties indicate that you can set them on the fly, such changes are ephemeral. A soft reload eradicates any settings you make to these properties, so you should make changes to these properties only in the same script function that submits the form (see `form.submit()` later in this chapter).

Related Items: `form.method`, `form.target`, `form.encoding` properties.

autocomplete

Value: String. Read/Write
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

Microsoft added a feature to forms with WinIE5 that allows the browser to supply hints for filling out form controls if the controls' names map to a set of single-line text controls defined via some additional attributes linked to the vCard XML schema. For details on implementing this browser feature, see `http://msdn.microsoft.com/workshop/author/forms/autocomplete_ovr.asp`. Values for the `autoComplete` property are your choice of two strings: `on` or `off`. In either case, the `form` element object does not report knowing about this property unless you set the `autocomplete` attribute in the form's tag.

Related Items: None.

elements[]

Value: Array of form control elements. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Elements include all the user interface elements defined for a form: text fields, buttons, radio buttons, checkboxes, selection lists, and more. The `elements` property is an array of all form control items defined within the current form. For example, if a form defines three `<input>` items, the `elements` property for that form is an array consisting of three entries (one for each item in source code order). Each entry is a valid reference to that element; so, to extract properties or call methods for those elements, your script must dig deeper in the reference. Therefore, if the first element of a form is a text field and you want to extract the string currently showing in the field (a text element's `value` property), the reference looks like this:

```
document.forms[0].elements[0].value
```

Notice that this reference summons two array-oriented properties along the way: one for the document's `forms` property and one for the form's `elements` property.

In practice, I suggest you refer to form controls (and forms) by their names. This allows you the flexibility to move controls around the page as you fine-tune the design, and you don't have to worry about the source code order of the controls. The `elements` array comes in handy when you need to iterate through all of the controls within a form. If your script needs to loop through all elements of a form in search of particular kinds of elements, use the `type` property of every `form` object (NN3+, IE4+, and W3C) to identify which kind of object it is. The `type` property consists of the same string used in the `type` attribute of an `<input>` tag.

Overall, I prefer to generate meaningful names for each form control element and use those names in references throughout my scripts. The `elements` array helps with form control names, as well. Instead of a numeric index to the `elements` array, you can use the string name of the control element as the index. Thus, you can create a generic function that processes any number of form control elements, and simply pass the string name of the control as a parameter to the function. Then use that parameter as the `elements` array index value. For example:

```
function putVal(controlName, val) {
   document.forms[0].elements[controlName].value = val;
}
```

If you want to modify the number of controls within a form, you should use the element and/or node management facilities of the browser(s) of your choice. For example, in IE4+ and W3C browsers, you can assemble the HTML string for an entirely new set of form controls and then assign that string to the `innerHTML` property of the `form` element object.

The document in Listing 21-2 demonstrates a practical use of the `elements` property. A form contains four fields and some other elements mixed in between (see Figure 21-2). The first part of the function that acts on these items repeats through all the elements in the form to find out which ones are text box objects and which text box objects are empty. Notice how I use the `type` property to separate text box objects from the rest, even when radio buttons appear amid the fields. If one field has nothing in it, I alert the user and use that same index value to place the insertion point at the field with the field's `focus()` method.

Listing 21-2: **Using the form.elements Array**

```
<html>
   <head>
      <title>Elements Array</title>
      <script type="text/javascript">
      function verifyIt() {
         var form = document.forms[0];
         for (i = 0; i < form.elements.length; i++) {
            if (form.elements[i].type == "text" &&
               form.elements[i].value == "") {
               alert("Please fill out all fields.");
               form.elements[i].focus();
               break;
            }
            // more tests
         }
         // more statements
      }
      </script>
   </head>
   <body>
      <form>
         Enter your first name:<input type="text" name="firstName" />
         <p>Enter your last name:<input type="text" name="lastName" /></p>
         <p><input type="radio" name="gender" />Male <input type="radio"
            name="gender" />Female</p>
         <p>Enter your address:<input type="text" name="address" /></p>
         <p>Enter your city:<input type="text" name="city" /></p>
         <p><input type="checkbox" name="retired" />I am retired</p>
      </form>
      <form>
         <input type="button" name="act" value="Verify" onclick="verifyIt()" />
      </form>
   </body>
</html>
```

Related Items: `text`, `textarea`, `button`, `radio`, `checkbox`, `select` objects.

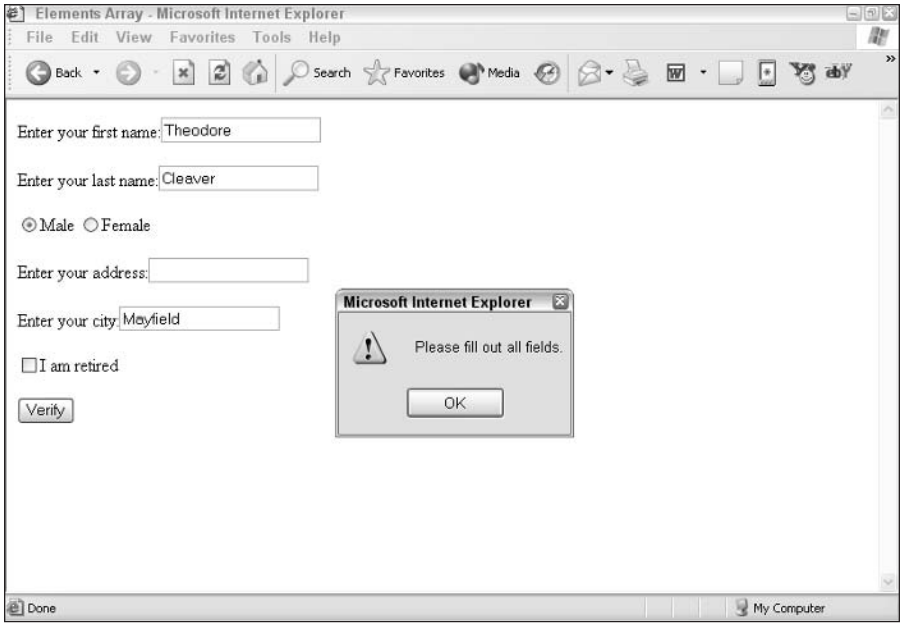


Figure 21-2: The elements array helps find text fields for validation.

encoding
enctype

Value: MIME type string. Read/Write (see text)
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

You can define a form to alert a server when the data you submit is in a MIME type. The `encoding` property reflects the setting of the `enctype` attribute in the form definition. The `enctype` property name is defined for `form` element objects in the W3C DOM (with `encoding` removed), but NN6+ provides both properties for backward and forward compatibility.

For `mailto:` URLs, I recommend setting this value (in the tag or via script) to `"text/plain"` to have the form contents placed in the mail message body. If the definition does not have an `enctype` attribute, this property is an empty string.

Related Items: `form.action`, `form.method` properties.

length

Value: Integer. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `length` property of a `form` element object provides the same information as the `length` property of the form's `elements` array. The property provides a convenient, if not entirely logical, shortcut to retrieving the number of controls in a form.

Related Items: `form.elements` property.

method

Value: String (GET or POST). Read/Write (see text)
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

A form's method property is either the GET or POST value (not case-sensitive) assigned to the method attribute in a <form> definition. Terminology overlaps here a bit, so be careful to distinguish a form's method of transferring its data to a server from the object-oriented method (action or function) that all JavaScript forms have.

The method property is of primary importance to HTML documents that submit a form's data to a server-based CGI script because it determines the format used to convey this information. For example, to submit a form to a mailto: URL, the method property must be POST. Details of forms posting and CGI processing are beyond the scope of this book. Consult HTML or CGI documentation to determine which is the appropriate setting for this attribute in your Web server environment. If a form does not have a method attribute explicitly defined for it, the default value is GET.

Related Items: form.action, form.target, form.encoding properties.

name

Value: Identifier string. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Assigning a name to a form via the name attribute is optional but highly recommended when your scripts need to reference a form or its elements. This attribute's value is retrievable as the name property of a form. You don't have much need to read this property unless you inspect another source's document for its form construction, as in:

```
var formName = parent.frameName.document.forms[0].name;
```

Moreover, because CGI programs frequently rely on the name of the form for validation purposes, it is unlikely you will need to change this property.

target

Value: Identifier string. Read/Write (see text)
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Whenever an HTML document submits a query to a server for processing, the server typically sends back an HTML page — whether it is a canned response or, more likely, a customized page based on the input provided by the user. You see this situation all the time when you perform a search at Web sites. In a multiframe or multiwindow environment, you may want to keep the form part of this transaction in view for the user but leave the responding page in a separate frame or window for viewing. The purpose of the target attribute of a <form> definition is to enable you to specify where the output from the server's query should be displayed.

The value of the target property is the name of the window or frame. For instance, if you define a frameset with three frames and assign the names Frame1, Frame2, and Frame3 to them, you need to supply one of these names (as a quoted string) as the parameter of the target attribute of the <form> definition. Browsers also observe four special window names that you can use in the <form> definition: _top, _parent, _self, and _blank. To set the target as a separate subwindow opened via a script, use the window name from the window.open() method's second parameter and not the window object reference that the method returns.

If you code your page to validate according to strict XHTML, you won't be able to include a `target` attribute for a form. But you can still use a script to assign a value to the property without interfering with the validation.

Related Items: `form.action`, `form.method`, `form.encoding` properties.

Methods

`handleEvent(event)`

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

See the discussion of the `window.handleEvent()` method in Chapter 16 for a description of this NN4-specific method.

`reset()`

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

A common practice, especially with a long form, is to provide a button that enables the user to return all the form elements to their default settings. The standard Reset button (a separate object type described in Chapter 22) does that task just fine. But if you want to clear the form using script control, you must do so by invoking the `reset()` method for the form. More than likely, such a call is initiated from outside the form, perhaps from a function or graphical button. In such cases, make sure that the reference to the `reset()` method includes the complete reference to the form you want to reset — even if the page only has one form defined for it.

In Listing 21-3, I assign the act of resetting the form to the `href` attribute of a link object (that is attached to a graphic called `reset.jpg`). I use the `javascript:` URL to invoke the `reset()` method for the form directly (in other words, without doing it via function). Note that the form's action in this example is to a nonexistent URL. If you click the Submit icon, you receive an "unable to locate" error from the browser.

Listing 21-3: form.reset() and form.submit() Methods

```
<html>
   <head>
      <title>Registration Form</title>
   </head>
   <body>
      <form name="entries" method="POST"
      action="http://www.u.edu/pub/cgi-bin/register">
         Enter your first name:<input type="text" name="firstName" />
         <p>Enter your last name:<input type="text" name="lastName" /></p>
         <p>Enter your address:<input type="text" name="address" /></p>
         <p>Enter your city:<input type="text" name="city" /></p>
         <p><input type="radio" name="gender" checked="checked" />Male <input
            type="radio" name="gender" />Female</p>
         <p><input type="checkbox" name="retired" />I am retired</p>
      </form>
```

Continued

Listing 21-3 *(continued)*

```
      <p><a href="javascript:document.forms[0].submit()"><img alt="image"
         src="submit.jpg" height="25" width="100" border="0" /></a> <a
         href="javascript:document.forms[0].reset()"><img alt="image"
         src="reset.jpg" height="25" width="100" border="0" /></a></p>
   </body>
</html>
```

Related Items: `onreset` event handler; `reset` object.

`submit()`

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The most common way to send a form's data to a server's CGI program for processing is to have a user click a Submit button. The standard HTML Submit button is designed to send data from all named elements of a form according to the specifications listed in the `<form>` definition's attributes. But if you want to submit a form's data to a server automatically for a user, or want to use a graphical button for submission, you can accomplish the submission with the `form.submit()` method.

Invoking this method is almost the same as a user clicking a form's Submit button (except that the `onsubmit` event handler is not triggered). Therefore, you may have an image on your page that is a graphical submission button. If that image is surrounded by a link object, you can capture a mouse click on that image and trigger a function whose content includes a call to a form's `submit()` method (see Listing 21-3).

In a multiple-form HTML document, however, you must reference the proper form either by name or according to its position in a `document.forms` array. Always make sure that the reference you specify in your script points to the desired form before you submit any data to a server.

As a security and privacy precaution for people visiting your site, JavaScript ignores all `submit()` methods whose associated form actions are set to a `mailto:` URL. Many Web page designers would love to have secret e-mail addresses captured from visitors. Because such a capture can be considered an invasion of privacy, the power has been disabled since NN2.02. You can, however, still use an explicit Submit button object to mail a form to you from browsers. (See the section "E-mailing forms" earlier in this chapter.)

Because the `form.submit()` method does not trigger the form's `onsubmit` event handler, you must perform any presubmission processing and forms validation in the same script that ends with the `form.submit()` statement. You also do not want to interrupt the submission process after the script invokes the `form.submit()` method. Script statements inserted after one that invokes `form.submit()` — especially those that navigate to other pages or attempt a second submission — cause the first submission to cancel itself.

Related Item: `onsubmit` event handler.

Event handlers

onreset

Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

Immediately before a Reset button returns a form to its default settings, JavaScript sends a `reset` event to the form. By including an `onreset` event handler in the form definition, you can trap that event before the reset takes place.

A friendly way of using this feature is to provide a safety net for a user who accidentally clicks the Reset button after filling out a form. The event handler can run a function that asks the user to confirm the action.

The `onreset` event handler employs a technique that surfaced with Navigator 3: The event handler must evaluate to `return true` for the event to continue to the browser. This may remind you of the way `onmouseover` and `onmouseout` event handlers work for links and image areas. This requirement is far more useful here because your function can control whether the reset operation ultimately proceeds to conclusion.

Listing 21-4 demonstrates one way to prevent accidental form resets or submissions. Using standard Reset and Submit buttons as interface elements, the `<form>` object definition includes both event handlers. Each event handler calls its own function that offers a choice for users. Notice how each event handler includes the word `return` and takes advantage of the Boolean values that come back from the `confirm()` method dialog boxes in both functions.

Listing 21-4: The onreset and onsubmit Event Handlers

```
<html>
   <head>
      <title>Submit and Reset Confirmation</title>
      <script type="text/javascript">
      function allowReset() {
         return window.confirm("Go ahead and clear the form?");
      }
      function allowSend() {
         return window.confirm("Go ahead and mail this info?");
      }
      </script>
   </head>
   <body>
      <form method="POST" enctype="text/plain"
      action="mailto:trash4@dannyg.com" onreset="return allowReset()"
      onsubmit="return allowSend()">
         Enter your first name:<input type="text" name="firstName" />
         <p>Enter your last name:<input type="text" name="lastName" /></p>
         <p>Enter your address:<input type="text" name="address" /></p>
         <p>Enter your city:<input type="text" name="city" /></p>
         <p><input type="radio" name="gender" checked="checked" />Male <input
            type="radio" name="gender" />Female</p>
         <p><input type="checkbox" name="retired" />I am retired</p>
         <p><input type="reset" /> <input type="submit" /></p>
      </form>
   </body>
</html>
```

onsubmit

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

No matter how a form's data is actually submitted (by a user clicking a Submit button or by a script invoking the `form.submit()` method), you may want your JavaScript-enabled HTML document to perform some data validation on the user input, especially with text fields, before the submission heads for the server. You have the option of doing such validation while the user enters data (see Chapter 43 on the CD-ROM) or in batch mode before sending the data to the server (or both). The place to trigger this last-ditch data validation is the form's `onsubmit` event handler. Note, however, that this event fires only from a genuine Submit type `<input>` element and not from the form's `submit()` method.

When you define an `onsubmit` handler as an attribute of a `<form>` definition, JavaScript sends the `submit` event to the form just before it dashes off the data to the server. Therefore, any script or function that is the parameter of the `onsubmit` attribute executes before the data is actually submitted. Note that this event handler fires only in response to a genuine Submit-style button and not from a `form.submit()` method.

Any code executed for the `onsubmit` event handler must evaluate to an expression consisting of the word `return` plus a Boolean value. If the Boolean value is `true`, the submission executes as usual; if the value is `false`, no submission is made. Therefore, if your script performs some validation prior to submitting data, make sure that the event handler calls that validation function as part of a return statement (as shown in Listing 21-4).

Even after your `onsubmit` event handler traps a submission, JavaScript's security mechanism can present additional alerts to the user depending on the server location of the HTML document and the destination of the submission.

fieldset and legend Element Objects

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`align`		
`form`		

Syntax

Accessing `fieldset` or `legend` element object properties or methods:

```
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About these objects

The `fieldset` and `legend` elements go hand in hand to provide some visual context to a series of form controls within a form. Browsers that implement the `fieldset` element draw a rectangle around the document space occupied by the form controls nested inside the `fieldset` element (although MacIE5 drops the space into a debossed area on the page — a nice effect). The rectangle renders the full width of the body, unless its width is controlled by appropriate stylesheet properties (for example, `width`). To that rectangle is added a text label that is assigned via the `legend` element nested inside the `fieldset` element. (For MacIE5, the legend text is rendered just inside the debossed space.) None of this HTML-controlled grouping is necessary if you design a page layout that already provides graphical elements to group the form controls together.

Nesting the elements properly is essential to obtaining the desired browser rendering. A typical HTML sequence looks like the following:

```
<form>
<fieldset>
<legend>Legend Text</legend>
All your form controls and their labels go here.
</fieldset>
</form>
```

You can have more than one `fieldset` element inside a form. Each set has a rectangle drawn around it. This can help organize a long form into more easily digestible blocks of controls for users — yet the single form retains its integrity for submission to the server.

A `fieldset` element acts like any HTML container with respect to stylesheets and the inheritance thereof. For example, if you set the `color` style property of a `fieldset` element, the color affects the text of elements nested within; however, the color of the border drawn by the browser is unaffected. Assigning a color to the `fieldset` style's `border-color` property colors just the border and not the textual content of nested elements.

Note that the content of the `legend` element can be any HTML. Alternatively, you can assign a distinctive stylesheet rule to the `legend` element. If your scripts need to modify the text of the legend, you can accomplish this with the `innerText` (IE4+), `innerHTML` (IE4+, NN6+, Moz1+, Safari1+), or `nodeValue` (IE5+, W3C) properties of HTML element objects.

Only two element-specific properties are assigned to this object pair. The first is the `align` property of the `legend` object. This property matches the capabilities of the `align` attribute for the element as specified in the HTML 4.0 recommendation (albeit the property is deprecated in favor of stylesheet rules). MacIE5+ and WinIE5.5+ enable you to adjust this property on the fly (generally between your choices of "right" and "left") to alter the location of the legend at the top of the fieldset rectangle.

Because these elements are children of a `form` element, it makes sense that the DOM Level 2 specification supplies the read-only `form` property to both of these objects. That property returns a reference to the `form` element object that encloses either element. The `form` property for the `fieldset` and `legend` objects is implemented in WinIE6, MacIE5, and W3C browsers.

label Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	Methods	Event Handlers
form		
htmlFor		

Syntax

Accessing `label` element object properties or methods:

```
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About this object

With the push in the HTML 4.0 specification to provide context-oriented tags for just about every bit of content on the page, the W3C HTML working group filled a gap with respect to text that usually hangs in front of or immediately after `input`, `select`, and `textarea` form control elements. You use these text chunks as labels for the items to describe the purpose of the control. The only `input` element that had an attribute for its label was the button input type. But even the newer `button` element did away with that.

A `label` element enables you to surround a control's label text with a contextual tag. In addition, one of the element's attributes — `for` — enables you to associate the label with a particular form control element. In the HTML, the `for` attribute is assigned the ID of the control with which the label is associated. A `label` element can be associated with a form control if the form control's tag is contained between the `label` element's start and end tags.

At first glance, browsers do nothing special (from a rendering point of view) for a `label` element. But for some kinds of elements, especially checkbox and radio input type elements, browsers help restore to users a vital user-interface convention: clicking the label is the same as clicking the control. For text elements, focus events are passed to the text input element associated with the label. In fact, all events that are directed at a label bubble upward to the form control associated with it. The following page fragment demonstrates how `fieldset`, `legend`, and `label` elements look in a form consisting of two radio buttons:

```
<form ...>
<fieldset id="form1set1">
<legend id="form1set1legend">Choose the Desired Performance</legend>
<input type="radio" name="speed" id="speed1" />
   <label for="speed1">Fastest (lower quality)</label><br />
<input type="radio" name="speed" id="speed2" />
   <label for="speed2">Slower (best quality)</label>
</fieldset>
</form>
```

Even so, a `label` and its associated form control element do not have to be adjacent to each other in the source code. For example, you can have a label in one cell of a table row with the form control in another cell (in the same or different row).

Properties

`htmlFor`

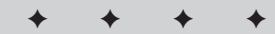
Value: Element object reference. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `htmlFor` property is the scripted equivalent of the `for` attribute of the `label` element. An acceptable value is a full reference to a form control element (`input`, `textarea`, or `select` element objects). It is highly unlikely that you would modify this property for an existing `label` element. However, if your script is creating a new `label` element (perhaps a replacement form), use this property to associate the label with a form control.

✦ ✦ ✦

CHAPTER 22

Button Objects

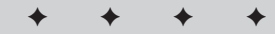


In This Chapter

Triggering action from a user's click of a button

Assigning hidden values to radio and checkbox buttons

Distinguishing between radio button families and their individual buttons



This chapter is devoted to those lovable buttons that invite users to initiate action and make choices with a single click of the mouse button. In this category fall the standard system-looking buttons with labels on them, as well as radio buttons and checkboxes. For such workhorses of the HTML form, these objects have a limited vocabulary of object-specific properties, methods, and event handlers.

I group together the button, submit, and reset objects for an important reason: They look alike yet they are intended for very different purposes. Knowing when to use which button is important — especially when to differentiate between the button and submit objects. Many a newcomer gets the two confused and winds up with scripting error headaches. That confusion won't happen to you by the time you finish this chapter.

The button Element Object, and the Button, Submit, and Reset Input Objects

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
form	click()	onclick
name		onmousedown
type		onmouseup
value		

Syntax

Accessing button object properties or methods:

```
(All)       [window.]document.formName.buttonName.property
| method([parameters])
(All)
[window.]document.formName.elements[index].property |
method([parameters])
```

```
(All)       [window.]document.forms[index].buttonName.property |
method([parameters])
(All)       [window.]document.forms["formName"].buttonName.property |
            method([parameters])
(All)       [window.]document.forms["formName"].elements[index].property |
            method([parameters])
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About these objects

Button objects generate standard, pushbutton-style user interface elements on the page, depending on the operating system on which the particular browser runs. In the early days, the browsers called upon the operating systems to generate these standard interface elements. In more recent versions, the browsers define their own look, albeit frequently still different for each operating system. More recently, the appearance of a button may also be influenced by browser-specific customizations that browser makers put into their products. Even so, any computer user will recognize a button when the browser produces it on the page.

Starting with IE4 and NN6, you have two ways to put standard buttons into a page. The first, and completely backward-compatible way, is to use `input` elements nested inside a `form` container. But a new HTML element, the `button` element, provides a slightly different way of specifying a button in a page, including the option of putting a button outside of a `form` (presumably for some client-side script execution, independent of form submission). From an HTML point of view, the difference between the two concerns itself with the way the label of the button is specified. With an `input` element, the string assigned to the `value` attribute becomes the label of the button; but a `button` element is a container (meaning with an end tag), whose content becomes the button's label. You can still assign a value to the `value` attribute, which, if a form contains the button, gets submitted to the server, independent of the label text.

Always give careful thought to the label that you assign to a button. Because a button initiates some action, make sure that the verb in the label clearly defines what happens after you click it. Also, take cues from experienced user interface designers who craft operating system and commercial software buttons: Be concise. If you find your button labels going longer than two or three words, reconsider the design of your page so that the user can clearly understand the purpose of any button from a shorter label.

Browsers automatically display a button sized to accommodate the label text. But only browsers that support stylesheets (IE4+ and W3C) allow you to control more visual aspects of the button, such as size, label font, and coloration. And, as for the position of the button on the page, buttons, as in all in-line elements, appear where they occur in the source code. You can, of course, use element positioning of recent browsers (Chapter 39 on the CD-ROM) to make a button appear wherever you want it. But if your pages run on multiple operating systems and generations of browsers, be aware that the appearance (and size) of a button will not be identical on all screens. Check out the results on as many platforms as possible.

Buttons in the Windows environment follow their normal behavior in that they indicate the focus with highlighted button-label text (usually with a dotted rectangle). Some newer browsers running on other operating systems offer this kind of highlighting and selection as a user option. IE5 provides additional `input` element features that prevent buttons from receiving this kind of visible focus.

The lone button object event handler that works on all browser versions is one that responds to a user clicking the pointer atop the button: the onclick event handler. Virtually all action surrounding a button object comes from this event handler. You rarely need to extract property values or invoke the click() method. NN4 and IE4 add events for the components of a click: mousedown and mouseup; and IE4+ and W3C browsers provide a plethora of user-initiated events for buttons.

Two special variants of the button object are the *submit* and *reset* button objects. With their heritages going back to early incarnations of HTML, these two button types perform special operations on their own. The submit-style button automatically sends the data within the same form object to the URL listed in the action attribute of the <form> definition. The method attribute dictates the format in which the button sends the data. Therefore, you don't have to script this action if your HTML page is communicating with a CGI program on the server.

If the form's action attribute is set to a mailto: URL, you must provide the page visitor with a Submit button to carry out the action. Setting the form's enctype attribute to text/plain is also helpful so that the form data arrives in a more readable form than the normal encoded name-value pairs. See "E-mailing forms" in Chapter 21 for details about submitting form content via e-mail.

The partner of the Submit button is the Reset button. This button, too, has special powers. A click of this button type restores all elements within the form to their default values. That goes for text objects, radio button groups, checkboxes, and selection lists. The most common application of the button is to clear entry fields of the last data entered by the user.

All that distinguishes these three types of buttons from each other in the <input> tag or <button> tag is the parameter of the type attribute. For buttons not intended to send data to a server, use the "button" style (this is the default value for the button element). Reserve "submit" and "reset" for their special powers.

If you want an image to behave like a button in all scriptable browsers, consider either associating a link with an image (see the discussion on the link object in Chapter 19) or creating a client-side image map (see the area object discussion in Chapter 20). But for IE4+ and W3C browsers, you can use the input element with a type attribute set to image (discussed later in this chapter).

Probably the biggest mistake scripters make with these buttons is using a Submit button to do the work of a plain button. Because these two buttons look alike, and the submit type of input element has a longer tradition than the button, confusing the two is easy. But if all you want is to display a button that initiates client-side script execution, use a plain button. The Submit button attempts to submit the form. If no action attribute is set, then the page reloads, and all previous processing and field entries are erased. The plain button does its job quietly without reloading the page (unless the script intentionally does so).

Properties

form

Value: Form object reference. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

A property of every input element object is a reference to the form element that contains the control. This property can be very convenient in a script when you are dealing with one form control that is passed as a parameter to the function and you want to either access another control in the same form or invoke a method of the form. An event handler of any

`input` element can pass `this` as the parameter, and the function can still get access to the form without having to hard-wire the script to a particular form name or document layout.

Related Items: `form` object.

name

Value: Identifier string. Read/Write (see text)
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

A button's name is fixed in the `input` or `button` element's `name` attribute and cannot be adjusted via scripting except in newer browsers. You may need to retrieve this property in a general-purpose function handler called by multiple buttons in a document. The function can test for a button name and perform the necessary statements for that button. If you change the name of the object, even a soft reload or window resize restores its original name.

Related Items: `name` property of all `form` elements.

type

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The precise value of the `type` property echoes the setting of the `type` attribute of the `<input>` or `<button>` tag that defines the object: `button`; `submit`; or `reset`.

value

Value: String. Read/Write (see text)
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Both `input` and `button` elements have the `value` attribute, which is represented by the `value` property in the object model. But the purpose of the attribute/property in the two elements differs. For the `input` element, the `value` property represents the label displayed on the button. For a `button` element, however, the label text is created by the HTML text between the start and end tags for the `button` element. When the `input` element has a `name` value associated with it, the name-value pair is submitted along with the form.

If you do not assign a `value` attribute to a reset or submit style button, the browsers automatically assign the labels `Reset` and `Submit` without assigning a value. A value property can be any string, including multiple words.

You can modify this text on the fly in a script, but some cautions apply. Browsers prior to IE4 and NN6 do not resize the width of the button to accommodate a new name that is longer or shorter than the original. Moreover, any soft reload or resize of the window restores the original label. IE4+ and NN6+, however, resize the button and reflow the page to meet the new space needs; the new label survives a window resizing, but not a soft reload of the page. Finally, IE4 for the Mac allows you to set this property, but it doesn't really stick.

Related Items: `value` property of text object.

Methods

click()

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

A button's `click()` method simulates, via scripting, the human action of clicking that button. Unfortunately, the method is highly unreliable in browsers prior to IE4 and NN4.

Related Items: `onclick` event handler.

Event handlers

onclick

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Virtually all button action takes place in response to the `onclick` event handler. A *click* is defined as a press and release of the mouse button while the screen pointer rests atop the button. The event goes to the button only after the user releases the mouse button.

For a Submit button, you should probably omit the `onclick` event handler and allow the form's `onsubmit` event handler to take care of last-minute data entry validation before sending the form. By triggering validation with the `onsubmit` event handler, your scripts can cancel the submission if something is not right (see the `form` object discussion in Chapter 21).

Listing 22-1 demonstrates not only the `onclick` event handler of a button but also how you may need to extract a particular button's `name` or `value` properties from a general-purpose function that services multiple buttons. In this case, each button passes its own object as a parameter to the `displayTeam()` function. The function then displays the results in an alert dialog box. A real-world application would probably use a more complex `if...else` decision tree to perform more sophisticated actions based on the button clicked (or use a `switch` construction on the `btn.value` expression for IE4+ and W3C browsers).

Listing 22-1: Three Buttons Sharing One Function

```
<html>
   <head>
      <title>Button Click</title>
      <script type="text/javascript">
      function displayTeam(btn) {
         if (btn.value == "Abbott") {
            alert("Abbott & Costello");
         }
         if (btn.value == "Rowan") {
            alert("Rowan & Martin");
         }
         if (btn.value == "Martin") {
            alert("Martin & Lewis");
         }
      }
      </script>
   </head>
```

Continued

Listing 22-1 *(continued)*

```
   <body>
      Click on your favorite half of a popular comedy team:
      <form>
         <input type="button" value="Abbott" onclick="displayTeam(this)" />
         <input type="button" value="Rowan" onclick="displayTeam(this)" />
         <input type="button" value="Martin" onclick="displayTeam(this)" />
      </form>
   </body>
</html>
```

Related Items: `button.onmousedown`, `button.onmouseup`, `form.onsubmit` event handlers.

onmousedown
onmouseup

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

More recent browsers have event handlers for the components of a `click` event: the `onmousedown` and `onmouseup` event handlers. These events fire in addition to the `onclick` event handler.

The system-level buttons provided by the operating system perform their change of appearance while a button is being pressed. Therefore, trapping for the components of a click action won't help you in changing the button's appearance via scripting. Remember that a user can roll the cursor off the button while the button is still down. When the cursor leaves the region of the button, the button's appearance returns to its unpressed look, but any setting you make with the `onmousedown` event handler won't undo itself with an `onmouseup` counterpart, even after the user releases the mouse button elsewhere. On the other hand, if you can precache a click-on and click-off sound, you can use these events to fire the respective sounds in response to the mouse button action.

Related Items: `button.onclick` event handler.

checkbox Input Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`checked`	`click()`†	`onclick`†
`form`†		
`name`†		
`type`		
`value`		

† See Button object.

Syntax

Accessing checkbox properties or methods:

```
(All)       [window.]document.formName.boxName.property | method([parameters])
(All)       [window.]document.formName.elements[index].property |
method([parameters])
(All)       [window.]document.forms[index].boxName.property |
method([parameters])
(All)       [window.]document.forms["formName"].boxName.property |
method([parameters])
(All)       [window.]document.forms["formName"].elements[index].property |
            method([parameters])
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

About this object

Checkboxes have a very specific purpose in modern graphical user interfaces: to toggle between "on" and "off" settings. As with a checkbox on a printed form, a mark in the box indicates that the label text is true or should be included for the individual who made that mark. When the box is unchecked or empty, the text is false or should not be included. If two or more checkboxes are physically grouped together, they should have no interaction: Each is an independent setting (see the discussion on the radio object for interrelated buttons).

I make these user interface points at the outset because, in order to present a user interface in your HTML pages consistent with the user's expectations based on exposure to other programs, you must use checkbox objects only for on/off choices that the user makes. Using a checkbox as an action button that, for example, navigates to another URL is not good form. Just as they do in a Windows or Mac dialog box, users make settings with checkboxes and radio buttons and initiate action by clicking a standard button or image map.

That's not to say that a checkbox object cannot perform some limited action in response to a user's click, but such actions are typically related to the context of the checkbox button's label text. For example, in some Windows and Macintosh dialog boxes, turning on a checkbox may activate a bunch of otherwise inactive settings elsewhere in the same dialog box. IE4+ and W3C browsers allow disabling (dimming) or hiding form elements, so a checkbox may control those visible attributes of related controls. Or, in a two-frame window, a checkbox in one frame may control whether the viewer is an advanced user. If so, the content in the other frame may be more detailed. Toggling the checkbox changes the complexity level of a document showing in the other frame (using different URLs for each level). The bottom line, then, is that you should use checkboxes for toggling between on/off settings. Provide regular buttons for users to initiate processing.

In the `<input>` tag for a checkbox, you can preset the checkbox to be checked when the page appears. Add the constant `checked` attribute to the definition. If you omit this attribute, the default, unchecked appearance rules. As for the checkbox label text, its definition lies outside the `<input>` tag. If you look at the way checkboxes behave in HTML browsers, this location makes sense: The label is not an active part of the checkbox (as it typically is in Windows and Macintosh user interfaces, where clicking the label is the same as clicking the box).

Naming a checkbox can be an important part of the object definition, depending on how you plan to use the information in your script or document. For forms whose content goes to a CGI program on the server, you must word the box name as needed for use by the CGI program, so that the program can parse the form data and extract the setting of the checkbox. For JavaScript client-side use, you can assign not only a name that describes the button, but also a value useful to your script for making `if...else` decisions or for assembling strings that are eventually displayed in a window or frame.

Properties

checked

Value: Boolean. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The simplest property of a checkbox reveals (or lets you set) whether or not a checkbox is checked. The value is `true` for a checked box and `false` for an unchecked box. To check a box via a script, simply assign `true` to the checkbox's `checked` property:

```
document.forms[0].boxName.checked = true;
```

Setting the `checked` property from a script does not trigger a `click` event for the checkbox object.

You may need an instance in which one checkbox automatically checks another checkbox elsewhere in the same or other form of the document. To accomplish this task, create an `onclick` event handler for the one checkbox and build a statement similar to the preceding one to set the other related checkbox to `true`. Don't get too carried away with this feature, however: For a group of interrelated, mutually exclusive choices, use a group of radio buttons instead.

If your page design requires that a checkbox be checked after the page loads, don't bother trying to script this checking action. Simply add the one-word `checked` attribute to the `<input>` tag. Because the `checked` property is a Boolean value, you can use its results as an argument for an `if` clause, as shown in the next example.

The simple example in Listing 22-2 passes a form object reference to the JavaScript function. The function, in turn, reads the `checked` value of the form's checkbox object (`checkThis.checked`) and uses its Boolean value as the test result for the `if...else` construction.

Listing 22-2: The checked Property as a Conditional

```
<html>
   <head>
      <title>Checkbox Inspector</title>
      <script type="text/javascript">
      function inspectBox(form) {
         if (form.checkThis.checked) {
            alert("The box is checked.");
         } else {
            alert("The box is not checked at the moment.");
         }
      }
      </script>
   </head>
```

```
   <body>
      <form>
         <input type="checkbox" name="checkThis" />Check here
         <p><input type="button" name="boxChecker" value="Inspect Box"
            onclick="inspectBox(this.form)" /></p>
      </form>
   </body>
</html>
```

Related Items: `defaultChecked`, `value` properties.

defaultChecked

Value: Boolean. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

If you add the `checked` attribute to the `<input>` definition for a checkbox, the `defaultChecked` property for that object is `true`; otherwise, the property is `false`. Having access to this property enables your scripts to examine checkboxes to see if they have been adjusted (presumably by the user, if your script does not set properties).

The function in Listing 22-3 is designed to compare the current setting of a checkbox against its default value. The `if` construction compares the current status of the box against its default status. Both are Boolean values, so they can be compared against each other. If the current and default settings don't match, the function goes on to handle the case in which the current setting is other than the default.

Listing 22-3: **Examining the defaultChecked Property**

```
function compareBrowser(thisBox) {
   if (thisBox.checked != thisBox.defaultChecked) {
      // statements about using a different set of HTML pages
   }
}
```

Related Items: `checked`, `value` properties.

type

Value: String (`checkbox`). Read-Only
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

Use the `type` property to help you identify a checkbox object from an unknown group of form elements.

Related Items: `form.elements` property.

value

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

A checkbox object's `value` property is a string of any text that you want to associate with the box. Note that the checkbox's `value` property is not the label, as it is for a regular button, but

hidden text associated with the checkbox. For instance, the label that you attach to a checkbox may not be worded in a way that is useful to your script. But if you place that useful wording in the `value` attribute of the checkbox tag, you can extract that string via the `value` property.

When a checkbox object's data is submitted to a CGI program, the `value` property is sent as part of the name-value pair if the box is checked (nothing about the checkbox is sent if the box is unchecked). If you omit the `value` attribute in your definition, the property always yields the string "on," which is submitted to a CGI program when the box is checked. From the JavaScript side, don't confuse this string with the on and off settings of the checkbox: Use the `checked` property to determine a checkbox's status.

The scenario for the skeleton HTML page in Listing 22-4 is a form with a checkbox whose selection determines which of two actions to follow for submission to the server. After the user clicks the Submit button, a JavaScript function examines the checkbox's `checked` property. If the property is `true` (the button is checked), the script sets the `action` property for the entire form to the content of the value property — thus influencing where the form goes on the server side. If you try this listing on your computer, the result you see varies widely with the browser version you use. For most browsers, you see some indication (an error alert or other screen notation) that a file with the name `primaryURL` or `alternateURL` doesn't exist. Unfortunately, WinIE5.5 does not display the name of the file that can't be opened. Try the example in another browser if you have one. The names and the error message come from the submission process for this demonstration.

Listing 22-4: **Adjusting a CGI Submission Action**

```
<html>
   <head>
      <title>Checkbox Submission</title>
      <script type="text/javascript">
      function setAction(form) {
         if (form.checkThis.checked) {
            form.action = form.checkThis.value;
         } else {
            form.action = "file://primaryURL";
         }
         return true;
      }
      </script>
   </head>
   <body>
      <form method="POST" action="">
         <input type="checkbox" name="checkThis"
         value="file://alternateURL" />Use alternate
         <p><input type="submit" name="boxChecker"
            onclick="return setAction(this.form)" /></p>
      </form>
   </body>
</html>
```

Related Items: `checked` property.

Methods

click()

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The intention of the `click()` method is to enact, via script, the physical act of clicking a checkbox (but without triggering the `onclick` event handler). However, your scripts are better served by setting the `checked` property so that you know exactly what the setting of the box is at any time.

Related Items: `checked` property; `onclick` event handler.

Event handlers

onclick

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Because users regularly click checkboxes, the objects have an event handler for the `click` event. Use this event handler only if you want your page (or variable values hidden from view) to respond in some way to the action of clicking a checkbox. Most user actions, as mentioned earlier, are initiated by clicking standard buttons rather than checkboxes, so be careful not to overuse event handlers in checkboxes.

The page in Listing 22-5 shows how to trap the click event in one checkbox to influence the visibility and display of other form controls. After you turn on the Monitor checkbox, a list of radio buttons for monitor sizes appears. Similarly, engaging the Communications checkbox makes two radio buttons visible. Your choice of radio button brings up one of two further choices within the same table cell (see Figure 22-1).

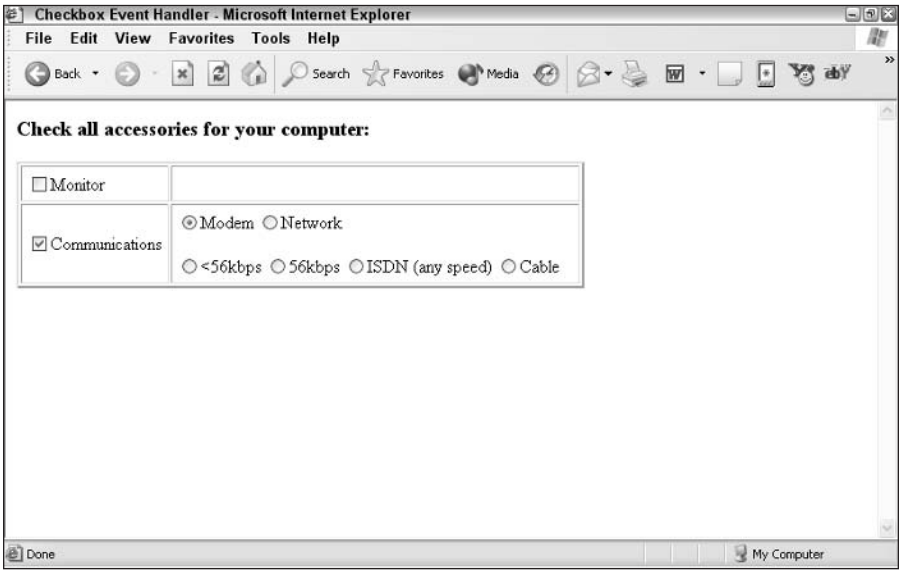


Figure 22-1: Clicking on button choices reveals additional relevant choices.

Notice how the `toggle()` function was written as a generalizable function. This function can accept a reference to any checkbox object and any related span. If five more groups like this were added to the table, no additional functions would be needed.

In the `swap()` function, an application of a nested `if...else` shortcut construction is used to convert the Boolean values of the `checked` property to the strings needed for the `display` style property. The nesting is used to allow a single statement to take care of two conditions: the group of buttons to be controlled and the `checked` property of the button invoking the function. This function is not generalizable, because it contains explicit references to objects in the document. The `swap()` function can be made generalizable, but due to the special relationships between pairs of span elements (meaning one has to be hidden while the other is displayed in its place), the function would require more parameters to fill in the blanks where explicit references are needed.

Listing 22-5: **A Checkbox and an onclick Event Handler**

```
<html>
   <head>
      <title>Checkbox Event Handler</title>
      <style type="text/css">
      #monGroup {visibility:hidden}
      #comGroup {visibility:hidden}
      </style>
      <script type="text/javascript">
      // toggle visibility of a main group spans
      function toggle(chkbox, group) {
         var visSetting = (chkbox.checked) ? "visible" : "hidden";
         document.getElementById(group).style.visibility = visSetting;
      }
      // swap display of communications sub group spans
      function swap(radBtn, group) {
         var modemsVisSetting = (group == "modems") ? ((radBtn.checked) ?
            "" : "none") : "none";
         var netwksVisSetting = (group == "netwks") ? ((radBtn.checked) ?
            "" : "none") : "none";
         document.getElementById("modems").style.display = modemsVisSetting;
         document.getElementById("netwks").style.display = netwksVisSetting;
      }
      </script>
   </head>
   <body>
      <form>
         <h3>Check all accessories for your computer:</h3>
         <table border="2" cellpadding="5">
            <tr>
               <td><input type="checkbox" name="monitor"
                  onclick="toggle(this, 'monGroup')" />Monitor</td>
               <td><span id="monGroup"><input type="radio"
                  name="monitorType" />15" <input type="radio"
                  name="monitorType" />17" <input type="radio"
                  name="monitorType" />21" <input type="radio"
                  name="monitorType" />&gt;21"</span></td>
            </tr>
```

```
            <tr>
               <td><input type="checkbox" name="comms"
                  onclick="toggle(this, 'comGroup')" />Communications</td>
               <td><span id="comGroup"><p><input type="radio" name="commType"
                  onclick="swap(this, 'modems')" />Modem <input type="radio"
                  name="commType" onclick="swap(this, 'netwks')" />Network</p>
                  <p><span id="modems" style="display:none"><input type="radio"
                  name="modemType" />&lt;56kbps <input type="radio"
                  name="modemType" />56kbps <input type="radio"
                  name="modemType" />ISDN (any speed) <input type="radio"
                  name="modemType" />Cable</span> <span id="netwks"
                  style="display:none"><input type="radio"
                  name="netwkType" />Ethernet 10Mbps (10-Base T) <input
                  type="radio" name="netwkType" />Ethernet 100Mbps (10/100)
                  <input type="radio" name="netwkType" />T1 or
                  greater</span>  </p></span></td>
            </tr>
         </table>
      </form>
   </body>
</html>
```

Related Items: `checkbox` mouse-related event handler.

radio Input Object

Properties	*Methods*	*Event Handlers*
See checkbox object.		

Syntax

Accessing radio object properties or methods:

```
(All)       [window.]document.formName.buttonGroupName[index].property |
            method([parameters])
(All)       [window.]document.formName.elements[index] [index].property |
            method([parameters])
(All)       [window.]document.forms[index]. buttonGroupName[index].property |
            method([parameters])
(All)       [window.]document.forms["formName"]. buttonGroupName[index].property
|
            method([parameters])
(All)       [window.]document.forms["formName"].elements[index].property |
            method([parameters])
(IE4+)      [window.]document.all.elemID[index].property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID")[index].property |
            method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

About this object

A radio button object is an unusual one within the body of JavaScript applications. In every other case of form control elements, one object equals one visual element on the screen. But a radio object actually consists of a group of radio buttons. Because of the nature of radio buttons — a mutually exclusive choice among two or more selections — a group always has multiple visual elements. All buttons in the group share the same name — which is how the browser knows to group buttons together and to let the clicking of a button deselect any other selected button within the group. Beyond that, however, each button can have unique properties, such as its `value` or `checked` property.

Use JavaScript array syntax to access information about an individual button within the button group. Look at the following example of defining a button group and see how to reference each button. This button group lets the user select a favorite member of the Three Stooges:

```
<form>
<b>Select your favorite Stooge:</b><br />
<input type="radio" name="stooges" value="Moe Howard" checked="checked" />Moe
<input type="radio" name="stooges" value="Larry Fine" />Larry
<input type="radio" name="stooges" value="Curly Howard" />Curly
<input type="radio" name="stooges" value="Shemp Howard" />Shemp
</form>
```

After this group displays on the page, the first radio button is preselected for the user. Only one property of a radio button object (`length`) applies to all members of the group. However, the other properties apply to individual buttons within the group. To access any button, use an array index value as part of the button group name. For example:

```
firstBtnValue = document.forms[0].stooges[0].value;  // "Moe Howard"
secondBtnValue = document.forms[0].stooges[1].value; // "Larry Fine"
```

Any time you access the checked, `defaultChecked`, `type`, or `value` property, you must point to a specific button within the group according to its order in the array (or, in IE4+/W3C, each button can also have a unique ID). The order of buttons in the group depends on the sequence in which the individual buttons are defined in the HTML document. In other words, to uncover the currently selected radio button, your script has to iterate through all radio buttons in the radio group. Examples of this come later in the discussion of this object.

Supplying a `value` attribute to a radio button can be very important in your script. Although the text label for a button is defined outside the `<input>` tag, the `value` attribute lets you store any string in the button's hip pocket. In the earlier example, the radio button labels were just first names, whereas the `value` properties were set in the definition to the full names of the actors. The values could have been anything that the script needed, such as birth dates, shoe sizes, URLs, or the first names again (because a script has no way to retrieve the labels except through `innerHTML` or node property access in more modern browsers). The point is that the `value` attribute should contain whatever string the script needs to derive from the selection made by the user. The `value` attribute contents are also what is sent to a CGI program on a server in a submit action for the form.

How you decide to orient a group of buttons on the screen is entirely up to your design and the real estate available within your document. You can string them in a horizontal row (as shown earlier), place `<br />` tags after each one to form a column, or do so after every other button to form a double column. Numeric order within the array is determined only by the order in which the buttons are defined in the source code, not by where they appear. To determine which radio button of a group is checked before doing processing based on that

choice, you need to construct a repeat loop to cycle through the buttons in the group (shown in the next example). For each button, your script examines the checked property.

Properties

checked

Value: Boolean. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Only one radio button in a group can be highlighted (checked) at a time (the browser takes care of highlighting and unhighlighting buttons in a group for you). That one button's `checked` property is set to `true`, whereas all others in the group are set to `false`.

Beginning with version 3 browsers, you can safely set the `checked` property of a radio button. By setting the `checked` property of one button in a group to `true`, all other buttons automatically uncheck themselves.

Listing 22-6 uses a repeat loop in a function to look through all buttons in the Stooges group in search of the checked button. After the loop finds the one whose checked property is `true`, it returns the value of the index. In one instance, that index value is used to extract the `value` property for display in the alert dialog box; in the other instance, the value helps determine which button in the group is next in line to have its `checked` property set to `true`.

Listing 22-6: **Finding the Selected Button in a Radio Group**

```
<html>
   <head>
      <title>Extracting Highlighted Radio Button</title>
      <script type="text/javascript">
      function getSelectedButton(buttonGroup){
         for (var i = 0; i < buttonGroup.length; i++) {
            if (buttonGroup[i].checked) {
               return i;
            }
         }
         return 0;
      }
      function fullName(form) {
         var i = getSelectedButton(form.stooges);
         alert("You chose " + form.stooges[i].value + ".");
      }
      function cycle(form) {
         var i = getSelectedButton(form.stooges);
         if (i+1 == form.stooges.length) {
            form.stooges[0].checked = true;
         } else {
            form.stooges[i+1].checked = true;
         }
      }
      </script>
   </head>
```

Continued

Listing 22-6 *(continued)*

```
   <body>
      <form>
         <b>Select your favorite Stooge:</b>
         <p><input type="radio" name="stooges" value="Moe Howard"
            checked="checked" />Moe <input type="radio" name="stooges"
            value="Larry Fine" />Larry <input type="radio" name="stooges"
            value="Curly Howard" />Curly <input type="radio" name="stooges"
            value="Shemp Howard" />Shemp</p>
         <p><input type="button" name="Viewer" value="View Full Name..."
            onclick="fullName(this.form)" /></p>
         <p><input type="button" name="Cycler" value="Cycle Buttons"
            onclick="cycle(this.form)" /></p>
      </form>
   </body>
</html>
```

Related Items: `defaultChecked` property.

defaultChecked

Value: Boolean. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

If you add the `checked` attribute to the `<input>` definition for a radio button, the `defaultChecked` property for that object is `true`; otherwise, the property is `false`. Having access to this property enables your scripts to examine individual radio buttons to see if they have been adjusted (presumably by the user, if your script does not perform automatic clicking).

In the script fragment of Listing 22-7, a function is passed a reference to a form containing the Stooges radio buttons. The goal is to see, in as general a way as possible (supplying the radio group name where needed), if the user changed the default setting. Looping through each of the radio buttons, you look for the one whose `checked` attribute is set in the `<input>` definition. With that index value (`i`) in hand, you then look to see if that entry is still checked. If not (notice the `!` negation operator), you display an alert dialog box about the change.

Listing 22-7: **Has a Radio Button Changed?**

```
function groupChanged(form) {
   for (var i = 0; i < form.stooges.length; i++) {
      if (form.stooges[i].defaultChecked) {
         if (!form.stooges[i].checked) {
            alert("This radio group has been changed.");
         }
      }
   }
}
```

Related Items: `checked`, `value` properties.

length

Value: Integer. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

A radio button group has *length* — the number of individual radio buttons defined for that group. Attempting to retrieve the length of an individual button yields a `null` value. The `length` property is valuable for establishing the maximum range of values in a repeat loop that must cycle through every button within that group. If you specify the `length` property to fill that value (rather than hard-wiring the value), the loop construction will be easier to maintain — as you make changes to the number of buttons in the group during page construction, the loop adjusts to the changes automatically.

Related Items: None.

name

Value: Identifier string. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `name` property, while associated with an entire radio button group, can be read only from individual buttons in the group, such as

```
btnGroupName = document.forms[0].groupName[2].name;
```

In that sense, each radio button element in a group inherits the name of the group. Your scripts have little need to extract the `name` property of a button or group. More often than not, you will hard-wire a button group's name into your script to extract other properties of individual buttons. Getting the `name` property of an object whose name you know is obviously redundant. But understanding the place of radio button group names in the scheme of JavaScript objects is important for all scripters.

Related Items: `value` property.

type

Value: String (`radio`). Read-Only
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

Use the `type` property to help identify a radio object from an unknown group of form elements.

Related Items: `form.elements` property.

value

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

As described earlier in this chapter for the checkbox object, the `value` property contains arbitrary information that you assign when mapping out the `<input>` definition for an individual radio button. Using this property is a handy shortcut to correlating a radio button label with detailed or related information of interest to your script or CGI program on a server. If you like, the `value` property can contain the same text as the label.

Related Items: `name` property.

Methods

click()

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The intention of the `click()` method is to enact, via a script, the physical act of clicking a radio button. However, you better serve your scripts by setting the `checked` properties of all buttons in a group so that you know exactly what the setting of the group is at any time.

Related Items: `checked` property; `onclick` event handler.

Event handlers

onclick

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Radio buttons, more than any user interface element available in HTML, are intended for use in making choices that other objects, such as submit or standard buttons, act upon later. You may see cases in Windows or Mac programs in which highlighting a radio button — at most — activates or brings into view additional, related settings (see Listing 22-5).

I strongly advise you not to use scripting handlers that perform significant actions at the click of any radio button. At best, you may want to use knowledge about a user's clicking of a radio button to adjust a global variable or `document.cookie` setting that influences subsequent processing. Be aware, however, that if you script such a hidden action for one radio button in a group, you must also script similar actions for others in the same group. That way, if a user changes the setting back to a previous condition, the global variable is reset to the way it was. JavaScript, however, tends to run fast enough so that a batch operation can make such adjustments after the user clicks a more action-oriented button.

Every time a user clicks one of the radio buttons in Listing 22-8, he or she sets a global variable to `true` or `false`, depending on whether the person is a Shemp lover. This action is independent of the action that is taking place if the user clicks on the View Full Name button. An `onunload` event handler in the `<body>` definition triggers a function that displays a message to Shemp lovers just before the page clears (click the browser's Reload button to leave the current page prior to reloading). Here I use an initialize function triggered by `onload` so that the current radio button selection sets the global value upon a reload.

Listing 22-8: An onclick Event Handler for Radio Buttons

```
<html>
   <head>
      <title>Radio Button onClick Handler</title>
      <script type="text/javascript">
      var ShempOPhile = false
      function initValue() {
         ShempOPhile = document.forms[0].stooges[3].checked;
      }
```

```
      function fullName(form) {
         for (var i = 0; i < form.stooges.length; i++) {
            if (form.stooges[i].checked) {
               break;
            }
         }
         alert("You chose " + form.stooges[i].value + ".");
      }
      function setShemp(setting) {
         ShempOPhile = setting;
      }
      function exitMsg() {
         if (ShempOPhile) {
            alert("You like SHEMP?");
         }
      }
      </script>
   </head>
   <body onload="initValue()" onunload="exitMsg()">
      <form>
         <b>Select your favorite Stooge:</b>
         <p><input type="radio" name="stooges" value="Moe Howard"
            checked="checked" onclick="setShemp(false)" />Moe <input
            type="radio" name="stooges" value="Larry Fine"
            onclick="setShemp(false)" />Larry <input type="radio"
            name="stooges" value="Curly Howard"
            onclick="setShemp(false)" />Curly <input type="radio"
            name="stooges" value="Shemp Howard"
            onclick="setShemp(true)" />Shemp</p>
         <p><input type="button" name="Viewer" value="View Full Name..."
            onclick="fullName(this.form)" /></p>
      </form>
   </body>
</html>
```

image Input Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
complete		
form†		
name†		
src		
type		

† See Button object.

Syntax

Accessing `image` input object properties or methods:

```
(All)        [window.]document.formName.imageName.property |
method([parameters])
(All)        [window.]document.formName.elements[index].property |
method([parameters])
(All)        [window.]document.forms[index].imageName.property |
method([parameters])
(All)        [window.]document.forms["formName"].imageName.property |
             method([parameters])
(All)        [window.]document.forms["formName"].elements[index].property |
             method([parameters])
(IE4+)       [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)   [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About this object

Browsers with fuller document object models include the `image` input element among scriptable objects. The `image` input object most closely resembles the button input object but replaces the `value` property (which defines the label for the button) with the `src` property, which defines the URL for the image that is to be displayed in the form control. This is a much simpler way to define a clickable image icon, for example, than the way required for compatibility with older browsers: wrapping an `img` element inside an `a` element so that you can use the `a` element's event handlers.

Although this element loads a regular Web image in the document, you have virtually no control over the image, which the `img` element provides. Be sure the rendering is as you predict.

Properties

`complete`

Value: Boolean. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `complete` property works as it does for an `img` element, reporting `true` if the image has finished loading. Otherwise the property returns `false`. Interestingly, there is no `onload` event handler for this object.

Related Items: `image.complete` property.

`src`

Value: URL string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Like the `img` element object, the `image` input element's `src` property controls the URL of the image being displayed in the element. The property can be used for image swapping in a form control, just as it is for a regular `img` element. Because the `image` input element has all necessary mouse event handlers available (for example, `onmouseover`, `onmouseout`, `onmousedown`) you can script rollovers, click-downs, or any other user interface technique that you feel is appropriate for your buttons and images. To adapt code written for link-wrapped images,

move the event handlers from the `a` element to the `image` input element, and make sure the name of the `image` input element is the same as your old `img` element.

Older browsers load images into an `image` input element, but no event handlers are recognized.

Related Items: `image.src` property.

type

Value: String (`image`). Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Use the `type` property to help you identify an image input object from an unknown group of form elements.

Related Items: `form.elements` property.

✦ ✦ ✦

CHAPTER 23

Text-Related Form Objects

In This Chapter

Capturing and modifying text field contents

Triggering action by entering text

Capturing individual keystroke events

The document object model for forms includes four text-related user interface objects — text, password, and hidden `input` element objects, plus the `textarea` element object. All four of these objects are used for entry, display, or temporary storage of text data. While all of these objects can have text placed in them by default as the page loads, scripts can also modify the contents of these objects. Importantly, all but the hidden objects retain their user- or script-modified content during a soft reload (for example, clicking the Reload button); hidden objects revert to their default values on all reloads.

A more obvious difference between the hidden object and the rest is that its invisibility removes it from the realm of user events and actions. Therefore, the range of scripted possibilities is much smaller for the hidden object.

The persistence of text and `textarea` object data through reloads (and window resizes) makes these objects prime targets for off-screen storage of data that may otherwise be stored temporarily in a cookie. If you create a frame with no size (for example, you set the `cols` or `rows` values of a `<frameset>` tag to let all visible frames occupy 100 percent of the space and assign the rest — `*` — to the hidden frame), you can populate the frame with fields that act as shopping cart information or other data holders. Therefore, if users have cookies turned off or don't usually respond affirmatively to cookie requests, your application can still make use of temporary client storage. The field contents may survive unloading of the page, but whether this happens and for how many navigations away from the page the contents last depends on the visitor's cache settings. If the user quits the browser or closes the browser window, the field entry is lost.

Text Input Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
defaultValue	select()	onafterupdate*
form		onbeforeupdate*
maxLength		onchange
name		onerrorupdate*
readOnly		onselect
size		
type		
value		

Syntax

Accessing text `input` object properties or methods:

```
(All)       [window.]document.formName.fieldName.property | method([parameters])
(All)       [window.]document.formName.elements[index].property |
method([parameters])
(All)       [window.]document.forms[index].fieldName.property |
method([parameters])
(All)       [window.]document.forms["formName"].fieldName.property |
            method([parameters])
(All)       [window.]document.forms["formName"].elements[index].property |
            method([parameters])
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

About this object

The text `input` object is the primary medium for capturing single-line, user-entered text. By default, browsers tend to display entered text in a monospaced font (usually Courier or a derivative) so that you can easily specify the width (`size`) of a field based on the anticipated number of characters that a user may put into the field. Until you get to IE4+ and W3C-compatible browsers, the font is a fixed size and always is left-aligned in the field. In those later browsers, stylesheets can control the font characteristics of a text field. If your design requires multiple lines of text, use the `textarea` object that comes later in this chapter.

Before the W3C DOM allowed dynamic modification of body content, a common practice was to use text objects to display results of a script calculation or other processing. Such fields may stand alone on a page or be part of a table.

Also prior to the W3C DOM, these fields could not be made fully write-protected, so it was easy to understand how a novice user may have become confused after he or she caused the text pointer or selection to activate a field used exclusively for output, simply by tabbing through a page.

Text object methods and event handlers use terminology that may be known to Windows users but not to Macintosh users. A field is said to have *focus* whenever the user clicks or tabs into the field. When a field has focus, either the text insertion pointer flashes, or any text in the field may be selected. Only one text object on a page can have focus at a time. The inverse user action — clicking or tabbing away from a text object — is called a *blur*. Clicking another object, whether it is another field or a button of any kind, causes a field that currently has focus to blur.

If you don't want the contents of a field to be changed by the user, you have three possibilities — depending on the vintage of browsers you need to support: forcing the field to lose focus; disabling the field; or setting the field's `readOnly` property.

The tactic that is completely backward compatible uses the following event handler in a field you want to protect:

```
onfocus="this.blur()"
```

Starting with IE4 and NN6, the object model provides a `disabled` property for form controls. Setting the property to `true` leaves the element visible on the page, but the user cannot access the control. The same browsers provide a `readOnly` property, which doesn't dim the field, but prevents typing in the field.

Text fields and events

Focus and blur also interact with other possible user actions to a text object: selecting and changing. *Selecting* occurs when the user clicks and drags across any text in the field; *changing* occurs when the user makes any alteration to the content of the field and then either tabs or clicks away from that field.

When you design event handlers for fields, be aware that a user's interaction with a field may trigger more than one event with a single action. For instance, clicking a field to select text may trigger both a `focus` and `select` event. If you have conflicting actions in the `onfocus` and `onselect` event handlers, your scripts can do some weird things to the user's experience with your page. Displaying alert dialog boxes, for instance, also triggers `blur` events, so a field that has both an `onselect` handler (which displays the alert) and an `onblur` handler gets a nasty interaction from the two.

As a result, be very judicious with the number of event handlers you specify in any text object definition. If possible, pick one user action that you want to use to initiate some JavaScript code execution and deploy it consistently on the page. Not all fields require event handlers — only those you want to perform some action as the result of user activity in that field.

Many newcomers also become confused by the behavior of the `change` event. To prevent this event from being sent to the field for every character the user types, any change to a field is determined only *after* the field loses focus by the user's clicking or tabbing away from it. At that point, instead of a `blur` event being sent to the field, only a `change` event is sent, triggering an `onchange` event handler if one is defined for the field. This extra burden of having to click or tab away from a field may entice you to shift any `onchange` event handler tasks to a separate button that the user must click to initiate action on the field contents.

Starting with version 4 browsers, text fields also have event handlers for keyboard actions, namely `onkeydown`, `onkeypress`, and `onkeyup`. With these event handlers, you can intercept keystrokes before the characters reach the text field. Thus, you can use keyboard events to prevent anything but numbers from being entered into a text box while the user types the characters.

To extract the current content of a text object, summon the property `document.`*`formName`*`.`*`fieldName`*`.value`. After you have the string value, you can use JavaScript's string object methods to parse or otherwise massage that text as needed for your script. If the field entry is a number and you need to pass that value to methods requiring numbers, you have to convert the text to a number with the help of the `parseInt()` or `parseFloat()` global functions.

Text Boxes and the Enter/Return Key

Early browsers established a convention that continues to this day. When a form consists of only one text box, a press of the Enter/Return key acts the same as clicking a Submit button for the form. You have probably experienced this many times when entering a value into a single search field of a form. Press the Enter/Return key, and the search request goes off to the server.

The flip side is that if the form contains more than one text box, the Enter/Return key does no submission from any of the text boxes (IE for the Mac and Safari are exceptions: they submit no matter how many text boxes there are). But with the advent of keyboard events, you can script this action (or the invocation of a client-side script) into any text boxes of the form you like. To make it work with all flavors of browsers capable of keyboard events requires a small conversion function that extracts the DOM-specific desired code from the keystroke. The following listing shows a sample page that demonstrates how to implement a function that inspects each keystroke from a text field and initiates processing if the key pressed is the Enter/Return key:

```
<html>
   <head>
      <title>Enter/Return Event Trigger</title>
      <script type="text/javascript">
      // Event object processor for NN4, IE4+, NN6
      function isEnterKey(evt) {
         evt = (evt) ? evt : (window.event) ? window.event : "";
         var theKey;
         if (evt) {
            theKey = (evt.which) ? evt.which : evt.keyCode;
         }
         return (theKey == 13);
      }

      function processOnEnter(fld, evt) {
         if (isEnterKey(evt)) {
            alert("Ready to do some work with the form.");
            return false;
         }
         return true;
      }
      </script>
   </head>
```

```
      <body>
         <h1>Enter/Return Event Trigger</h1>
         <hr />
         <form onsubmit="return false">
            Field 1: <input type="text" name="field1"
            onkeydown="return processOnEnter(this, event)" />
            Field 2: <input
            type="text" name="field2"
            onkeydown="return processOnEnter(this, event)" />
            Field 3: <input
            type="text" name="field3"
            onkeydown="return processOnEnter(this, event)" />
         </form>
      </body>
   </html>
```

Notice that to accommodate the NN4+ and W3C event models, a reference to the `event` object must be passed as a parameter to the processing function. For more details on event handling, see Chapter 25.

Properties

`defaultValue`

Value: String. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Though your users and your scripts are free to muck with the contents of a `text` object by assigning strings to the value property, you can always extract (and thus restore, if necessary) the string assigned to the text object in its `<input>` definition. The `defaultValue` property yields the string parameter of the `value` attribute.

Listings 23-1, 23-2, and 23-3 feature a form with only one text `input` element. The rules of HTML forms say that such a form submits itself if the user presses the Enter key whenever the field has focus. Such a submission to a form whose action is undefined causes the page to reload, thus stopping any scripts that are running at the time. `form` elements for these example listings contain an `onsubmit` event handler that both blocks the submission and attempts to trigger the text box `onchange` event handler to run the demonstration script. In some browsers, such as MacIE5, you may have to press the Tab key or click outside of the text box to trigger the `onchange` event handler after you enter a new value.

Listing 23-1 has a simple form with a single field that has a default value set in its tag. A function (`resetField()`) restores the contents of the page's lone field to the value assigned to it in the `<input>` definition. For a single-field page such as this, defining a `type="reset"` button or calling `form.reset()` works the same way because such buttons reestablish default values of all elements of a form. But if you want to reset only a subset of fields in a form, follow the example button and function in Listing 23-1.

Listing 23-1: **Resetting a Text Object to Default Value**

```
<html>
   <head>
      <title>Text Object DefaultValue</title>
      <script type="text/javascript">
      function upperMe(field) {
         field.value = field.value.toUpperCase();
      }
      function resetField(form) {
         form.converter.value = form.converter.defaultValue;
      }
      </script>
   </head>
   <body>
      <form onsubmit="window.focus(); return false">
         Enter lowercase letters for conversion to uppercase: <input
         type="text" name="converter" value="sample"
         onchange="upperMe(this)" /> <input type="button" value="Reset Field"
         onclick="resetField(this.form)" />
      </form>
   </body>
</html>
```

Related Items: `value` property.

`form`

Value: Form object reference. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

A property of every `input` element object is a reference to the `form` element that contains the control. This property can be very convenient in a script when you are dealing with one form control that is passed as a parameter to the function and you want to either access another control in the same form or invoke a method of the form. An event handler of any `input` element can pass `this` as the parameter, and the function can still get access to the form without having to hard-wire the script to a particular form name or document layout.

The following function fragment receives a reference to a text element as the parameter. The text element reference is needed to decide which branch to follow; then the form is submitted.

```
function setAction(fld) {
   if (fld.value.indexOf("@") != -1) {
      fld.form.action = "mailto:" + fld.value;
   } else {
      fld.form.action = "cgi-bin/normal.pl";
   }
   fld.form.submit();
}
```

Notice how this function doesn't have to worry about the form reference, because its job is to work with whatever form encloses the text field that triggers this function.

Related Items: `form` object.

maxLength

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `maxLength` property controls the maximum number of characters allowed to be typed into the field. There is no interaction between the `maxLength` and `size` properties. This value is normally set initially via the `maxlength` attribute of the `input` element.

Use The Evaluator (Chapter 13) to experiment with the `maxLength` property. The top text field has no default value, but you can temporarily set it to only a few characters and see how it affects entering new values:

```
document.forms[0].input.maxLength = 3;
```

Try typing this into the field to see the results of the change. To restore the default value, reload the page.

Related Items: `size` property.

name

Value: Identifier string. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Text object names are important for two reasons. First, if your HTML page submits information to CGI scripts, the input device passes the name of the text object along with the data to help the server program identify the data being supplied by the form. Second, you can use a text object's name in its reference within JavaScript coding. If you assign distinctive, meaningful names to your fields, these names will help you read and debug your JavaScript listings (and will help others follow your scripting tactics).

Be as descriptive about your text object names as you can. Borrowing text from the field's on-page label may help you mentally map a scripted reference to a physical field on the page. Like all JavaScript object names, text object names must begin with a letter and be followed by any number of letters or numbers. Avoid punctuation symbols with the exception of the very safe underscore character.

Although I urge you to use distinctive names for all objects you define in a document, you can make a case for assigning the same name to a series of interrelated fields — and JavaScript is ready to help. Within a single form, any reused name for the same object type is placed in an indexed array for that name. For example, if you define three fields with the name `entry`, the following statements retrieve the `value` property for each field:

```
data = document.forms[0].entry[0].value;
data = document.forms[0].entry[1].value;
data = document.forms[0].entry[2].value;
```

This construction may be useful if you want to cycle through all of a form's related fields to determine which ones are blank. Elsewhere, your script probably needs to know what kind of information each field is supposed to receive, so that it can process the data intelligently. I don't often recommend reusing object names, but you should be aware of how the object model handles them in case you need this construction. See "Form element arrays" in Chapter 21 for more details.

Consult Listing 23-2 later in this chapter, where I use the text object's name, `convertor`, as part of the reference when assigning a value to the field. To extract the name of a text object,

you can use the property reference. Therefore, assuming that your script doesn't know the name of the first object in the first form of a document, the statement is

```
var objectName = document.forms[0].elements[0].name;
```

Related Items: `form.elements` property; all other `form` element objects' `name` property.

readOnly

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

To display text in a text field yet prevent users from modifying it, newer browsers offer the `readOnly` property (and tag attribute). When set to `true`, the property prevents users from changing or removing the content of the text field. Unlike a disabled text field, a read-only text field looks just like an editable one.

For older browsers, you can partially simulate this behavior by including the following event handler in the `input` element:

```
onfocus="this.blur()"
```

The event handler approach is not foolproof, however, in that quick-fingered users may be able to change a field before the event handler completes its task. For NN4, you can also trap for any keyboard events and prevent them from putting characters in the field.

Use The Evaluator (Chapter 13) to set the bottom text box to be read-only. Begin by typing anything you want in the bottom text box. Then enter the following statement into the top text box:

```
document.forms[0].inspector.readOnly = true;
```

While existing text in the box is selectable (and therefore can be copied into the clipboard), it cannot be modified or removed.

Related Items: `disabled` property.

size

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Unless otherwise directed, a text box is rendered to accommodate approximately 20 characters of text for the font family and size assigned to the element's stylesheet. You can adjust this under script control (in case the `size` attribute of the tag wasn't enough) via the `size` property, whose value is measured in characters (not pixels). Be forewarned, however, that browsers don't always make completely accurate estimates of the space required to display a set number of characters. If you are setting the `maxlength` attribute of a text box, making the `size` one or two characters larger is often a safe bet.

Resize the bottom text box of The Evaluator (Chapter 13) by entering the following statements into the top text box:

```
document.forms[0].inspector.size = 20;
document.forms[0].inspector.size = 400;
```

Reload the page to return the size back to normal (or set the value to `80`).

Related Items: `maxLength` property.

type

Value: String (`text`). Read-Only
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

Use the `type` property to help you identify a text input object from an unknown group of form elements.

Related Items: `form.elements` property.

value

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

A text object's `value` property is the two-way gateway to the content of the field. A reference to an object's value property returns the string currently showing in the field. Note that all values coming from a text object are string values. If your field prompts a user to enter a number, your script may have to perform data conversion to the number-as-string value ("42" instead of plain, old 42) before a script can perform math operations on it. JavaScript tries to be as automatic about this data conversion as possible and follows some rules about it (see Chapter 27). If you see an error message that says a value is not a number (for a math operation), the value is still a string.

Your script places text of its own into a field for display to the user by assigning a string to the `value` property of a text object. Use the simple assignment operator. For example:

```
document.forms[0].ZIP.value = "90210";
```

JavaScript is more forgiving about data types when assigning values to a text object. JavaScript does its best to convert a value to a string on its way to a text object display. Even Boolean values get converted to their string equivalents `true` or `false`. Scripts can place numeric values into fields without a hitch. But remember that if a script later retrieves these values from the text object, they will come back as strings. About the only values that don't get converted are objects. They typically show up in text boxes as `[object]` or, in some browsers, a more descriptive label for the object.

Storing arrays in a field requires special processing. You need to use the `array.join()` method to convert an array into a string. Each array entry is delimited by a character you establish in the `array.join()` method. Later you can use the `string.split()` method to turn this delimited string into an array.

As a demonstration of how to retrieve and assign values to a text object, Listing 23-2 shows how the action in an `onchange` event handler is triggered. Enter any lowercase letters into the field and click out of the field. I pass a reference to the entire form object as a parameter to the event handler. The function extracts the value, converts it to uppercase (using one of the JavaScript string object methods), and assigns it back to the same field in that form.

Listing 23-2: **Getting and Setting a Text Object's Value**

```
<html>
   <head>
      <title>Text Object Value</title>
      <script type="text/javascript">
```

Continued

Listing 23-2 *(continued)*

```
      function upperMe(form) {
         inputStr = form.converter.value;
         form.converter.value = inputStr.toUpperCase();
      }
      </script>
   </head>
   <body>
      <form onsubmit="window.focus(); return false">
         Enter lowercase letters for conversion to uppercase: <input
         type="text" name="converter" value="sample"
         onchange="upperMe(this.form)" />
      </form>
   </body>
</html>
```

I also show two other ways to accomplish the same task, each one more efficient than the previous example. Both utilize the shortcut object reference to get at the heart of the text object. Listing 23-3 passes the text object — contained in the `this` reference — to the function handler. Because that text object contains a complete reference to it (out of sight, but there just the same), you can access the `value` property of that object and assign a string to that object's `value` property in a simple assignment statement.

Listing 23-3: **Passing a Text Object (as this) to the Function**

```
<html>
   <head>
      <title>Text Object Value</title>
      <script type="text/javascript">
      function upperMe(field) {
         field.value = field.value.toUpperCase();
      }
      </script>
   </head>
   <body>
      <form onsubmit="window.focus(); return false">
         Enter lowercase letters for conversion to uppercase: <input
         type="text" name="converter" value="sample"
         onchange="upperMe(this)" />
      </form>
   </body>
</html>
```

Yet another way is to deal with the field values directly in an embedded event handler — instead of calling an external function (which is easier to maintain because all scripts are grouped together in the Head). With the function removed from the document, the event handler attribute of the `<input>` tag changes to do all the work:

```
<input type="text" name="converter" value="sample"
   onchange="this.value = this.value.toUpperCase()" />
```

The right-hand side of the assignment expression extracts the current contents of the field and (with the help of the `toUpperCase()` method of the string object) converts the original string to all uppercase letters. The result of this operation is assigned to the `value` property of the field.

The application of the `this` keyword in the previous examples may be confusing at first, but these examples represent the range of ways in which you can use such references effectively. Using `this` by itself as a parameter to an object's event handler refers only to that single object — a text object in Listing 23-3. If you want to pass along a broader scope of objects that contain the current object, use the `this` keyword along with the outer object layer that you want. In Listing 23-2, I sent a reference to the entire form along by specifying `this.form` — meaning the form that contains "this" object, which is being defined in the line of HTML code.

At the other end of the scale, you can use similar-looking syntax to specify a particular property of the `this` object. Thus, in the last example, I zeroed in on just the `value` property of the current object being defined — `this.value`. Although the formats of `this.form` and `this.value` appear the same, the fact that one is a reference to an object and the other just a value can influence the way your functions work. When you pass a reference to an object, the function can read and modify properties of that object (as well as invoke its functions); but when the parameter passed to a function is just a property value, you cannot modify that value without building a complete reference to the object and its value.

Related Items: `form.defaultValue` property.

Methods

blur()

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Just as a camera lens blurs when it goes out of focus, a text object blurs when it loses focus — when someone clicks or tabs out of the field. Under script control, `blur()` deselects whatever may be selected in the field, and the text insertion pointer leaves the field. The pointer does not proceed to the next field in tabbing order, as it does if you perform a blur by tabbing out of the field manually.

The following statement invokes the `blur()` method on a text box named `vanishText`:

```
document.forms[0].vanishText.blur();
```

Related Items: `focus()` method; `onblur` event handler.

focus()

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

For a text object, having focus means that the text insertion pointer is flashing in that text object's field (having focus means something different for buttons in a Windows environment). Giving a field focus is like opening it up for human editing.

Setting the focus of a field containing text does not let you place the cursor at any specified location in the field. The cursor usually appears at the beginning of the text (although in WinIE4+, you can use the `TextRange` object to position the cursor wherever you want in the field, as shown in Chapter 35 on the CD-ROM). To prepare a field for entry to remove the existing text, use both the `focus()` and `select()` methods.

See Listing 23-4 for an example of an application of the `focus()` method in concert with the `select()` method.

Related Items: `select()` method; `onfocus` event handler.

`select()`

Returns: Nothing.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Selecting a field under script control means selecting all text within the text object. A typical application is one in which an entry validation script detects a mistake on the part of the user. After alerting the user to the mistake (via a `window.alert()` dialog box), the script finishes its task by selecting the text of the field in question. Not only does this action draw the user's eye to the field needing attention (especially important if the validation code is checking multiple fields), but it also keeps the old text there for the user to examine for potential problems. With the text selected, the next key the user presses erases the former entry.

Trying to select a text object's contents with a click of a button is problematic. One problem is that a click of the button brings the document's focus to the button, which disrupts the selection process. For more ensured selection, the script should invoke both the `focus()` and the `select()` methods for the field, in that order. No penalty exists for issuing both methods, and the extra insurance of the second method provides a more consistent user experience with the page.

Some versions of WinIE are known to exhibit anomalous (meaning buggy) behavior when using the technique of focusing and selecting a text field after the appearance of an alert dialog box. The fix is not elegant, but it works: inserting an artificial delay via the `setTimeout()` method before invoking a separate function that focuses and selects the field. Better-behaved browsers accept the workaround with no penalty.

Selecting a text object via script does *not* trigger the same `onselect` event handler for that object as the one that triggers if a user manually selects text in the field. Therefore, no event handler script is executed when a user invokes the `select()` method.

A click of the Verify button in Listing 23-4 performs a validation on the contents of the text box, making sure the entry consists of all numbers. All work is controlled by the `checkNumeric()` function, which receives a reference to the field needing inspection as a parameter. Because of the way the delayed call to the `doSelection()` function has to be configured, various parts of what will become a valid reference to the form are extracted from the field's and form's properties. If the validation (performed in the `isNumber()` function) fails, the `setSelection()` method is invoked after an artificial delay of zero milliseconds. As goofy as this sounds, this method is all that IE needs to recover from the display and closure of the alert dialog box. Because the first parameter of the `setTimeout()` method must be a string, the example assembles a string invocation of the `setSelection()` function via string versions of the form and field names. All that the `setSelection()` function does is focus and select the field whose reference is passed as a parameter. This function is now generalizable to work with multiple text boxes in a more complex form.

Listing 23-4: **Selecting a Field**

```
<html>
   <head>
      <title>Text Object Select/Focus</title>
```

```
      <script type="text/javascript">
      // general purpose function to see if a suspected numeric input is a
      // number
      function isNumber(inputStr) {
         for (var i = 0; i < inputStr.length; i++) {
            var oneChar = inputStr.charAt(i);
            if (oneChar < "0" || oneChar > "9") {
               alert("Please make sure entries are numbers only.");
               return false;
            }
         }
         return true;
      }

      function checkNumeric(fld) {
         var inputStr = fld.value;
         var fldName = fld.name;
         var formName = fld.form.name;
         if (isNumber(inputStr)) {
            // statements if true
         } else {
            setTimeout("doSelection(document." + formName + ". " + fldName +
               ")", 0);
         }
      }

      function doSelection(fld) {
         fld.focus();
         fld.select();
      }
      </script>
   </head>
   <body>
      <form name="entryForm" onsubmit="return false">
         Enter any positive integer: <input type="text" name="numeric" />
         <p><input type="button" value="Verify"
            onclick="checkNumeric(this.form.numeric)" /></p>
      </form>
   </body>
</html>
```

Related Items: `focus()` method; `onselect` event handler.

Event handlers

onafterupdate
onbeforeupdate
onerrorupdate

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

If you are using WinIE data binding on a text element, the element is subject to three possible events in the course of retrieving updated data. The `onbeforeupdate` and `onafterupdate` events fire immediately before and after (respectively) the update takes place. If an error occurs in the retrieval of data from the database, the `onerrorupdate` event fires.

All three events may be used for advisory purposes. For example, an `onafterupdate` event handler may temporarily change the font characteristics of the element to signify the arrival of fresh data. Or an `onerrorupdate` event handler may fill the field with hyphens because no valid data exists for the field. These events apply only to `input` elements of type text (meaning not password or hidden types).

Related Items: `dataFld`, `dataSrc` properties (Chapter 15).

onblur
onfocus
onselect

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

All three of these event handlers should be used only after you have a firm understanding of the interrelationships of the events that reach text objects. You must use extreme care and conduct lots of user testing before including more than one of these three event handlers in a text object. Because some events cannot occur without triggering others either immediately before or after (for example, an `onfocus` occurs immediately before an `onselect` if the field did not have focus before), whatever actions you script for these events should be as distinct as possible to avoid interference or overlap.

The `onselect` event handler does not work in Windows versions of NN through Version 4.

In particular, be careful about displaying modal dialog boxes (for example, `window.alert()` dialog boxes) in response to the `onfocus` event handler. Because the text field loses focus when the alert displays and then regains focus after the alert is closed, you can get yourself into a loop that is difficult to break out of. If you get trapped in this manner, try the keyboard shortcut for reloading the page (Ctrl+R or ⌘-R) repeatedly as you keep closing the dialog box window.

A question often arises about whether data-entry validation should be triggered by the `onblur` or `onchange` event handler. An `onblur` validation cannot be fooled, whereas an `onchange` one can be (the user simply doesn't change the bad entry as he or she tabs out of the field). What I don't like about the `onblur` way is it can cause a frustrating experience for a user who wants to tab through a field now and come back to it later (assuming your validation requires data be entered into the field before submission). As in Chapter 43's discussion (on the CD-ROM) about form data validation, I recommend using `onchange` event handlers to trigger immediate data checking and then using another last-minute check in a function called by the form's `onsubmit` event handler.

To demonstrate one of these event handlers, Listing 23-5 shows how you may use the window's status bar as a prompt message area after a user activates any field of a form. When the user tabs to or clicks on a field, the prompt message associated with that field appears in the status bar. In Figure 23-1, the user has tabbed to the second text box, which caused the status bar message to display a prompt for the field.

Listing 23-5: **The onfocus Event Handler**

```
<html>
   <head>
      <title>onfocus Event Handler</title>
      <script type="text/javascript">
      function prompt(msg) {
         window.status = "Please enter your " + msg + ".";
      }
      </script>
   </head>
   <body>
      <form>
         Enter your first name:<input type="text" name="firstName"
         onfocus="prompt('first name')" />
         <p>Enter your last name:<input type="text" name="lastName"
            onfocus="prompt('last name')" /></p>
         <p>Enter your address:<input type="text" name="address"
            onfocus="prompt('address')" /></p>
         <p>Enter your city:<input type="text" name="city"
            onfocus="prompt('city')" /></p>
      </form>
   </body>
</html>
```

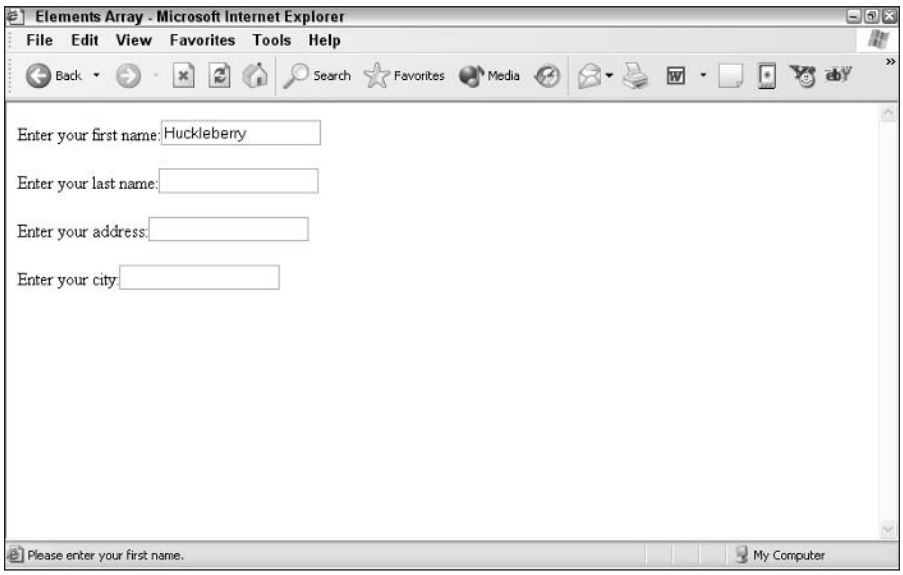


Figure 23-1: An onfocus event handler triggers a status bar display.

onchange

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Of all the event handlers for a text object, you will probably use the `onchange` handler the most in your forms (see Listing 23-6). This event is the one I prefer for triggering the validation of whatever entry the user just typed in the field. The potential hazard of trying to do only a batch-mode data validation of all entries before submitting an entire form is that the user's mental focus is away from the entry of a given field as well. When you immediately validate an entry, the user is already thinking about the information category in question. See Chapter 43 on the CD-ROM for more about data-entry validation.

In NN4 (only), if you have both `onchange` and any keyboard event handlers defined for the same text field tag, the `onchange` event handlers are ignored. This is not true for IE4+, NN6+, and other W3C browsers, where all events fire.

Listing 23-6: **Data Validation via an onchange Event Handler**

```
<html>
   <head>
      <title>Text Object Select/Focus</title>
      <script type="text/javascript">
      // general purpose function to see if a suspected numeric input is a
      // number
      function isNumber(inputStr) {
         for (var i = 0; i < inputStr.length; i++) {
            var oneChar = inputStr.substring(i, i + 1);
            if (oneChar < "0" || oneChar > "9") {
               alert("Please make sure entries are numbers only.");
               return false;
            }
         }
         return true;
      }

      function checkIt(form) {
         inputStr = form.numeric.value;
         if (isNumber(inputStr)) {
            // statements if true
         } else {
            form.numeric.focus();
            form.numeric.select();
         }
      }
      </script>
   </head>
   <body onsubmit="checkIt(this); return false">
      <form>
         Enter any positive integer: <input type="text" name="numeric"
         onchange="checkIt(this.form)" />
      </form>
   </body>
</html>
```

password Input Object

Properties	Methods	Event Handlers
See "Text Input Object"		

Syntax

See "Text Input Object"

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

About this object

A password-style field looks like a text object, but when the user types something into the field, only asterisks or bullets (depending on your operating system) appear in the field. For the sake of security, any password exchanges should be handled by a server-side program (CGI, Java servlet, and so on).

Scripts can treat a password object exactly like a text `input` object. This may lead a scripter to capture a user's Web site password for storage in the `document.cookie` of the client machine. A password object value property is returned in plain language, so that such a captured password would be stored in the cookie file the same way. Because a client machine's cookie file can be examined on the local computer (perhaps by a snoop during lunch hour), plain-language storage of passwords is a potential security risk. Instead, develop a scripted encryption algorithm for your page for reading and writing the password in the cookie. Most password-protected sites, however, usually have a server program (CGI, for example) encrypt the password prior to sending it back to the cookie.

See the text object discussion for the behavior of password object's properties, methods, and event handlers. The `type` property for this object returns `password`.

hidden Input Object

Properties	Methods	Event Handlers
See "Text Input Object"		

Syntax

See "Text Input Object"

Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

About this object

A hidden object is a simple string holder within a form object whose contents are not visible to the user of your Web page. Despite the long list of properties, methods, and event handlers that this input element type inherits by virtue of being an input element, you will be doing little with a `hidden` element beyond reading and writing its `value` property.

The `hidden` object plays a vital role in applications that rely on CGI programs on the server. Very often, the server has data that it needs to convey to itself the next time the client makes a submission (for example, a user ID captured at the application's login page). A CGI program can generate an HTML page with the necessary data hidden from the user but located in a field transmitted to the server at submit time.

Along the same lines, a page for a server application may present a user-friendly interface that makes data-entry easy for the user. But on the server end, the database or other application requires that the data be in a more esoteric format. A script located in the page generated for the user can use the `onsubmit` event handler to perform the last-minute assembly of user-friendly data into database-friendly data in a hidden field. When the CGI program receives the request from the client, it passes along the hidden field value to the database.

I am not a fan of the `hidden` object for use on client-side-only JavaScript applications. If I want to deliver with my JavaScript-enabled pages some default data collections or values, I do so in JavaScript variables and arrays as part of the script.

Because scripted changes to the contents of a hidden field are fragile (for example, a soft reload erases the changes), the only place you should consider making such changes is in the same script that submits a form to a CGI program or in a function triggered by an `onsubmit` event handler. In effect, you're just using the hidden fields as holding pens for the scripted data to be submitted. For more persistent storage, use the `document.cookie` property or genuine text fields in hidden frames, even if just for the duration of the visit to the page.

For information about the properties of the `hidden` object, consult the earlier listing for the text input object. The `type` property for this object returns `hidden`.

textarea Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`cols`	`createTextRange()`	`onafterupdate`†
`form`†	`select()`†	`onbeforeupdate`†
`name`†		`onchange`
`readOnly`†		`onerrorupdate`†
`rows`		
`type`†		
`wrap`		

† See "Text Input Object."

Syntax

Accessing `textarea` element object properties or methods:

```
(All)       [window.]document.formName.textareaName.property |
method([parameters])
(All)       [window.]document.formName.elements[index].property |
method([parameters])
(All)       [window.]document.forms[index].textareaName.property |
method([parameters])
(All)       [window.]document.forms["formName"].textareaName.property |
            method([parameters])
(All)       [window.]document.forms["formName"].elements[index].property |
            method([parameters])
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

About this object

Although not in the same HTML syntax family as other `<input>` elements of a form, a `textarea` object is indeed a form input element, providing multiple-line text input facilities. Although some browsers let you put a `textarea` element anywhere in a document, it really should be contained by a `form` element.

A `textarea` object closely resembles a text object, except for attributes that define its physical appearance on the page. Because the intended use of a `textarea` object is for multiple-line text input, the attributes include specifications for height (number of rows) and width (number of columns in the monospaced font). No matter what size you specify, the browser displays a textarea with horizontal and vertical scrollbars in older browsers; more recent browsers tend to be smarter about displaying scrollbars only when needed (although there are exceptions). Text entered in the textarea wraps within the visible rectangle of the field if you set the `wrap` attribute to virtual or physical in NN and `soft` or `hard` in IE; otherwise the text scrolls for a significant distance horizontally (the horizontal scrollbar appears when wrapping has the default off setting). This field is, indeed, a primitive text field by GUI computing standards in that font specifications made possible in newer browsers by way of stylesheets apply to all text in the box.

Use The Evaluator Sr. (Chapter 13) to play with the `cols` and `rows` property settings for the Results textarea on that page. Shrink the width of the textarea by entering the following statement into the top text box:

```
document.forms[0].output.cols = 30;
```

And make the textarea one row deeper:

```
document.forms[0].output.rows++;
```

All properties, methods, and event handlers of text objects apply to the `textarea` object. They all behave exactly the same way (except, of course, for the `type` property, which is `textarea`). Therefore, refer to the previous listings for the text object for scripting details for those items. Some additional properties that are unique to the `textarea` object are discussed next.

Carriage returns inside textareas

The three classes of operating systems supported by Netscape Navigator — Windows, Macintosh, and UNIX — do not agree about what constitutes a carriage return character in a text string. This discrepancy carries over to the `textarea` object and its contents on these platforms.

After a user enters text and uses Enter/Return on the keyboard, one or more unseen characters are inserted into the string. In the parlance of JavaScript's literal string characters, the carriage return consists of some combination of the newline (`\n`) and return (`\r`) character. The following table shows the characters inserted into the string for each operating system category.

Operating System	*Character String*
Windows	`\r\n`
Macintosh	`\r`
UNIX	`\n`

This tidbit is valuable if you need to remove carriage returns from a textarea for processing in a CGI or local script. The problem is that you obviously need to perform platform-specific operations on each. For the situation in which you must preserve the carriage return locations, but your server-side database cannot accept the carriage return values, I suggest you use the `string.escape()` method to URL-encode the string. The return character is converted to `%0D` and the newline character is converted to `%0A`. Of course these characters occupy extra character spaces in your database, so these additions must be accounted for in your database design.

As far as writing carriage returns into textareas, the situation is a bit easier. From NN3 and IE4 onward, if you specify any one of the combinations in the preceding table, all platforms know how to automatically convert the data to the form native to the operating system. Therefore, you can set the value of a `textarea` object to `1\r\n2\r\n3` in all platforms, and a columnar list of the numbers 1, 2, and 3 will appear in those fields. Or, if you URL-encoded the text for saving to a database, you can unescape that character string before setting the textarea value, and no matter what platform the visitor has, the carriage returns are rendered correctly. Upon reading those values again by script, you can see that the carriage returns are in the form of the platform (shown in the previous table).

Properties

`cols`
`rows`

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The displayed size of a `textarea` element is defined by its `cols` and `rows` attributes, which are represented in the object model by the `cols` and `rows` properties, respectively. Values for these properties are integers. For `cols`, the number represents the number of characters that

can be displayed without horizontal scrolling of the textarea; for `rows`, the number is the number of lines of text that can be displayed without vertical scrolling.

Related Items: `wrap` property.

wrap

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `wrap` property represents the `wrap` attribute, which, surprisingly, is not a W3C-sanctioned attribute as of HTML 4.01. In any case, IE4+ lets you adjust the property by scripting. Allowable string values are `soft`, `hard`, and `off`. The browser adds soft returns (the default in IE) to word-wrap the content, but no carriage return characters are actually inserted into the text. A setting for hard returns means that carriage return characters are added to the text (and would be submitted with the value to a server CGI). With wrap set to `off`, text continues to extend beyond the right edge of the textarea until the user manually presses the Enter/Return key.

Related Items: `cols` property.

Methods

createTextRange()

Returns: `TextRange` object.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The `createTextRange()` method for a `textarea` operates just as the `document.createTextRange()` method, except that the range consists of text inside the `textarea` element, apart from the regular body content. This version of the `TextRange` object comes in handy when you want a script to control the location of the text insertion pointer inside a `textarea` element for the user.

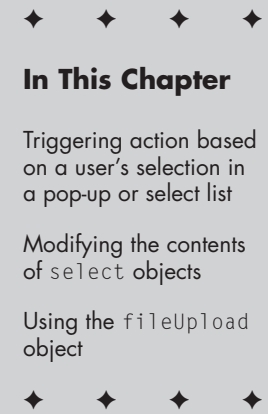
See the example for the `TextRange.move()` method in Chapter 5 to see how to control the text insertion pointer inside a `textarea` element.

Related Items: `TextRange` object (Chapter 35 on the CD-ROM).

✦ ✦ ✦

CHAPTER 24

Select, Option, and FileUpload Objects



Selection lists — whether in the form of pop-up menus or scrolling lists — are space-saving form elements in HTML pages. They enable designers to present a lot of information in a comparatively small space. At the same time, users are familiar with the interface elements from working in their own operating systems' preference dialog boxes and application windows.

However, selection lists are more difficult to script, especially in older browsers, because the objects themselves are complicated entities. Scripts find all the real data associated with the form control in `option` elements that are nested inside `select` elements. As you can see throughout this chapter, backward-compatible references necessary to extract information from a `select` element object and its `option` objects can get pretty long. The results, however, are worth the effort.

The other object covered in this chapter, the `fileUpload` input object, is frequently misunderstood as being more powerful than it actually is. It is, alas, not the great file transfer elixir desired by many page authors.

select Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
form†	add()	onchange
length	options[i].add()	
multiple	item()	
name†	namedItem()	
options[]	remove()	
selectedIndex	options[i].remove()	
size		
type		
value		

†See "Text Input Object" (Chapter 23).

Syntax

Accessing `select` element object properties:

```
(All)       [window.]document.formName.selectName.property |
method([parameters])
(All)       [window.]document.formName.elements[index].property |
method([parameters])
(All)       [window.]document.forms[index].selectName.property |
method([parameters])
(All)       [window.]document.forms["formName"].selectName.property |
            method([parameters])
(All)       [window.]document.forms["formName"].elements[index].property |
            method([parameters])
(IE4+)      [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)  [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

About this object

`select` element objects are perhaps the most visually interesting user interface elements among the standard built-in objects. In one format, they appear on the page as pop-up lists; in another format, they appear as scrolling list boxes. Pop-up lists, in particular, offer efficient use of page real estate for presenting a list of choices for the user. Moreover, only the choice selected by the user shows on the page, minimizing the clutter of unneeded verbiage.

Compared with other JavaScript objects, `select` objects are difficult to script — mostly because of the complexity of data that goes into a list of items. What the user sees as a `select` element on the page consists of both that element and `option` elements that contain the actual choices from which the user makes a selection. Some properties that are of value to scripters belong to the `select` object, while others belong to the nested `option` objects. For example, you can extract the number (index) of the currently selected option in the list — a property of the entire `select` object. To get the displayed text of the selected option, however, you must zero in further to extract the `text` property of a single option among all options defined for the object.

When you define a `select` object within a form, the construction of the `<select>...</select>` tag pair is easy to inadvertently mess up. First, most attributes that define the entire object — such as `name`, `size`, and event handlers — are attributes of the opening `<select>` tag. Between the end of the opening tag and the closing `</select>` tag are additional tags for each option to be displayed in the list. The following object definition creates a selection pop-up list containing three color choices:

```
<form>
<select name="RGBColors" onchange="changeColor(this)">
   <option selected="selected">Red</option>
   <option>Green</option>
   <option>Blue</option>
</select>
</form>
```

The indented formatting of the tags in the HTML document is not critical. I indent the lines of options merely for the sake of readability.

By default, a `select` element is rendered as a pop-up list. To make it appear as a scrolled list, assign an integer value greater than 1 to the `size` attribute to specify how many options should be visible in the list without scrolling — how tall the list's box should be, measured in lines. Because scrollbars in GUI environments tend to require a fair amount of space to display a minimum set of clickable areas (including sliding "thumbs"), you should set list-box style sizes to no less than 4. If that makes the list box too tall for your page design, consider using a pop-up menu instead.

Significant differences exist in the way each GUI platform presents pop-up menus. Because each browser sometimes relies on the operating system to display its native pop-up menu style (and sometimes the browser designers go their own way), considerable differences exist among the OS and browser platforms in the size of a given pop-up menu. What fits nicely within a standard window width of one OS may not fit in the window of another OS in a different browser. In other words, you cannot rely on any `select` object having a precise dimension on a page (in case you're trying to align a `select` object with an image).

In list-box form, you can set a `select` object to accept multiple, noncontiguous selections. Users typically accomplish such selections by holding down a modifier key (the Shift, Ctrl, or ⌘ key, depending on the operating system) while clicking additional options. To switch on this capability for a `select` object, include the `multiple` attribute constant in the definition.

For each entry in a list, your `<select>` tag definition must include an `<option>` tag plus the text as you want it to appear in the list. If you want a pop-up list to show a default selection when the page loads, you must attach a `selected` attribute to that item's `<option>` tag. Without this attribute, the default item may be empty or the first item, depending on the browser. (I go more in depth about this in the `option` object discussion later in this chapter.) You can also assign a string to each `option`'s `value` attribute. As with radio buttons, this value can be text other than the wording displayed in the list. In essence, your script can act on that "hidden" value rather than on the displayed text, such as letting a plain-language select listing actually refer to a complex URL. This string value is also the value sent to a CGI program (as part of the name-value pair) when the user submits the `select` object's form.

One behavioral aspect of the `select` object may influence your page design. The `onchange` event handler triggers immediately when a user makes a new selection in a pop-up list. If you prefer to delay any action until the user makes other settings in the form, omit an `onchange` event handler in the `select` object — but be sure to create a button that enables users to initiate an action governed by those user settings.

Modifying select options (NN3+, IE4+)

Script control gives you considerable flexibility in modifying the contents and selection of a `select` object. These powers are available only in NN3+ or IE4+ (I discuss a W3C approach a bit later in the chapter). Some of this flexibility is rather straightforward, such as changing the *`selectObj`*`.options[i].text` property to alter the display of a single-option entry. The situation gets tricky, though, when the number of options in the `select` object changes. Your choices include

- ✦ Removing an individual option (and thus collapsing the list)
- ✦ Reducing an existing list to a fewer number of options
- ✦ Removing all options
- ✦ Adding new options to a `select` object

select

To remove an option from the list, set the specific option to `null`. For example, if a list contains five items and you want to eliminate the third item altogether (reducing the list to four items), the syntax (from the `select` object reference) for doing that task is this:

```
selectObj.options[2] = null;
```

After this statement, `selectObj.options.length` equals 4.

In another scenario, suppose that a `select` object has five options in it and you want to replace it with one having only three options. You first must hard-code the `length` property to `3`:

```
selectObj.options.length = 3;
```

Then, set individual `text` and `value` properties for index values `0` through `2`.

Perhaps you want to start building a new list of contents by completely deleting the original list (without harming the `select` object). To accomplish this, set the `length` to `0`:

```
selectObj.options.length = 0;
```

From here, you have to create new options (as you do when you want to expand a list from, say, three to seven options). The mechanism for creating a new option involves an object constructor: `new Option()`. This constructor accepts up to four parameters, which enable you to specify the equivalent of an `<option>` tag's attributes:

- ✦ Text to be displayed in the option
- ✦ Contents of the option's `value` property
- ✦ Whether the item is the `defaultSelected` option (Boolean)
- ✦ Whether the item is selected (Boolean)

You can set any (or none) of these items as part of the constructor and return to other statements to set their properties. I suggest setting the first two parameters (leave the others blank) and then setting the `selected` property separately. The following is an example of a statement that creates a new, fifth entry, in a `select` object and sets both its displayed `text` and `value` properties:

```
selectObj.options[4] = new Option("Yahoo","http://www.yahoo.com");
```

To demonstrate all of these techniques, Listing 24-1 enables you to change the text of a `select` object — first by adjusting the text properties in the same number of options and then by creating an entirely new set of options. Radio button `onclick` event handlers trigger functions for making these changes — rare examples of when radio buttons can logically initiate visible action.

Listing 24-1: **Modifying select Options**

```
<html>
   <head>
      <title>Changing Options On The Fly</title>
      <script type="text/javascript" language="JavaScript">
      // flag to reload page for older NNs
      var isPreNN6 = (navigator.appName == "Netscape" &&
         parseInt(navigator.appVersion) <= 4);
```

```
// initialize color list arrays
plainList = new Array(6);
hardList = new Array(6);
plainList[0] = "cyan";
hardList[0] = "#00FFFF";
plainList[1] = "magenta";
hardList[1] = "#FF00FF";
plainList[2] = "yellow";
hardList[2] = "#FFFF00";
plainList[3] = "lightgoldenrodyellow";
hardList[3] = "#FAFAD2";
plainList[4] = "salmon";
hardList[4] = "#FA8072";
plainList[5] = "dodgerblue";
hardList[5] = "#1E90FF";

// change color language set
function setLang(which) {
   var listObj = document.forms[0].colors;
   // filter out old browsers
   if (listObj.type) {
      // find out if it's 3 or 6 entries
      var listLength = listObj.length;
      // save selected index
      var currSelected = listObj.selectedIndex;
      // replace individual existing entries
      for (var i = 0; i < listLength; i++) {
         if (which == "plain") {
            listObj.options[i].text = plainList[i];
         } else {
            listObj.options[i].text = hardList[i];
         }
      }
      if (isPreNN6) {
         history.go(0);
      } else {
         listObj.selectedIndex = currSelected;
      }
   }
}

// create entirely new options list
function setCount(choice) {
   var listObj = document.forms[0].colors;
   // filter out old browsers
   if (listObj.type) {
      // get language setting
      var lang = (document.forms[0].geekLevel[0].checked) ?
         "plain" : "hard";
      // empty options from list
      listObj.length = 0;
      // create new option object for each entry
      for (var i = 0; i < choice.value; i++) {
         if (lang == "plain") {
            listObj.options[i] = new Option(plainList[i]);
         } else {
            listObj.options[i] = new Option(hardList[i]);
```

Continued

Listing 24-1 *(continued)*

```
                }
            }
            listObj.options[0].selected = true;
            if (isPreNN6) {
               history.go(0);
            }
         }
      }
      </script>
   </head>
   <body>
      <h1>Flying Select Options</h1>
      <form>
         Choose a palette size: <input type="radio" name="paletteSize"
         value="3" onclick="setCount(this)" checked="checked" />Three <input
         type="radio" name="paletteSize" value="6"
         onclick="setCount(this)" />Six
         <p>Choose geek level: <input type="radio" name="geekLevel" value=""
            onclick="setLang('plain')" checked="checked" />Plain-language
            <input type="radio" name="geekLevel" value=""
            onclick="setLang('hard')" />Gimme hex-triplets!</p>
         <p>Select a color: <select name="colors">
               <option selected="selected">cyan</option>
               <option>magenta</option>
               <option>yellow</option>
            </select></p>
      </form>
   </body>
</html>
```

In an effort to make this code easily maintainable, the color choice lists (one in plain language, the other in hexadecimal triplet color specifications) are established as two separate arrays. Repeat loops in both large functions can work with these arrays no matter how big they get.

The first two radio buttons (see Figure 24-1) trigger the `setLang()` function. This function's first task is to extract a reference to the `select` object to make additional references shorter (just `listObj`). Then by way of the `length` property, you find out how many items are currently displayed in the list because you just want to replace as many items as are already there. In the repeat loop, you set the `text` property of the existing `select` options to corresponding entries in either of the two array listings.

In the second pair of radio buttons, each button stores a value indicating how many items should be displayed when the user clicks the button. This number is picked up by the `setCount()` function and is used in the repeat loop as a maximum counting point. In the meantime, the function finds the selected language radio button and zeros out the `select` object entirely. Options are rebuilt from scratch using the `new Option()` constructor for each option. The parameters are the corresponding display text entries from the arrays. Because none of these new options have other properties set (such as which one should be selected by default), the function sets that property of the first item in the list.

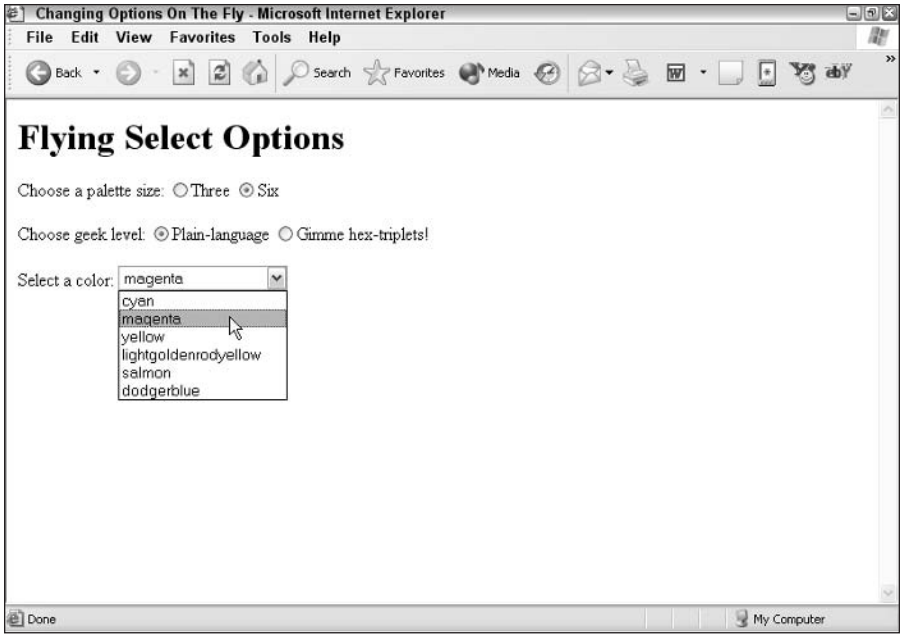


Figure 24-1: Radio button choices alter the contents of the `select` object on the fly.

Notice that both functions call `history.go(0)` for NN3 and NN4 browsers after setting up their `select` objects. The purpose of this call is to give these earlier Navigator versions an opportunity to resize the `select` object to accommodate the contents of the list. The difference in size here is especially noticeable when you switch from the six-color, plain-language list to any other list. Without resizing, some long items are not readable. IE4+ and NN6+, on the other hand, automatically redraw the page to the newly sized form element.

Modifying select options (IE4+)

Microsoft offers another way to modify `select` element options for IE4+, but the technique involves two proprietary methods of the `options` array property of the `select` object. Because I cover all other ways of modifying the `select` element in this section, I cover the IE way of doing things here as well.

The two `options` array methods are `add()` and `remove()`. The `add()` method takes one required parameter and one optional parameter. The required parameter is a reference to an `option` element object that your script creates in another statement (using the `document.createElement()` method). If you omit the second parameter to `add()`, the new `option` element is appended to the current collection of items. But you can also specify an index value as the second parameter. The index points to the position in the `options` array where the new item is to be inserted.

Listing 24-2 shows how to modify the two main functions from Listing 24-1 using the IE approach exclusively (changes and additions appear in bold). The script assumes that only IE browsers ever load the page (in other words, there is no filtering for browser brand here).

When replacing one set of options with another, there are two approaches demonstrated. In the first (the `setLang()` function), the replacements have the same number of items, so the length of existing options provides a counter and index value for the `remove()` and `add()` methods. But when the number of items may change (as in the `setCount()` function), a tight loop removes all items before they are added back via the `add()` method without a second parameter (items are appended to the list). The approach shown in Listing 24-2 has no specific benefit over that of Listing 24-1.

Listing 24-2: **Modifying select Options (IE4+)**

```
// change color language set
function setLang(which) {
   var listObj = document.forms[0].colors;
   var newOpt;
   // filter out old IE browsers
   if (listObj.type) {
      // find out if it's 3 or 6 entries
      var listLength = listObj.length;
      // save selected index
      var currSelected = listObj.selectedIndex;
      // replace individual existing entries
      for (var i = 0; i < listLength; i++) {
         newOpt = document.createElement("option");
         newOpt.text = (which == "plain") ? plainList[i] : hardList[i];
         listObj.options.remove(i);
         listObj.options.add(newOpt, i);
      }
      listObj.selectedIndex = currSelected;
   }
}

// create entirely new options list
function setCount(choice) {
   var listObj = document.forms[0].colors;
   var newOpt;
   // filter out old browsers
   if (listObj.type) {
      // get language setting
      var lang = (document.forms[0].geekLevel[0].checked) ? "plain" : "hard";
      // empty options from list
      while (listObj.options.length) {
         listObj.options.remove(0);
      }
      // create new option object for each entry
      for (var i = 0; i < choice.value; i++) {
         newOpt = document.createElement("option");
         newOpt.text = (lang == "plain") ? plainList[i] : hardList[i];
         listObj.options.add(newOpt);
      }
      listObj.options[0].selected = true;
   }
}
```

Modifying select options (W3C DOM)

Yet another approach is possible in browsers that closely adhere to the W3C DOM Level 2 standard. In NN6+, Moz1+, and Safari1+, for example, you can use the `add()` and `remove()` methods of the `select` element object. They work very much like the same-named methods for the `options` array in IE4+, but these are methods of the `select` element object itself. The other main difference between the two syntaxes is that the `add()` method does not use the index value as the second parameter but rather a reference to the `option` element object before which the new option is inserted. The second parameter is required, so to simply append the new item at the end of the current list, supply `null` as the parameter. Listing 24-3 shows the W3C-compatible version of the `select` element modification scripts shown in Listings 24-1 and 24-2. I highlight source code lines in bold that exhibit differences between the IE4+ and W3C DOM versions.

Listing 24-3: **Modifying select Options (W3C)**

```
// change color language set
function setLang(which) {
   var listObj = document.forms[0].colors;
   var newOpt;
   // filter out old IE browsers
   if (listObj.type) {
      // find out if it's 3 or 6 entries
      var listLength = listObj.length;
      // save selected index
      var currSelected = listObj.selectedIndex;
      // replace individual existing entries
      for (var i = 0; i < listLength; i++) {
         newOpt = document.createElement("option");
         newOpt.text = (which == "plain") ? plainList[i] : hardList[i];
         listObj.remove(i);
         listObj.add(newOpt, listObj.options[i]);
      }
      listObj.selectedIndex = currSelected;
   }
}

// create entirely new options list
function setCount(choice) {
   var listObj = document.forms[0].colors;
   var newOpt;
   // filter out old browsers
   if (listObj.type) {
      // get language setting
      var lang = (document.forms[0].geekLevel[0].checked) ? "plain" : "hard";
      // empty options from list
      while (listObj.options.length) {
         listObj.remove(0);
      }
      // create new option object for each entry
      for (var i = 0; i < choice.value; i++) {
         newOpt = document.createElement("option");
```

Continued

Listing 24-3 *(continued)*

```
            newOpt.text = (lang == "plain") ? plainList[i] : hardList[i];
            listObj.add(newOpt, null);
        }
        listObj.options[0].selected = true;
    }
}
```

As with the IE4 version, the W3C version offers no specific benefit over the original, backward-compatible approach. Choose the most modern one that fits the types of browsers you need to support with your page.

Properties

length

Value: Integer. Read/Write (see text)
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Like all JavaScript arrays, the `options` array has a `length` property of its own. But rather than having to reference the `options` array to determine its length, the `select` object has its own `length` property that you use to find out how many items are in the list. This value is the number of options in the object. A `select` object with three choices in it has a `length` property value of `3`.

In NN3+ and IE4+, you can adjust this value downward after the document loads. This is one way to decrease the number of options in a list. Setting the value to `0` causes the `select` object to empty but not disappear.

See Listing 24-1 for an illustration of the way you use the `length` property to help determine how often to cycle through the repeat loop in search of selected items. Because the loop counter, `i`, must start at `0`, the counting continues until the loop counter is one less than the actual length value (which starts its count with `1`).

Related Item: `options` property.

multiple

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `multiple` property represents the `multiple` attribute setting for a `select` element object. If the value is `true`, the element accepts multiple selections by the user (for example, Ctrl+clicking in Windows). If you want to convert a pop-up list into a multiple `select` pick list, you must also adjust the `size` property to direct the browser to render a set number of visible choices in the list.

The following statement toggles between single and multiple selections on a `select` element object whose `size` attribute is set to a value greater than `1`:

```
document.forms[0].mySelect.multiple = !document.forms[0].mySelect.multiple;
```

Related Item: `size` property.

options[*index*]

Value: Array of `option` element objects. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

You typically don't summon this property by itself. Rather, it is part of a reference to a specific option's properties (or methods in later browsers) within the entire `select` object. In other words, the `options` property is a kind of gateway to more specific properties, such as the value assigned to a single option within the list.

In newer browsers (IE4+ and W3C), you can reference individual options as separate HTML element objects. These references do not require the reference to the containing `form` or `select` element objects. For backward compatibility, however, I recommend you stick with the long references through the `select` objects.

I list the next several properties here in the `select` object discussion because they are backward-compatible with all browsers, including browsers that don't treat the `option` element as a distinct object. Be aware that all properties shown here that include `options[index]` as part of their references are also properties of the `option` element object in IE4+ and W3C browsers.

See Listings 24-1 through 24-3 for examples of how the `options` array references information about the options inside a `select` element.

Related Items: All `options[index].property` items.

options[*index*].defaultSelected

Value: Boolean. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

If your `select` object definition includes one option that features the `selected` attribute, that option's `defaultSelected` property is set to `true`. The `defaultSelected` property for all other options is `false`. If you define a `select` object that allows multiple selections (and whose `size` attribute is greater than `1`), however, you can define the `selected` attribute for more than one option definition. When the page loads, all items with that attribute are preselected for the user (even in noncontiguous groups).

The following statement preserves a Boolean value if the first option of the select list is the default selected item:

```
var zeroIsDefault = document.forms[0].listName.options[0].defaultSelected;
```

Related Item: `options[index].selected` property.

options[*index*].index

Value: Integer. Read-Only
Compatibility: WinIE3+, MacIE3+, NN6+, Moz1+, Safari1+

The `index` value of any single option in a `select` object likely is a redundant value in your scripting. Because you cannot access the option without knowing the index anyway (in brackets as part of the `options[index]` array reference), you have little need to extract the `index` value. The value is a property of the item just the same.

The following statement assigns the index integer of the first option of a `select` element named `listName` to a variable named `itemIndex`.

```
var itemIndex = document.forms[0].listName.options[0].index;
```

Related Item: `options` property.

options[*index*].selected

Value: Boolean. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

As mentioned earlier in the discussion of this object, better ways exist for determining which option a user selects from a list than looping through all options and examining the `selected` property. An exception to that "rule" occurs when you set up a list box to enable multiple selections. In this situation, the `selectedIndex` property returns an integer of only the topmost item selected. Therefore, your script needs to look at the `true` or `false` values of the `selected` property for each option in the list and determine what to do with the text or value data.

To accumulate a list of all items selected by the user, the `seeList()` function in Listing 24-4 systematically examines the `options[index].selected` property of each item in the list. The text of each item whose `selected` property is `true` is appended to the list. I add the "`\n`" inline carriage returns and spaces to make the list in the alert dialog box look nice and indented. If you assign other values to the `value` attributes of each option, the script can extract the `options[index].value` property to collect those values instead.

Listing 24-4: **Cycling through a Multiple-Selection List**

```
<html>
   <head>
      <title>Accessories List</title>
      <script type="text/javascript">
      function seeList(form) {
         var result = "";
         for (var i = 0; i < form.accList.length; i++) {
            if (form.accList.options[i].selected) {
               result += "\n  " + form.accList.options[i].text;
            }
         }
         alert("You have selected:" + result);
      }
      </script>
   </head>
   <body>
      <form>
         <p>Control/Command-click on all accessories you use: <select
            name="accList" size="9" multiple="multiple">
               <option selected="selected">Color Monitor</option>
               <option>Modem</option>
               <option>Scanner</option>
               <option>Laser Printer</option>
               <option>Tape Backup</option>
               <option>MO Drive</option>
               <option>Video Camera</option>
            </select></p>
         <p><input type="button" value="View Summary..."
            onclick="seeList(this.form)" /></p>
      </form>
   </body>
</html>
```

Related Items: `options[index].text`, `options[index].value`, `selectedIndex` properties.

options[*index*].text

Value: String. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `text` property of an option is the text of the item as it appears in the list. If you can pass that wording along with your script to perform appropriate tasks, this property is the one you want to extract for further processing. But if your processing requires other strings associated with each option, assign a `value` attribute in the definition and extract the `options[index].value` property (see Listing 24-6).

To demonstrate the `text` property of an option, Listing 24-5 applies the text from a selected option to the `document.bgColor` property of a document in the current window. The color names are part of the collection built into all scriptable browsers; fortunately, the values are case-insensitive so that you can capitalize the color names displayed and assign them to the property.

Listing 24-5: Using the options[*index*].text Property

```
<html>
   <head>
      <title>Color Changer 1</title>
      <script type="text/javascript">
      function seeColor(form) {
         var newColor =
            (form.colorsList.options[form.colorsList.selectedIndex].text);
         document.bgColor = newColor;
      }
      </script>
   </head>
   <body>
      <form>
         <p>Choose a background color: <select name="colorsList">
               <option selected="selected">Gray</option>
               <option>Lime</option>
               <option>Ivory</option>
               <option>Red</option>
            </select></p>
         <p><input type="button" value="Change It"
            onclick="seeColor(this.form)" /></p>
      </form>
   </body>
</html>
```

Related Item: `options[index].value` property.

options[*index*].value

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

In many instances, the words in the options list appear in a form that is convenient for the document's users but inconvenient for the scripts behind the page. Rather than set up an elaborate lookup routine to match the `selectedIndex` or `options[index].text` values with

the values your script needs, you can easily store those values in the `value` attribute of each `<option>` definition of the `select` object. You can then extract those values as needed.

You can store any string expression in the `value` attributes. That includes URLs, object properties, or even entire page descriptions that you want to send to a `parent.frames[index].document.write()` method.

Starting with IE4 and W3C browsers, the `select` element object itself has a `value` property that returns the `value` property of the selected option. But for backward compatibility, be sure to use the longer approach shown in the example in Listing 24-6.

Listing 24-6 requires the option text that the user sees to be in familiar, multiple-word form. But to set the color using the browser's built-in color palette, you must use the one-word form. Those one-word values are stored in the `value` attributes of each `<option>` definition. The function then reads the `value` property, assigning it to the `bgColor` of the current document. If you prefer to use the hexadecimal triplet form of color specifications, those values are assigned to the `value` attributes (`<option value="#e9967a">Dark Salmon`).

Listing 24-6: **Using the options[*index*].value Property**

```
<html>
   <head>
      <title>Color Changer 2</title>
      <script type="text/javascript">
      function seeColor(form) {
         var newColor =
            (form.colorsList.options[form.colorsList.selectedIndex].value);
         document.bgColor = newColor;
      }
      </script>
   </head>
   <body>
      <form>
         <p>Choose a background color: <select name="colorsList">
            <option selected="selected" value="cornflowerblue">
            Cornflower Blue</option>
            <option value="darksalmon">Dark Salmon</option>
            <option value="lightgoldenrodyellow">
            Light Goldenrod Yellow</option>
            <option value="seagreen">Sea Green</option>
            </select></p>
         <p><input type="button" value="Change It"
            onclick="seeColor(this.form)" /></p>
      </form>
   </body>
</html>
```

Related Item: `options[index].text` property.

selectedIndex

Value: Integer. Read/Write
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

When a user clicks a choice in a selection list, the `selectedIndex` property changes to a zero-based number corresponding to that item in the list. The first item has a value of `0`. This information is valuable to a script that needs to extract the value or text of a selected item for further processing.

You can use this information as a shortcut to getting at a selected option's properties. To examine a `select` object's `selected` property, rather than cycling through every option in a repeat loop, use the object's `selectedIndex` property to fill in the index value for the reference to the selected item. The wording gets kind of long; but from an execution standpoint, this methodology is much more efficient. Note, however, that when the `select` object is a multiple-style, the `selectedIndex` property value reflects the index of only the topmost item selected in the list.

To script the selection of a particular item, assign an integer value to the `select` element object's `selectedIndex` property, as shown in Listings 24-1 through 24-3.

In the `inspect()` function of Listing 24-7, notice that the value inside the `options` property index brackets is a reference to the object's `selectedIndex` property. Because this property always returns an integer value, it fulfills the needs of the index value for the `options` property. Therefore, if you select Green in the pop-up menu, `form.colorsList.selectedIndex` returns a value of `1`; that reduces the rest of the reference to `form.colorsList.options[1].text`, which equals "Green."

Listing 24-7: **Using the selectedIndex Property**

```
<html>
   <head>
      <title>Select Inspector</title>
      <script type="text/javascript">
      function inspect(form) {
         alert(form.colorsList.options[form.colorsList.selectedIndex].text);
      }
      </script>
   </head>
   <body>
      <form>
         <p><select name="colorsList">
               <option selected="selected">Red</option>
               <option value="Plants">Green</option>
               <option>Blue</option>
            </select></p>
         <p><input type="button" value="Show Selection"
            onclick="inspect(this.form)" /></p>
      </form>
   </body>
</html>
```

Related Item: `options` property.

size

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The size property represents the size attribute setting for a select element object. You can modify the integer value of this property to change the number of options that are visible in a pick list without having to scroll.

The following statement uses the size property to set the number of visible items to 5:

```
document.forms[0].mySelect.size = 5;
```

Related Item: multiple property.

type

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

Use the type property to help you identify a select object from an unknown group of form elements. The precise string returned for this property depends on whether the select object is defined as a single- (select-one) or multiple- (select-multiple) type.

Related Item: form.elements property.

value

Value: String. Read/Write (see text)
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The more recent browsers (and the W3C DOM) provide a value property for the select element object. This property returns the string assigned to the value attribute (or value property) of the currently selected option element. If you do not assign a string to the attribute or property, the value property returns an empty string. For these browser generations, you can use this shortcut reference to the select element object's value property instead of the longer version that requires a reference to the selectedIndex property and the options array of the element object.

The seeColor() function in Listing 24-6 that accesses the chosen value the long way can be simplified for newer browsers only with the following construction:

```
function seeColor(form) {
   document.bgColor = form.colorsList.value;
}
```

Related Item: options[*index*].value property.

Methods

add(newOption*ElementRef*[, *index*])
add(newOption*ElementRef*, *optionElementRef*)
remove(index)

Returns: Nothing.
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

These methods represent the W3C approach to adding and removing option elements from a selection. The first parameter to each of the add() methods is the new option element object to be added to the selection. The second parameters differ due to variances in IE and

other W3C browsers. The first version of `add()` is the IE version, which allows you to specify an optional index position for the new option; the option is placed just before the index position or it is appended to the end of the selection list if no index is provided. The W3C approach is represented by the second `add()` method, which requires an `option` object reference as the second parameter. This reference is to an option already in the selection list; the new option is added just before the option or it is appended to the end of the selection list if `null` is passed as the second parameter.

The `remove()` method requires the index of the option to be removed, and simply removes the option from the selection list.

`options[index].add(elementRef[, index])`
`options[index].remove()`

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

These two IE-specific methods belong to the `options` array property of a `select` element object. See the discussion at the opening of the `select` element object earlier in this chapter to see how to use these methods and their counterparts in other browser versions and object models.

`item(index)`
`namedItem("optionID")`

Returns: `option` element reference.
Compatibility: WinIE5+, MacIE5+, NN-, Moz-, Safari-

The `item()` and `namedItem()` methods are IE-specific convenience methods that access `option` element objects nested inside a `select` object. In a sense, they provide shortcuts to referencing nested options without having to use the `options` array property and the indexing within that array.

The parameter for the `item()` method is an index integer value. For example, the following two statements refer to the same `option` element object:

```
document.forms[0].mySelect.options[2]
document.forms[0].mySelect.item(2)
```

If your script knows the ID of an `option` element, it can use the `namedItem()` method, supplying the string version of the ID as the parameter, to return a reference to that option element.

The following statement assigns an option element reference to a variable:

```
var oneOption = document.forms[0].mySelect.namedItem("option3_2");
```

Related Item: `options` property.

Event handlers

`onchange`

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

As a user clicks a new choice in a `select` object, the object receives a `change` event that the `onchange` event handler can capture. In examples earlier in this section (Listings 24-6 and 24-7, for example), the action is handed over to a separate button. This design may make sense in

some circumstances, especially when you use multiple `select` lists or any list box. (Typically, clicking a list box item does not trigger any action that the user sees.) But for most pop-up menus, triggering the action when the user makes a choice is desirable.

To bring a pop-up menu to life, add an `onchange` event handler to the `<select>` definition. If the user makes the same choice as previously selected, the `onchange` event handler is not triggered. In this case, you can still trigger an action via the `onclick` event handler; but this event works for the `select` object only in IE4+ and W3C browsers.

Listing 24-8 is a version of Listing 24-6 that invokes all action as the result of a user making a selection from the pop-up menu. The `onchange` event handler in the `<select>` tag replaces the action button. For this application — when you desire a direct response to user input — an appropriate method is to have the action triggered from the pop-up menu rather than by a separate action button.

Notice two other important changes. First, the `select` element now contains a blank first option. When a user visits the page, nothing is selected yet, so you should present a blank option to encourage the user to make a selection. The function also makes sure that the user selects one of the color-valued items before it attempts to change the background color.

Second, the `body` element contains an `onunload` event handler that resets the form. The purpose behind this is that if the user navigates to another page and uses the Back button to return to the page, the script-adjusted background color does not persist. I recommend you return the `select` element to its original setting. Unfortunately, the reset does not stick to the form in IE4 and IE5 for Windows (although this problem appears to be repaired in IE5.5+). Another way to approach this issue is to use the `onload` event handler to invoke `seeColor()`, passing as a parameter a reference to the `select` element. Thus, if the `select` element choice persists, the background color is adjusted accordingly after the page loads.

Listing 24-8: **Triggering a Color Change from a Pop-Up Menu**

```
<html>
   <head>
      <title>Color Changer 2</title>
      <script type="text/javascript">
      function seeColor(list) {
         var newColor = (list.options[list.selectedIndex].value);
         if (newColor) {
            document.bgColor = newColor;
         }
      }
      </script>
   </head>
   <body onunload="document.forms[0].reset()">
      <form>
         <p>Choose a background color: <select name="colorsList"
            onchange="seeColor(this)">
               <option selected="selected" value=""></option>
               <option value="cornflowerblue">Cornflower Blue</option>
               <option value="darksalmon">Dark Salmon</option>
               <option value="lightgoldenrodyellow">
               Light Goldenrod Yellow</option>
               <option value="seagreen">Sea Green</option>
            </select></p>
      </form>
   </body>
</html>
```

option Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	Methods	Event Handlers
defaultSelected		
form†		
label		
selected		
text		
value		

†See "Text Input Object" (Chapter 23).

Syntax

Accessing `option` object properties:

```
(All)          [window.]document.formName.selectName.options[index].property |
               method([parameters])
(All)          [window.]document.formName.elements[index].options[index].property
|
                method([parameters])
(All)          [window.]document.forms[index].selectName.options[index].property
|
               method([parameters])
(All)
[window.]document.forms["formName"].selectName.options[index].property |
               method([parameters])
(All)
[window.]document.forms["formName"].elements[index].options[index].
               property | method([parameters])
(IE4+)         [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)   [window.]document.getElementById("elemID").property |
               method([parameters])
(W3C)          [window.]document.forms[index].selectName.item(index).property |
                method([parameters])
(W3C)          [window.]document.forms["formName"].selectName.namedItem(elemID).
               property | method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

About this object

`option` elements are nested inside `select` elements. Each option represents an item in the list of choices presented by the `select` element. Properties of the `option` element object let scripts inspect whether a particular option is currently selected or is the default selection. Other properties enable you to get or set the hidden value associated with the option as well as the visible text. For more details about the interaction between the `select` and `option` element objects, see the discussion about the `select` object earlier in this chapter as well as the discussion of the properties and methods associated with the `options` array returned by the `select` object's `options` property.

I discuss all backward-compatible `option` object properties (`defaultSelected`, `selected`, `text`, and `value`) among the `options` property descriptions in the `select` object section. The only items listed in this section are those that are unique to the `option` element object defined in newer browsers.

In NN3+ and IE4+, there is a provision for creating a new `option` object via an `Option` object constructor function. The syntax is as follows:

```
var newOption = new Option("text","value");
```

Here, *text* is the string that is displayed for the item in the list, and *value* is the string assigned to the `value` property of the new option. This new `option` object is not added to a `select` object until you assign it to a slot in the `options` array of the `select` object. You can see an example of this approach to modifying options in Listing 24-1.

Properties

label

Value: String. Read/Write
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The `label` property corresponds to the HTML 4.01 `label` attribute of an `option` element. This attribute (and property) enables you to assign alternate text for an option. In MacIE5, any string assigned to the `label` attribute or corresponding property overrides the display of text found between the start and end tags of the `option` element. Therefore, you can assign content to both the attribute and tag, but only browsers adhering to the HTML 4.01 standard for this element display the value assigned to the label. Although the `label` property is implemented in NN6, the browser does not modify the option item's text to reflect the property's setting.

The following statement modifies the text that appears as the selected text in a pop-up list:

```
document.forms[0].mySelect.options[3].label = "Widget 9000";
```

If this option is the currently selected one, the text on the pop-up list at rest changes to the new label.

Related Item: `text` property.

optgroup Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`form`†		
`label`		

†See "Text Input Object" (Chapter 23).

Syntax

Accessing `optgroup` object properties:

```
(IE)        [window.]document.all.elemID".property | method([parameters])
(W3C)       [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

About this object

An `optgroup` element in the HTML 4.01 specification enables authors to group options into subgroups within a `select` list. The label assigned to the `optgroup` element is rendered in the list as a non-selectable item, usually differentiated from the selectable items by some alternate display. In W3C browsers, `optgroup` items by default are shown in bold italic, while all `option` elements nested within an `optgroup` are indented but with normal font characteristics.

Browsers not recognizing this element ignore it. All options are presented as if the `optgroup` elements are not there.

Properties

label

Value: String. Read/Write
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The `label` property corresponds to the HTML 4.01 `label` attribute of an `optgroup` element. This attribute (and property) enables you to assign text to the label that encompasses a group of nested `option` elements in the pop-up list display.

MacIE5 exhibits a bug that prevents scripts from assigning values to the last `optgroup` element inside a `select` element.

I present Listing 24-9 in the hope that Microsoft and Netscape will eventually eradicate the bugs that afflict their current implementations of the `label` property. When the feature works as intended, Listing 24-9 demonstrates how a script can alter the text of option group labels. This page is an enhanced version of the background color setters used in other examples of this chapter. Be aware that several versions of IE do not alter the last `optgroup` element's label, and NN6+ achieves only a partial change to the text displayed in the `select` element.

Listing 24-9: **Modifying optgroup Element Labels**

```
<html>
   <head>
      <title>Color Changer 3</title>
      <script type="text/javascript">
      var regularLabels = ["Reds","Greens","Blues"];
      var naturalLabels = ["Apples","Leaves","Sea"];
```

Continued

Listing 24-9 *(continued)*

```
      function setRegularLabels(list) {
         var optGrps = list.getElementsByTagName("optgroup");
         for (var i = 0; i < optGrps.length; i++) {
            optGrps[i].label = regularLabels[i];
         }
      }
      function setNaturalLabels(list) {
         var optGrps = list.getElementsByTagName("optgroup");
         for (var i = 0; i < optGrps.length; i++) {
            optGrps[i].label = naturalLabels[i];
         }
      }
      function seeColor(list) {
         var newColor = (list.options[list.selectedIndex].value);
         if (newColor) {
            document.bgColor = newColor;
         }
      }
      </script>
   </head>
   <body onunload="document.forms[0].reset()">
      <form>
         <p>Choose a background color: <select name="colorsList"
            onchange="seeColor(this)">
               <optgroup id="optGrp1" label="Reds">
                  <option value="#ff9999">Light Red</option>
                  <option value="#ff3366">Medium Red</option>
                  <option value="#ff0000">Bright Red</option>
                  <option value="#660000">Dark Red</option>
               </optgroup>
               <optgroup id="optGrp2" label="Greens">
                  <option value="#ccff66">Light Green</option>
                  <option value="#99ff33">Medium Green</option>
                  <option value="#00ff00">Bright Green</option>
                  <option value="#006600">Dark Green</option>
               </optgroup>
               <optgroup id="optGrp3" label="Blues">
                  <option value="#ccffff">Light Blue</option>
                  <option value="#66ccff">Medium Blue</option>
                  <option value="#0000ff">Bright Blue</option>
                  <option value="#000066">Dark Blue</option>
               </optgroup>
            </select></p>
         <p><input type="radio" name="labels" checked="checked"
            onclick="setRegularLabels(this.form.colorsList)" />Regular Label
            Names <input type="radio" name="labels"
            onclick="setNaturalLabels(this.form.colorsList)" />Label Names from
            Nature</p>
      </form>
   </body>
</html>
```

Related Item: `option.label` property.

file Input Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	Methods	Event Handlers
defaultValue†	select()†	onchange†
form†		
name†		
readOnly†		
size†		
type†		
value†		

†See "Text Input Object" (Chapter 23).

Syntax

Accessing `file` input element object properties:

```
(NN3+/IE4+)    [window.]document.formName.inputName.property |
method([parameters])
(NN3+/IE4+)    [window.]document.formName.elements[index].property |
               method([parameters])
(NN3+/IE4+)    [window.]document.forms[index].inputName.property |
method([parameters])
(NN3+/IE4+)    [window.]document.forms["formName"].inputName.property |
               method([parameters])
(NN3+/IE4+)    [window.]document.forms["formName"].elements[index].property |
               method([parameters])
(IE4+)         [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)     [window.]document.getElementById("elemID").property |
               method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

About this object

Some Web sites enable you to upload files from the client to the server, typically by using a form-style submission to a CGI program on the server. The `input` element whose type is set to `"file"` (also known as a `fileUpload` object) is merely a user interface that enables users to specify which file on their PC they want to upload. Without a server process capable of receiving the file, the `file` input element does nothing. Moreover, you must also set two `form` element attributes as follows:

```
method="POST"
enctype="multipart/form-data"
```

This element displays a field and a Browse button. The Browse button leads to an Open File dialog box (in the local operating system's interface vernacular) where a user can select a file.

After you make a selection, the filename (or pathname, depending on the operating system) appears in the `file` input element's field. The `value` property of the object returns the filename.

You do not have to script much for this object on the client side. The `value` property, for example, is read-only in earlier browsers; in addition, a form cannot surreptitiously upload a file to the server without the user's knowledge or consent.

Listing 24-10 helps you see what the `file` input element looks like in an example page.

Listing 24-10: **file Input Element**

```
<html>
   <head>
      <title>FileUpload Object</title>
   </head>
   <body>
      <form method="POST" action="yourCGIURL" enctype="multipart/form-data">
         File to be uploaded: <input type="file" size="40" name="fileToGo" />
         <p><input type="button" value="View Value"
            onclick="alert(this.form.fileToGo.value)" /></p>
      </form>
   </body>
</html>
```

In a true production environment, a Submit button and a URL to your CGI process are specified for the `action` attribute of the `<form>` tag.

✦ ✦ ✦

CHAPTER 25

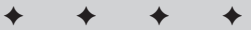
Event Objects

In This Chapter

The "life" of an event object

Event support in different browser generations

Retrieving information from an event



Prior to version 4 browsers, user and system actions — events — were captured predominantly by event handlers defined as attributes inside HTML tags. For instance, when a user clicked a button, the `click` event triggered the `onclick` event handler in the tag. That handler may invoke a separate function or perform some inline JavaScript script. Even so, the events themselves were rather dumb: Either an event occurred or it didn't. Where an event occurred (that is, the screen coordinates of the pointer at the moment the mouse button was clicked) and other pertinent event tidbits (for example, whether a keyboard modifier key was pressed at the same time) were not part of the equation. Until version 4 browsers, that is.

While remaining fully backward-compatible with the event handler mechanism of old, version 4 browsers had the first event model that turned events into first-class objects whose properties automatically carry a lot of relevant information about the event when it occurs. These properties are fully exposed to scripts, allowing pages to respond more intelligently about what the user does with the page and its elements.

Another new aspect of version 4 event models was the notion of "event propagation." It was possible to have an event processed by an object higher up the element containment hierarchy whenever it made sense to have multiple objects share one event handler. That the event being processed carried along with it information about the intended target, plus other golden information nuggets, made it possible for event handler functions to be smart about processing the event without requiring an event handler call to pass all kinds of target-specific information.

Unfortunately, the joy of this newly found power is tempered by the forces of object model incompatibility. No fewer than three event object models are in use today: the one initiated by, and unique to, NN4 (whose importance diminishes with each passing day as users migrate to other, newer browsers); the IE4+ model; and the model adopted by the W3C DOM Level 2 as implemented in NN6+/Moz1+/Safari1+. Many of these distinctions are addressed in the overviews of the object models in Chapter 15. In this chapter, you find out more about the actual event objects that contain all the "goodies." Where possible, cross-browser concerns are addressed.

Why "Events"?

Graphical user interfaces are more difficult to program than the "old-fashioned" command-line interface. With a command-line or menu-driven system, users were intentionally restricted in the

types of actions they could take at any given moment. The world was very modal, primarily as a convenience to programmers who led users through rigid program structures.

That all changed in a graphical user interface, such as Windows, MacOS, XWindow System, and all others derived from the pioneering work of the Xerox Star system. The challenge for programmers is that a good user interface in this realm must make it possible for users to perform all kinds of actions at any given moment: roll the mouse, click a button, type a key, select text, choose a pull-down menu item, and so on. To accommodate this, a program (or, better yet, the operating system) must be on the lookout for any possible activity coming from all input ports, whether it be the mouse, keyboard, or network connection.

A common methodology to accomplish this at the operating system level is to look for any kind of event, whether it comes from user action or some machine-generated activity. The operating system or program then looks up how it should process each kind of event. Such events, however, must have some smarts about them so that the program knows what and where on the screen the event is.

What an event knows (and when it knows it)

Although the way to reference an event object varies a bit among the three event models, the one concept they all share is that an event object is created the instant the event action occurs. For instance, if you click a button, an event object is created in the browser's memory. As the object is created, the browser assigns values to the object's properties — properties that reflect numerous characteristics of that specific event. For a `click` event, that information includes the coordinates of the click and which mouse button was used to generate the event. To be even more helpful, the browser does some quick calculations to determine that the coordinates of the `click` event coincide with the rectangular space of a button element on the screen. Therefore, the event object has as one of its properties a reference to the "screen thing" that you clicked on.

Most event object properties (all of them in some event models) are read-only, because an event object is like a snapshot of an event action. If the event model were to allow modification of event properties, performing both potentially useful and potentially unfriendly actions would be possible. For example, how frustrating would it be to a user to attempt to type into a text box only to have a keystroke modified after the actual key press and then have a totally different character appear in the text box? On the other hand, perhaps it may be useful in some situations to make sure that anything typed into a text box is converted to uppercase characters, no matter what is typed. Each event model brings its own philosophy to the table in this regard. For example, the IE4+ event model allows keyboard character events to be modified by script; the NN4 and W3C DOM event models do not.

Perhaps the most important aspect of an event object to keep in mind is that it exists only as long as scripts process the event. An event can trigger an event handler — usually a function. That function, of course, can invoke other functions. As long as statements are still executing in response to the event handler, the event object and all its properties are still "alive" and available to your scripts. But after the last script statement runs, the event object reverts to an empty object.

The reason an event object has such a brief life is that there can be only one event object at a time. In other words, no matter how complex your event handler functions are or how rapidly events fire, they are executed serially (for experienced programmers: there is one execution thread). The operating system buffers events that start to bunch up on each other. Except in rare cases in which the buffer gets full and events are not recorded, event handlers are executed in the order in which the events occur.

The static Event object

Up to this point, the discussion has been about the event object (with a lowercase "e"), which is one instance of an event, with all the properties associated with that specific event action. In the NN4 and W3C DOM event models, there is also a static `Event` object (with an uppercase "E"). In the W3C DOM event model are additional subcategories of the `Event` object. These subcategories are all covered later in this chapter, but they are introduced here to draw the contrast between the event and `Event` objects. The former, as you've seen, is a transient object with details about a specific event action; the latter serves primarily as a holder of event-related constant values that scripts can use. The static `Event` object is always available to scripts inside any window or frame. If you want to see a list of all `Event` object properties in NN4 and NN6+, use The Evaluator (Chapter 13): enter `Event` into the bottom text box (also check out the `KeyEvent` object in NN6+).

The static `Event` object also turns out to be the object from which event objects are cloned. Thus, the static `Event` object has a number of properties and methods that apply to (are inherited by) the event objects created by event actions. These relationships are more important in the W3C DOM event model, which builds upon the DOM's object-oriented tendencies to implement the event model.

Event Propagation

Prior to version 4 browsers, an event fired on an object. If an event handler was defined for that event and that object, the handler executed; if there was no event handler, the event just disappeared into the ether. Newer browsers, however, send events on a longer ride, causing them to propagate through the document object models. As you know by now, three propagation models exist, one for each of the event models in use today: NN4, IE4+, and W3C DOM as implemented in NN6+/Moz1+/Safari1+. Conceptually, the NN4 and IE4+ propagation models are diametrically opposite each other — any NN4 event propagates inward toward the target, whereas an IE event starts at the target and propagates outward. But the W3C DOM model manages to implement both models simultaneously, albeit with all new syntax so as not to step on the older models.

At the root of all three models is the notion that every event has a target. For user-initiated actions, this is fairly obvious. If you click a button or type in a text box, that button is the target of your mouse-related event; the text box is the target of your keyboard event. System-generated events are not so obvious, such as the `onload` event after a page finishes loading. In all event models, this event fires on the `window` object. What distinguishes the event propagation models is how an event reaches its target, and what, if anything, happens to the event after it finishes executing the event handler associated with the target.

NN4-only event propagation

Although the installed base of NN4 has largely given way to newer browsers, its propagation model initiated some concepts that are found in the modern W3C DOM event propagation model. The name for the model is *event capture*.

In NN4, all events propagate from the top of the document object hierarchy (starting with the `window` object) downward to the target object. For example, if you click a button in a form, the `click` event passes through the `window` and `document` (and, if available, `layer`) objects before reaching the button (the `form` object is not part of the propagation path). This propagation happens instantaneously, so that there is no performance penalty by this extra journey.

The event that passes through the window, document, and layer objects is a fully formed event object, complete with all properties relevant to that event action. Therefore, if the event were processed at the window level, one of the event object's properties is a reference to the target object, so that the event handler scripts at the window level can find out information, such as the name of the button, and even get a reference to its enclosing form.

By default, event capture is turned off. To instruct the window, document, or layer object levels to process that passing click object requires turning on event capture for the window, document, and/or layer object.

Enabling NN4 event capture

All three objects just mentioned — window, document, and layer — have a captureEvents() method. You use this method to enable event capture at any of those object levels. The method requires one or more parameters, which are the event types (as supplied by Event object constants) that the object should capture, while letting all others pass untouched. For example, if you want the window object to capture all keypress events, you include the following statement in a script that executes as the page loads:

```
window.captureEvents(Event.KEYPRESS);
```

Defining event handlers in the intended targets is also a good idea, even if they are empty (for example, onkeypress="") to help NN4 generate the event in the first place. If you want the window to capture multiple event types, string the event type constants together, separated by the pipe character:

```
window.captureEvents(Event.KEYPRESS | Event.CLICK);
```

Now you must assign an action to the event at the window's level for each event type. More than likely, you have defined functions to execute for the event. Assign a function reference to the event handler by setting the handler property of the window object:

```
window.onkeypress = processKeyEvent;
window.onclick = processClickEvent;
```

Hereafter, if a user clicks a button or types into a field inside that window, the events are processed by their respective window-level event handler functions.

Turning off event capture

As soon as you enable event capture for a particular event type in a document, that capture remains in effect until the page unloads or you specifically disable the capture. You can turn off event capture for each event via the window, document, or layer releaseEvents() method. The releaseEvents() method takes the same kind of parameters — Event object type constants — as the captureEvents() method.

The act of releasing an event type simply means that events go directly to their intended targets without stopping elsewhere for processing, even if an event handler for the higher-level object is still defined. And because you can release individual event types based on parameters set for the releaseEvents() method, other events being captured are not affected by the release of others.

To demonstrate not only the captureEvents() and releaseEvents() methods, but other event model techniques, I present a series of several versions of the same document. Each successive version implements an added feature to help you experience the numerous interactions among events and event handling methods. The document merely contains a few buttons, plus some switches to enable and disable various methods being demonstrated in the section. A layer object is also thrown into the mixture because a lot of impetus for capturing and modifying event handling comes from application of layers in a document.

Listing 25-1 is the first example, which shows the basic event capture and release from the outermost document level. A checkbox lets you enable or disable the document-level capture of `click` events (all checkboxes in these examples use `onmouseup` event handlers to avoid getting in the way of tracing `click` events). Because all `click` events are being captured by the outermost document, even clicks to the layer's buttons get trapped by the outermost document when `captureEvents()` is set.

Listing 25-1: **Event Capture and Release**

```
<html>
  <head>
     <title></title>
     <script type="text/javascript">
     function setDocCapture(enable) {
        if (!enable) {
           document.captureEvents(Event.CLICK);
        } else {
           document.releaseEvents(Event.CLICK);
        }
     }
     function doMainClick(e) {
        if (e.target.type == "button") {
           alert("Captured in top document");
        }
     }
     document.captureEvents(Event.CLICK);
     document.onclick=doMainClick;
     </script>
  </head>
  <body>
     <b>Basic document-level capture of Event.CLICK</b>
     <hr />
     <form>
        <input type="checkbox" onmousedown="setDocCapture(this.checked)"
        checked="checked" />Enable Document Capture
        <hr />
        <input type="button" value="Button 'main1'" name="main1"
        onclick="alert('Event finally reached Button:' + this.name)" />
     </form>
     <layer id="layer1" left="200" top="150" bgcolor="coral">
        <form>
           <br />
           <p><input type="button" value="Button 'layerButton1'"
              name="layerButton1"
              onclick="alert('Event finally reached Button:' + this.name)" />
              </p>
           <p><input type="button" value="Button 'layerButton2'"
              name="layerButton2"
              onclick="alert('Event finally reached Button:' + this.name)" />
              </p>
        </form>
     </layer>
  </body>
</html>
```

With document-level event capture turned on (the default), all `click` events are trapped by the document's `onclick` event handler property, a function that alerts the user that the event was captured by the top document. Because all `click` events for buttons are trapped there, even `click` events of the layer's buttons are trapped at the top. But if you turn off event capture, the events reach their intended targets.

If the logic of the `setDocCapture()` function seems backward to you, recall that when the `onmousedown` event fires on the checkbox, its state is the opposite of what it is being changed to.

In Listing 25-2, I add some code (shown in boldface) that lets the `layer` object capture `click` events whenever the outer document event capture is turned off. Inside the `<layer>` tag, a script sets the layer to capture `click` events. Therefore, if you disable the outer document capture, the `click` event goes straight to the `main1` button and to the layer event capture. Event capture in the `layer` object prevents the events from ever reaching the buttons in the layer, unless you disable event capture for both the document and the layer.

Listing 25-2: **Document and Layer Event Capture and Release**

```
<html>
   <head>
      <title></title>
      <script type="text/javascript">
      function setDocCapture(enable) {
         if (!enable) {
            document.captureEvents(Event.CLICK);
         } else {
            document.releaseEvents(Event.CLICK);
         }
      }
      function setLayerCapture(enable) {
         if (!enable) {
            document.layer1.captureEvents(Event.CLICK);
         } else {
            document.layer1.releaseEvents(Event.CLICK);
         }
      }
      function doMainClick(e) {
         if (e.target.type == "button") {
            alert("Captured in top document");
         }
      }
      document.captureEvents(Event.CLICK);
      document.onclick=doMainClick;
      </script>
   </head>
   <body>
      <b>Document-level and/or Layer-level capture of Event.CLICK</b>
      <hr />
      <form>
         <input type="checkbox" onmousedown="setDocCapture(this.checked)"
         checked="checked" />Enable Document Capture
         <input type="checkbox" onmousedown="setLayerCapture(this.checked)"
         checked="checked" />Enable Layer Capture
         <hr />
```

```
            <input type="button" value="Button 'main1'" name="main1"
            onclick="alert('Event finally reached Button:' + this.name)" />
         </form>
         <layer id="layer1" left="200" top="150" bgcolor="coral">
            <script type="text/javascript">
            function doLayerClick(e) {
               if (e.target.type == "button") {
                  alert("Captured in layer1");
               }
            }
            layer1.captureEvents(Event.CLICK);
            layer1.onclick=doLayerClick;
            </script>
            <form>
                layer1<br />
               <p><input type="button" value="Button 'layerButton1'"
                  name="layerButton1"
                  onclick="alert('Event finally reached Button:' + this.name)" />
                  </p>
               <p><input type="button" value="Button 'layerButton2'"
                  name="layerButton2"
                  onclick="alert('Event finally reached Button:' + this.name)" />
                  </p>
            </form>
         </layer>
      </body>
   </html>
```

Passing events toward their targets

If you capture a particular event type, your script may need to perform some limited processing on that event before letting it reach its intended target. For example, perhaps you want to do something special if a user clicks an element with the Shift meta key pressed. In that case, the function that handles the event at the document level inspects the event's `modifiers` property to determine if the Shift key was pressed at the time of the event. If the Shift key was not pressed, you want the event to continue on its way to the element that the user clicked.

To let an event pass through the object hierarchy to its target, you use the `routeEvent()` method, passing as a parameter the event object being handled in the current function. A `routeEvent()` method does not guarantee that the event will reach its intended destination, because another object in between may have event capturing for that event type turned on and will intercept the event. That object, too, can let the event pass through with its own `routeEvent()` method.

Listing 25-3 demonstrates event routing by adding onto the document being built in previous examples. While the clickable button objects are the same, additional powers are added to the document and layer function handlers that process events that come their way. For each of these event-capturing objects, you have additional checkbox settings to allow or disallow events from passing through after each level has processed them.

The default settings for the checkboxes are like the ones in Listing 25-2, where event capture (for the `click` event) is set for both the `document` and `layer` objects. Clicking any button causes the `document` object's event handler to process and none other. But if you then enable the checkbox that lets the event continue, you find that click events on the layer buttons cause alerts to display from both the `document` and `layer` object event handler functions. If you then also let events continue from the `layer` object, a click on the button displays a third

alert, showing that the event has reached the buttons. Because the `main1` button is not in the layer, none of the `layer` object event handling settings affect its behavior.

Listing 25-3: **Capture, Release, and Route Events**

```
<html>
   <head>
      <title></title>
      <script type="text/javascript">
      function setDocCapture(enable) {
         if (!enable) {
            document.captureEvents(Event.CLICK);
         } else {
            document.releaseEvents(Event.CLICK);
            document.forms[0].setDocRte.checked = false;
            docRoute = false;
         }
      }
      function setLayerCapture(enable) {
         if (!enable) {
            document.layer1.captureEvents(Event.CLICK);
         } else {
            document.layer1.releaseEvents(Event.CLICK);
            document.forms[0].setLyrRte.checked = false;
            layerRoute = false;
         }
      }
      var docRoute = false;
      var layerRoute = false;
      function setDocRoute(enable) {
         docRoute = !enable;
      }
      function setLayerRoute(enable) {
         layerRoute = !enable;
      }
      function doMainClick(e) {
         if (e.target.type == "button") {
            alert("Captured in top document");
            if (docRoute) {
               routeEvent(e);
            }
         }
      }
      document.captureEvents(Event.CLICK);
      document.onclick=doMainClick;
      </script>
   </head>
   <body>
      <b>Capture, Release, and Routing of Event.CLICK</b>
      <hr />
      <form>
         <input type="checkbox" name="setDocCap"
         onmousedown="setDocCapture(this.checked)" checked="checked" />Enable
         Document Capture 
         <input type="checkbox" name="setDocRte"
         onmousedown="setDocRoute(this.checked)" />And let event continue<br />
         <input type="checkbox" name="setLyrCap"
```

```
            onmousedown="setLayerCapture(this.checked)"
            checked="checked" />Enable Layer Capture 
            <input type="checkbox" name="setLyrRte"
            onmousedown="setLayerRoute(this.checked)" />And let event
            continue<br />
            <hr />
            <input type="button" value="Button 'main1'" name="main1"
            onclick="alert('Event finally reached Button:' + this.name)" />
         </form>
         <layer id="layer1" left="200" top="150" bgcolor="coral">
            <script type="text/javascript">
            function doLayerClick(e) {
               if (e.target.type == "button") {
                  alert("Captured in layer1");
                  if (layerRoute) {
                     routeEvent(e);
                  }
               }
            }
            layer1.captureEvents(Event.CLICK);
            layer1.onclick=doLayerClick;
            </script>
            <form>
                layer1<br />
               <p><input type="button" value="Button 'layerButton1'"
                  name="layerButton1"
                  onclick="alert('Event finally reached Button:' + this.name)" />
                  </p>
               <p><input type="button" value="Button 'layerButton2'"
                  name="layerButton2"
                  onclick="alert('Event finally reached Button:' + this.name)" />
                  </p>
            </form>
         </layer>
      </body>
   </html>
```

In some cases, your scripts need to know if an event that is passed onward by the `routeEvent()` method activated a function that returns a value. This knowledge is especially valuable if your event must return a `true` or `false` value to let an object know if it should proceed with its default behavior (for example, whether a link should activate its `href` attribute URL or cancel after the event handler evaluates to `return true` or `return false`). When a function is invoked by the action of a `routeEvent()` method, the return value of the destination function is passed back to the `routeEvent()` method. That value, in turn, can be returned to the object that originally captured the event.

Event traffic cop

The last scenario is one in which a higher-level object captures an event and directs the event to a particular object elsewhere in the hierarchy. For example, you could have a document-level event handler function direct every `click` event whose `modifiers` property indicates that the Alt key was pressed to a Help button object whose own `onclick` event handler displays a help panel (perhaps shows an otherwise hidden layer).

You can redirect an event to any object via the `handleEvent()` method. This method works differently from the others described in this chapter, because the object reference of this

method is the reference of the object to handle the event (with the event object being passed as a parameter, such as the other methods). As long as the target object has an event handler defined for that event, it will process the event as if it had received the event directly from the system (even though the event object's target property may be some other object entirely).

To demonstrate how this event redirection works, Listing 25-4 includes the final additions to the document being built so far in this chapter. The listing includes mechanisms that allow all `click` events to be sent directly to the second button in the layer (`layerButton2`). The previous interaction with document and layer event capture and routing is still intact, although you cannot have event routing and redirection on at the same time.

The best way to see event redirection at work is to enable both document and layer event capture (the default settings). When you click the `main1` button, the event reaches only as far as the document-level capture handler. But if you then turn on the "Send event to 'layerButton2'" checkbox associated with the document level, a click of the `main1` button reaches both the document-level event handler and `layerButton2`, even though the `main1` button is not anywhere near the layer button in the document object hierarchy. Click other checkboxes to work with the interaction of event capturing, routing, and redirection.

Listing 25-4: **Redirecting Events**

```
<html>
   <head>
      <title></title>
      <script type="text/javascript">
      function setDocCapture(enable) {
         if (!enable) {
            document.captureEvents(Event.CLICK);
         } else {
            document.releaseEvents(Event.CLICK);
            document.forms[0].setDocRte.checked = false;
            docRoute = false;
         }
      }
      function setLayerCapture(enable) {
         if (!enable) {
            document.layer1.captureEvents(Event.CLICK);
         } else {
            document.layer1.releaseEvents(Event.CLICK);
            document.forms[0].setLyrRte.checked = false;
            layerRoute = false;
         }
      }
      var docRoute = false;
      var layerRoute = false;
      function setDocRoute(enable) {
         docRoute = !enable;
         document.forms[0].setDocShortCircuit.checked = false;
         docShortCircuit = false;
      }
      function setLayerRoute(enable) {
         layerRoute = !enable;
```

```
      document.forms[0].setLyrShortCircuit.checked = false;
      layerShortCircuit = false;
   }
   var docShortCircuit = false;
   var layerShortCircuit = false;
   function setDocShortcut(enable) {
      docShortCircuit = !enable;
      if (docShortCircuit) {
         document.forms[0].setDocRte.checked = false;
         docRoute = false;
      }
   }
   function setLayerShortcut(enable) {
      layerShortCircuit = !enable;
      if (layerShortCircuit) {
         document.forms[0].setLyrRte.checked = false;
         layerRoute = false;
      }
   }
   function doMainClick(e) {
      if (e.target.type == "button") {
         alert("Captured in top document");
         if (docRoute) {
            routeEvent(e);
         } else if (docShortCircuit) {
            document.layer1.document.forms[0].layerButton2.handleEvent(e);
         }
      }
   }
   document.captureEvents(Event.CLICK);
   document.onclick=doMainClick;
   </script>
</head>
<body>
   <b>Redirecting Event.CLICK</b>
   <hr />
   <form>
      <input type="checkbox" name="setDocCap"
      onmousedown="setDocCapture(this.checked)" checked="checked" />Enable
      Document Capture  <input type="checkbox" name="setDocRte"
      onmousedown="setDocRoute(this.checked)" />And let event continue
      <input type="checkbox" name="setDocShortCircuit"
      onmousedown="setDocShortcut(this.checked)" />Send event to
      'layerButton2'<br />
      <input type="checkbox" name="setLyrCap"
      onmousedown="setLayerCapture(this.checked)"
      checked="checked" />Enable Layer Capture  <input
      type="checkbox" name="setLyrRte"
      onmousedown="setLayerRoute(this.checked)" />And let event continue
      <input type="checkbox" name="setLyrShortCircuit"
      onmousedown="setLayerShortcut(this.checked)" />Send event to
      'layerButton2'<br />
      <hr />
      <input type="button" value="Button 'main1'" name="main1"
      onclick="alert('Event finally reached Button:' + this.name)" />
   </form>
```

Continued

Listing 25-4 *(continued)*

```
      <layer id="layer1" left="200" top="200" bgcolor="coral">
         <script type="text/javascript">
         function doLayerClick(e) {
            if (e.target.type == "button") {
               alert("Captured in layer1");
               if (layerRoute) {
                  routeEvent(e);
               } else if (layerShortCircuit) {
                  document.forms[0].layerButton2.handleEvent(e);
               }
            }
         }
         layer1.captureEvents(Event.CLICK)
         layer1.onclick=doLayerClick
         </script>
         <form>
             layer1<br />
            <p><input type="button" value="Button 'layerButton1'"
               name="layerButton1"
               onclick="alert('Event finally reached Button:' + this.name)" />
               </p>
            <p><input type="button" value="Button 'layerButton2'"
               name="layerButton2"
               onclick="alert('Event finally reached Button:' + this.name)" />
               </p>
         </form>
      </layer>
   </body>
</html>
```

IE4+ event propagation

IE's event propagation model is called *event bubbling*, in which events "bubble" upward from the target object through the HTML element containment hierarchy. It's important to distinguish between the old-fashioned document object hierarchy (followed in the NN4 event capture model) and the more modern notion of HTML element containment — a concept that carries to the W3C DOM as well.

A good way to demonstrate the effect of event bubbling — a behavior that is turned on by default — is to populate a simple document with lots of event handlers to see which ones fire and in what order. Listing 25-5 has `onclick` event handlers defined for a button inside a form, the form itself, and other elements and objects all the way up the hierarchy out to the window.

Listing 25-5: **Event Bubbling Demonstration**

```
<html onclick="alert('Event is now at the HTML element.')">
   <head>
      <title>Event Bubbles</title>
      <script type="text/javascript">
```

```
      function init() {
      window.onclick = winEvent
         document.onclick = docEvent;
         document.body.onclick = docBodEvent;
      }
      function winEvent() {
         alert("Event is now at the window object level.");
      }
      function docEvent() {
         alert("Event is now at the document object level.");
      }
      function docBodEvent() {
         alert("Event is now at the BODY element.");
      }
      </script>
   </head>
   <body onload="init()">
      <h1>Event Bubbles</h1>
      <hr />
      <form onclick="alert('Event is now at the FORM element.')">
         <input type="button" value="Button 'main1'" name="main1"
         onclick="alert('Event started at Button: ' + this.name)" />
      </form>
   </body>
</html>
```

You can try this listing in IE4+ and even NN6+ or Safari, because W3C DOM browsers also observe event bubbling. But you will notice differences in the precise propagation among WinIE4+, MacIE4+, and W3C DOM browsers. But first, notice that after you click the button in Listing 25-5, the event first fires at the target: the button. Then the event bubbles upward through the HTML containment to fire at the enclosing `form` element; next to the enclosing `body` element; and so on. Where the differences occur are after the `body` element. Table 25-1 shows the objects for which event handlers are defined in Listing 25-5 and which objects have the `click` event bubble to them in the three classes of browsers.

Table 25-1: Event Bubbling Variations for Listing 25-5

Event Handler Location	*WinIE4+*	*MacIE4+*	*NN6+/Moz1+/Safari*
`button`	yes	yes	yes
`form`	yes	yes	yes
`body`	yes	yes	yes
`HTML`	yes	no	yes
`document`	yes	yes	yes
`window`	no	no	yes

Despite the discrepancies in Table 25-1, events do bubble through the most likely HTML containers that come to mind. The object level with the most global scope and that works in all browser categories shown in the table is the `document` object.

Preventing IE event bubbling

Because bubbling occurs by default, there are times when you may prefer to prevent an event from bubbling up the hierarchy. For example, if you have one handler at the `document` level whose job is to deal with the `click` event from a related series of buttons, any other object that receives `click` events will allow those events to bubble upward to the `document` level unless the bubbling is cancelled. Having the event bubble up could conflict with the document-level event handler.

Each `event` object in IE has a property called `cancelBubble`. The default value of this property is `false`, which means that the event bubbles to the next outermost container that has an event handler for that event. But if, in the execution of an event handler, that property is set to `true`, the processing of that handler finishes its job, but the event does not bubble up any higher. Therefore, to stop an event from bubbling beyond the current event handler, include the following statement somewhere in the handler function:

```
event.cancelBubble = true;
```

You can prove this to yourself by modifying the page in Listing 25-5 to cancel bubbling at any level. For example, if you change the event handler of the `form` element to include a statement that cancels bubbling, the event goes no further than the `form` in IE (the syntax is different for NN6+/Moz1+, as discussed later):

```
<form onclick="alert('Event is now at the form element.');
   event.cancelBubble=true">
```

Preventing IE event default action

In the days when events were almost always bound to elements by way of attributes in tags, the technique to block the event's default action was to make sure the event handler evaluated to `return false`. This is how, for instance, a `form` element's `onsubmit` event handler could prevent the form from carrying out the submission if client-side form validation failed.

To enhance that capability — especially when events are bound by other means, such as object element properties — IE's `event` object includes a `returnValue` property. Assign `false` to this property in the event handler function to block the element's default action to the event:

```
event.returnValue = false;
```

This way of blocking default actions in IE is often more effective than the old `return false` technique.

Redirecting events

Starting with IE5.5, you can redirect an event to another element, but with some limitations. The mechanism that makes this possible is the `fireEvent()` method of all HTML element objects (see Chapter 15). This method isn't so much redirecting an event as causing a brand-new event to be fired. But you can pass most of the properties of the original `event` object with the new event by specifying a reference to the old `event` object as the optional second parameter to the `fireEvent()` method.

The big limitation in this technique, however, is that the reference to the target element gets lost in this hand-off to the new event. The `srcElement` property of the old event gets overwritten with a reference to the object that is the target of the call to `fireEvent()`. For example, consider the following `onclick` event handler function for a button inside a `form` element:

```
function buttonEvent() {
   event.cancelBubble = true;
   document.body.fireEvent("onclick", event);
}
```

By cancelling event bubbling, the event does not propagate upward to the enclosing `form` element. Instead, the event is explicitly redirected to the `body` element, passing the current `event` object as the second parameter. When the event handler function for the `body` element runs, its `event` object has information about the original event, such as the mouse button used for the click and the coordinates. But the `event.srcElement` property points to the `document.body` object. As the event bubbles upward from the `body` element, the `srcElement` property continues to point to the `document.body` object. You can see this at work in Listing 25-6 for IE5.5+.

Listing 25-6: **Cancelling and Redirecting Events in IE5.5+**

```
<html onclick="revealEvent('HTML', event)">
  <head>
     <title>Event Cancelling & Redirecting</title>
     <script type="text/javascript">
     // display alert with event object info
     function revealEvent(elem, evt) {
        var msg = "Event (from " + evt.srcElement.tagName + " at ";
        msg += event.clientX + "," + event.clientY + ") is now at the ";
        msg += elem + " element.";
        alert(msg);
     }
     function init() {
        document.onclick = docEvent;
        document.body.onclick = docBodEvent;
     }
     function docEvent() {
        revealEvent("document", event);
     }
     function docBodEvent() {
        revealEvent("BODY", event);
     }
     function buttonEvent(form) {
        revealEvent("BUTTON", event);
        // cancel if checked (IE4+)
        event.cancelBubble = form.bubbleCancelState.checked;
        // redirect if checked (IE5.5+)
        if (form.redirect.checked) {
           document.body.fireEvent("onclick", event);
        }
     }
     </script>
  </head>
  <body onload="init()">
     <h1>Event Cancelling &amp;amp; Redirecting</h1>
     <hr />
     <form onclick="revealEvent('FORM', event)">
        <p><button name="main1" onclick="buttonEvent(this.form)">Button
           'main1'</button></p>
```

Continued

Listing 25-6 *(continued)*

```
        <p><input type="checkbox" name="bubbleCancelState"
           onclick="event.cancelBubble=true" />Cancel Bubbling at BUTTON<br />
           <input type="checkbox" name="redirect"
           onclick="event.cancelBubble=true" /> Redirect Event to BODY</p>
     </form>
  </body>
</html>
```

Listing 25-6 is a modified version of Listing 25-5. Major additions are enhanced event handlers at each level so that you can see the tag name of the event that is regarded as the `srcElement` of the event as well as the coordinates of the `click` event. With both checkboxes unchecked, events bubble upward from the button, and the `button` element is then shown to be the original target all the way up the bubble hierarchy. If you check the Cancel Bubbling checkbox, the event goes no further than the `button` element, because that's where event bubbling is turned off. If you then check the Redirect Event to `body` checkbox, the original event is cancelled at the `button` level, but a new event is fired at the `body` element. But notice that by passing the old `event` object as the second parameter, the click location properties of the old event are applied to the new event directed at the `body`. This event then continues to bubble upward from the `body`.

As a side note, if you uncheck the Cancel Bubbling checkbox but leave the Redirect Event box checked, you can see how the redirection is observed at the end of the `button`'s event handler, and something special goes on. The original event is held aside by the browser while the redirected event bubbles upward. As soon as that event-processing branch finishes, the original bubbling propagation carries on with the `form`. Notice, though, that the `event` object still knows that it was targeted at the `button` element, and the other properties are intact. This means that for a time, two `event` objects were in the browser's memory, but only one is "active" at a time. While the redirected event is propagating, the `window.event` object refers to that `event` object only.

Applying event capture

WinIE 5 and later also provide a kind of event capture, which overrides all other event propagation. Intended primarily for temporary capture of mouse events, it is controlled not through the event object but via the `setCapture()` and `releaseCapture()` methods of all HTML element objects (described in Chapter 15).

When you engage capture mode, all mouse events are directed to the element object that invoked the `setCapture()` method, regardless of the actual target of the event. This action facilitates such activities as element dragging so that mouse events that might fire outside of the intended target (for example, when dragging the cursor too fast for the animation to track) continue to go to the target. When the drag mode is no longer needed, invoke the `releaseCapture()` method to allow mouse events to propagate normally.

W3C event propagation

Yielding to arguments in favor of both NN4's event capture and IE's event bubbling, the W3C DOM group managed to assemble an event model that employs both propagation systems. Although forced to use new syntax so as not to conflict with older browsers, the W3C DOM propagation model works like the NN4 one for capture and like IE4+ for bubbling. In other

words, an event bubbles by default, but you can also turn on event capture if you want. Thus, an event first trickles down the element containment hierarchy to the target; then it bubbles up through the reverse path.

Event bubbling is on by default, just as in IE4+. To enable capture, you must apply a W3C DOM event listener to an object at some higher container. Use the `addEventListener()` method (see Chapter 15) for any visible HTML element or node. One of the parameters of the `addEventListener()` method determines whether the event listener function should be triggered while the event is bubbling or is captured.

Listing 25-7 is a simplified example that demonstrates how a `click` event aimed at a button can be both captured and allowed to bubble. Most event handling functions are assigned inside the `init()` function. Borrowing code from Listing 25-5, event handlers are assigned to the `window`, `document`, and `body` objects as property assignments. These are automatically treated as bubble-type event listeners. Next, two objects — the `document` and a form — are given capture-type event listeners for the `click` event. The `document` object event listener invokes the same function as the bubble-type event handler (the alert text includes some asterisks to remind you that it is the same alert being displayed in both the capture and bubble phases of the event). For the `form` object, however, the capture-type event listener is directed to one function, while a bubble-type listener for the same object is directed at a separate function. In other words, the `form` object invokes one function as the event trickles down to the target and another function when the event starts bubbling back up. Many of the event handler functions dynamically read the `eventPhase` property of the event object to reveal which phase of event propagation is in force at the instance the event handler is invoked (although an apparent bug reports the incorrect phase at the `document` object during event capture).

Listing 25-7: **NN6+ Event Capture and Bubble**

```
<html>
   <head>
      <title>W3C DOM Event Propagation</title>
      <script type="text/javascript">
      function init() {
         // using old syntax to assign bubble-type event handlers
         window.onclick = winEvent;
         document.onclick = docEvent;
         document.body.onclick = docBodEvent;
         // turn on click event capture for document and form objects
         document.addEventListener("click", docEvent, true);
         document.forms[0].addEventListener("click", formCaptureEvent, true);
         // set event listener for bubble
         document.forms[0].addEventListener("click", formBubbleEvent, false);
      }
      function winEvent(evt) {
         alert("Event is now at the window object level (" +
            getPhase(evt) + ").");
      }
      function docEvent(evt) {
         alert("Event is now at the **document** object level (" +
            getPhase(evt) + ").");
      }
```

Continued

Listing 25-7 *(continued)*

```
      function docBodEvent(evt) {
         alert("Event is now at the BODY level (" + getPhase(evt) + ").");
      }
      function formCaptureEvent(evt) {
         alert("This alert triggered by FORM only on CAPTURE.");
      }
      function formBubbleEvent(evt) {
         alert("This alert triggered by FORM only on BUBBLE.");
      }
      // reveal event phase of current event object
      function getPhase(evt) {
         switch (evt.eventPhase) {
         case 1:
            return "CAPTURING";
            break;
         case 2:
            return "AT TARGET";
            break;
         case 3:
            return "BUBBLING";
            break;
         default:
            return "";
         }
      }
      </script>
   </head>
   <body onload="init()">
      <h1>W3C DOM Event Propagation</h1>
      <hr />
      <form>
         <input type="button" value="Button 'main1'" name="main1"
         onclick="alert('Event is now at the button object level (' +
         getPhase(event) + ').')" />
      </form>
   </body>
</html>
```

If you want to remove event capture after it has been enabled, use the `removeEventListener()` method on the same object as the event listener that was originally added (see Chapter 15). And, because multiple event listeners can be attached to the same object, specify the exact same three parameters to the `removeEventListener()` method as applied to the `addEventListener()` method.

Preventing W3C event bubbling or capture

Corresponding to the `cancelBubble` property of the IE4+ `event` object is an event object method in the W3C DOM. The method that prevents propagation in any event phase is the `stopPropagation()` method. Invoke this method anywhere within an event listener handler function. The current function executes to completion, but the event propagates no further.

Listing 25-8 extends the example of Listing 25-7 to include two checkboxes that let you stop propagation type at the `form` element in your choice of the capture or bubble phase.

Listing 25-8: **Preventing Bubble and Capture**

```
<html>
   <head>
      <title>W3C DOM Event Propagation</title>
      <script type="text/javascript">
      function init() {
         // using old syntax to assign bubble-type event handlers
         window.onclick = winEvent;
         document.onclick = docEvent;
         document.body.onclick = docBodEvent;
         // turn on click event capture for two objects
         document.addEventListener("click", docEvent, true);
         document.forms[0].addEventListener("click", formCaptureEvent, true);
         // set event listener for bubble
         document.forms[0].addEventListener("click", formBubbleEvent, false);
      }
      function winEvent(evt) {
         if (evt.target.type == "button") {
            alert("Event is now at the window object level (" +
               getPhase(evt) + ").");
         }
      }
      function docEvent(evt) {
         if (evt.target.type == "button") {
            alert("Event is now at the **document** object level (" +
               getPhase(evt) + ").");
         }
      }
      function docBodEvent(evt) {
         if (evt.target.type == "button") {
            alert("Event is now at the BODY level (" + getPhase(evt) + ").");
         }
      }
      function formCaptureEvent(evt) {
         if (evt.target.type == "button") {
            alert("This alert triggered by FORM only on CAPTURE.");
            if (document.forms[0].stopAllProp.checked) {
               evt.stopPropagation();
            }
         }
      }
      function formBubbleEvent(evt) {
         if (evt.target.type == "button") {
            alert("This alert triggered by FORM only on BUBBLE.");
            if (document.forms[0].stopDuringBubble.checked) {
               evt.preventBubble();
            }
         }
      }
```

Continued

Listing 25-8 *(continued)*

```
      // reveal event phase of current event object
      function getPhase(evt) {
         switch (evt.eventPhase) {
         case 1:
            return "CAPTURING";
            break;
         case 2:
            return "AT TARGET";
            break;
         case 3:
            return "BUBBLING";
            break;
         default:
            return "";
         }
      }
      </script>
   </head>
   <body onload="init()">
      <h1>W3C DOM Event Propagation</h1>
      <hr />
      <form>
         <input type="checkbox" name="stopAllProp" />Stop all propagation at
         FORM<br />
         <input type="checkbox" name="stopDuringBubble" />Prevent bubbling past
         FORM
         <hr />
         <input type="button" value="Button 'main1'" name="main1"
         onclick="alert('Event is now at the button object level (' +
         getPhase(event) + ').')" />
      </form>
   </body>
</html>
```

In addition to the W3C DOM `stopPropagation()` method, NN6+, Moz1+, and Safari also support IE's `cancelBubble` property for syntactical convenience.

Preventing W3C event default action

The W3C DOM counterpart to IE's `returnValue` property is the event object's `preventDefault()` method. Invoke this method in an event handler function when you wish to block the element's default action to the event:

```
evt.preventDefault();
```

Redirecting W3C DOM events

The mechanism for sending an event to an object outside the normal propagation pattern in W3C is similar to that of IE4+, although with different syntax. In place of the IE4+ `fireEvent()` method, NN6+/Moz1+/Safari uses the W3C DOM `dispatchEvent()` method. The sole parameter of the method is an event object, such as the current event object. Listing 25-9 is the same

as the IE4+ Listing 25-6, but with just a few modifications to run in the W3C event model. Notice that the `dispatchEvent()` method passes the current event object as its sole parameter.

Listing 25-9: **Cancelling and Redirecting Events in the W3C DOM**

```
<html onclick="revealEvent('HTML', event)">
   <head>
      <title>Event Cancelling & Redirecting</title>
      <script type="text/javascript">
      // display alert with event object info
      function revealEvent(elem, evt) {
         var msg = "Event (from " + evt.target.tagName + " at ";
         msg += evt.clientX + "," + evt.clientY + ") is now at the ";
         msg += elem + " element.";
         alert(msg);
      }
      function init() {
         document.onclick = docEvent;
         document.body.onclick = docBodEvent;
      }
      function docEvent(evt) {
         revealEvent("document", evt);
      }
      function docBodEvent(evt) {
         revealEvent("BODY", evt);
      }
      function buttonEvent(form, evt) {
         revealEvent("BUTTON", evt);
         // redirect if checked
         if (form.redirect.checked) {
            document.body.dispatchEvent(evt);
         }
         // cancel if checked
         if (form.bubbleCancelState.checked) {
            evt.stopPropagation();
         }
      }
      </script>
   </head>
   <body onload="init()">
      <h1>Event Cancelling & Redirecting</h1>
      <hr />
      <form onclick="revealEvent('FORM', event)">
         <p><button name="main1" onclick="buttonEvent(this.form, event)">Button
            'main1'</button></p>
         <p><input type="checkbox" name="bubbleCancelState"
            onclick="event.stopPropagation()" />Cancel Bubbling at BUTTON<br />
            <input type="checkbox" name="redirect"
            onclick="event.stopPropagation()" /> Redirect Event to BODY</p>
      </form>
   </body>
</html>
```

Referencing the event object

While there may be essentially three different event object models in today's browsers, the way your scripts access those objects is divided into two camps: the IE way and the W3C (NN4+/Moz1+/Safari1+) way. I start with the simpler, IE way.

IE4+ event object references

In IE4+, the `event` object is accessible as a property of the `window` object:

```
window.event
```

But, as you are well aware, the `window` part of references is optional, so your scripts can treat the `event` object as if it were a global reference:

```
event.propertyName
```

Thus, any statement in an event handler function can access the `event` object without any special preparation or initializations.

W3C event object references

The situation is a bit more complicated in the W3C event model. In some cases you must explicitly pass the event object as a parameter to an event handler function, while in other cases, the event object is delivered as a parameter automatically. The difference depends on how the event handler function is bound to the object.

Using the original way of binding event handlers to objects — via an attribute in the element's tag — you must specify the event object as a parameter by passing `event` as a parameter, as in

```
onclick="doSomething(event)"
```

This is the only time in the W3C model that you see an explicit reference to the `event` (lowercase "e") object as if it were a global reference. This reference does not work in any other context — only as a parameter to an event handler function. If you have multiple parameters, the `event` reference can go in any order, but I tend to put it last:

```
onclick="doSomething(this, event)"
```

The function definition that is bound to the element should therefore have a parameter variable in place to "catch" the event object parameter:

```
function doSomething(widget, evt) {...}
```

You have no restrictions on how you name this parameter variable. In some examples of this book, you may see the variable assigned as `event` or, more commonly, `evt`. When working with cross-browser scripts, avoid using `event` as a parameter variable name so as not to interfere with IE's `event` property.

Other ways of binding event handler functions to objects — via property assignments and the W3C DOM `addEventListener()` method — assign references of those handlers to the desired objects in the document, as in either of the following:

```
document.forms[0].someButton.onclick = doSomething;
document.getElementById("myButton").addEventListener("click", doSomething,
   false);
```

Event binding through these approaches prevents explicit passage of your own parameters to the invoked functions. But W3C browsers automatically pass as the sole parameter a

reference to the event object created in response to the user or system action that triggered the event. This means that your functions should "receive" the passed event object in a parameter variable:

```
function doSomething(evt) {...}
```

Recall that the `event` object contains a reference to the object that was the target of the event. From that, you can access any properties of that object, such as the `form` object that contains a form control object.

You can see the way the `event` object is passed as a parameter in Listing 25-9. For all event handlers that are assigned by reference (both to an event handler property of an object and to an `addEventListener()` method call), the functions have a parameter variable in place to act as a reference to the event object for statements within the function. If you need to invoke other functions from there, you can pass the `event` object reference further along as needed. The `event` object retains its properties as long as the chain of execution triggered by the event action continues.

event Object Compatibility

Despite the incompatible ways that W3C DOM and IE `event` objects arrive at an event handler function, you can easily stuff the object into one variable that both browser types can use. For example, the following function fragment receives a W3C DOM `event` object but also accommodates the IE `event` object:

```
function doSomething(evt) {
   evt = (evt) ? evt : ((window.event) ? window.event : null);
   if (evt) {
      // browser has an event to process
      ...
   }
}
```

If an `event` object arrives as a parameter, it continues to be available as `evt`; but if not, the function makes sure that a `window.event` object is available and assigns it to the `evt` variable; finally, if the browser doesn't know about an `event` object, the `evt` variable is made `null`. Processing continues only if `evt` contains an event object.

That's the easy part. The madness comes in the details: reading properties of the event object when the property names can vary widely across the three event object models. Sections later in this chapter provide details of each property and method of all three event object models, but seeing an overview of the property terminology on a comparative basis is helpful. Table 25-2 lists the common information bits and actions you are likely to want from an event object and the property or method names used in the three event object models.

Table 25-2: Common event Object Properties and Methods

Property/Action	*NN4*	*IE4+*	*W3C DOM*
Target element	`target`	`srcElement`	`target`
Event type	`type`	`type`	`type`
X coordinate in element	n/a†	`offsetX`	n/a†

Continued

Table 25-2 *(continued)*

Property/Action	*NN4*	*IE4+*	*W3C DOM*
Y coordinate in element	n/a†	offsetY	n/a†
X coordinate in positioned element	layerX	x	layerX
Y coordinate in positioned element	layerY	y	layerY
X coordinate on page	pageX	n/a†	pageX
Y coordinate on page	pageY	n/a†	pageY
X coordinate in window	n/a	clientX	clientX
Y coordinate in window	n/a	clientY	clientY
X coordinate on screen	screenX	screenX	screenX
Y coordinate on screen	screenY	screenY	screenY
Mouse button	which	button	button
Keyboard key	which	keyCode	keyCode
Shift key pressed	modifiers	shiftKey	shiftKey
Alt key pressed	modifiers	altKey	altKey
Ctrl key pressed	modifiers	ctrlKey	ctrlKey
Previous Element	n/a	fromElement	relatedTarget
Next Element	n/a	toElement	relatedTarget
Cancel bubbling	n/a	cancelBubble	preventBubble()
Prevent default action	return false	returnValue	preventDefault()

†Value can be derived through calculations with other properties.

As you can see in Table 25-2, properties for the IE4+ and NN6+/W3C event objects have a lot in common. This is good news, given that the installed base of NN4 users has diminished dramatically thanks to newer browsers. The primary incompatibility is how to reference the element that is the intended target of the event. This, too, can be branched in your code to achieve a common variable that references the element. For example, embedded within the previous function fragment can be a statement, such as the following:

```
var elem = (evt.target) ? evt.target : evt.srcElement;
```

Each event model has additional properties that are not shared by the other. Details about these are covered in the rest of this chapter.

Dueling Event Models

Despite the sometimes widely divergent ways event object models treat their properties, accommodating a wide range of browsers for event manipulation is not difficult. In this section, you see two scripts that examine important event properties. The first script reveals which, if any, modifier keys are held down during an event; the second script extracts the codes for both mouse buttons and keyboard keys. Both scripts work with all browsers that have event objects, including NN4. If your audience no longer uses NN4, you can eliminate the code branches that support it.

Cross-platform modifier key check

Listing 25-10 demonstrates branching techniques for examining the modifier key(s) being held down while an event fires. You can find details of the event object properties, such as `modifiers` and `altKey`, later in this chapter. To see the page in action, click a link, type into a text box, and click a button while holding down any combination of modifier keys (the link is all that works in Safari 1.0). A series of four checkboxes representing the four modifier keys is at the bottom. As you click or type, the checkbox(es) of the pressed modifier key(s) become checked.

Listing 25-10: **Checking Events for Modifier Keys**

```
<html>
   <head>
      <title>Event Modifiers</title>
      <script type="text/javascript">
      function checkMods(evt) {
         evt = (evt) ? evt : ((window.event) ? window.event : null);
         if (evt) {
            var elem = (evt.target) ? evt.target : evt.srcElement;
            var form = document.output;
            if (evt.modifiers) {
               form.modifier[0].checked = evt.modifiers & Event.ALT_MASK;
               form.modifier[1].checked = evt.modifiers & Event.CONTROL_MASK;
               form.modifier[2].checked = evt.modifiers & Event.SHIFT_MASK;
               form.modifier[3].checked = evt.modifiers & Event.META_MASK;
            } else {
               form.modifier[0].checked = evt.altKey;
               form.modifier[1].checked = evt.ctrlKey;
               form.modifier[2].checked = evt.shiftKey;
               form.modifier[3].checked = false;
            }
         }
         return false;
      }
      </script>
   </head>
   <body>
      <h1>Event Modifiers</h1>
      <hr />
      <p>Hold one or more modifier keys and click on <a
         href="javascript:void(0)" onmousedown="return checkMods(event)">this
         link</a> to see which keys you are holding.</p>
      <form name="output">
         <p>Enter some text with uppercase and lowercase letters: <input
            type="text" size="40" onkeyup="checkMods(event)" /></p>
         <p><input type="button" value="Click Here With Modifier Keys"
            onclick="checkMods(event)" /></p>
         <p><input type="checkbox" name="modifier" />Alt <input type="checkbox"
            name="modifier" />Control <input type="checkbox"
            name="modifier" />Shift <input type="checkbox"
            name="modifier" />Meta</p>
      </form>
   </body>
</html>
```

Because all three event handlers call the same `checkMods()` function, branching is needed only in this function. Notice, though, that branching is done by object detection, rather than `navigator.userAgent` detection. This method makes the most sense for this example, because the scripts rely on the existence of particular objects and properties for their proper execution. For NN4, the event object is passed as a parameter (`evt`) whose `modifiers` property is Bitwise ANDed with an `Event` object constant for each modifier key. For IE4+ and NN6+/W3C, the script checks the event object property for each of three modifiers.

Cross-platform key capture

To demonstrate keyboard events in both browsers, Listing 25-11 captures the key character being typed into a text box, as well as the mouse button used to click a button. As with Listing 25-10, NN4 has a very different way of getting this information compared to IE4+ and NN6+/W3C. In this arena, however, NN6+ continues to support the NN4 syntax as well, so you can use the old or new syntax as you like. Whereas NN4 combines the features of key character code and mouse button into one event object property (depending on the event type), newer browsers have entirely separate properties for these values. Listing 25-11 is written such that NN6+ follows the NN4 syntax path, but even if the NN4 syntax should disappear in a future NN version, the browser would follow the new syntax path without blinking an eye.

Listing 25-11: **Checking Events for Key and Mouse Button Pressed**

```
<html>
  <head>
     <title>Button and Key Properties</title>
     <script type="text/javascript">
     function checkWhich(evt) {
        evt = (evt) ? evt : ((window.event) ? window.event : null);
        if (evt) {
           var thingPressed = "";
           var elem = (evt.target) ? evt.target : evt.srcElement;
           if (elem.type == "textarea") {
              thingPressed = (evt.charCode) ? evt.charCode :
                 ((evt.which) ? evt.which : evt.keyCode);
           } else if (elem.type == "button") {
              thingPressed = (typeof evt.button != "undefined") ? evt.button :
                 ((typeof evt.which != "undefined") ? evt.which : "n/a");
           }
           window.status = thingPressed;
        }
        return false;
     }
     </script>
  </head>
  <body>
     <h1>Button and Key Properties</h1>
     (results in the status bar)
     <hr />
     <form>
        <p>Mouse down atop this <input type="button" value="Button"
           onmousedown="checkWhich(event)" /> with either mouse button (if you
           have more than one).</p>
```

```
        <p>Enter some text with uppercase and lowercase letters: <textarea
           cols="40" rows="4" onkeypress="checkWhich(event)"
           wrap="virtual"></textarea></p>
      </form>
   </body>
</html>
```

The codes displayed for the keyboard event are equivalent to the ASCII values of character keys. If you need the codes of other keys, the `onkeydown` and `onkeyup` event handlers provide Unicode values for any key that you press on the keyboard. See the `charCode` and `keyCode` property listings for event objects later in this chapter for more details.

Event Types

Although browsers prior to version 4 did not have an accessible event object, this is a good time to summarize the evolution of what in today's browsers is known as the `type` property. The `type` property reveals the kind of event that generates an event object (the event handler name minus the "on"). Object models in IE4+ and NN6+/W3C provide event handlers for virtually every HTML element, so that it's possible, for example, to define an `onclick` event handler for not only a clickable button but also a `p` or even an arbitrary `span` element. We'll come back to the current crop of browsers in a moment. But first, in case you must write scripts that work on older browsers, you need to know which elements in those browsers support which event handlers. This knowledge will help you determine a common denominator of event handlers to implement in your pages, based on the browsers you anticipate will be accessing the pages.

Older browsers

Earlier browsers tended to limit the number of event handlers for any particular element to just those that made sense for the kind of element it was. Even so, many scripters wanted more event handlers on more objects. But until that became a reality in IE4+ and NN6+/W3C, authors had to know the limits of the object models. Table 25-3 shows the event handlers available for objects within three generations of early browsers. Each column represents the version in which the event type was introduced. For example, the `window` object started out with four event types and gained three more when NN4 was released. In contrast, the `area` object was exposed as an object for the first time in NN3, which is where the first event types for that object are listed.

Table 25-3: Event Types through the Early Ages

Object	*NN2/IE3*	*NN3*	*NN4*
`window`	`blur`		`dragdrop`
	`focus`		`move`
	`load`		`resize`
	`unload`		
`layer`			`blur`
			`focus`

Continued

Table 25-3 *(continued)*

Object	NN2/IE3	NN3	NN4
			load
			mouseout
			mouseover
			mouseup
link	click	mouseout	dblclick
	mouseover		mousedown
			onmouseup
area		mouseout	click
		mouseover	
image		abort	
		error	
		load	
Form	submit	reset	
text, textarea, password			
	blur		keydown
	change		keypress
	focus		keyup
	select		
all buttons	click		mousedown
			mouseup
select	blur		
	change		
	focus		
fileUpload		blur	
		focus	
		select	

With the exception of the NN4 `layer` object, all objects shown in Table 25-3 have survived into the newer browsers, so that you can use these event handlers with confidence. Again, keep in mind that of the browsers listed in Table 25-3, only NN4 has an `event` object of any kind exposed to scripts.

Event types in IE4+ and NN6+/W3C

By now you should have at least scanned the list of event handlers defined for elements in common, as shown in Chapter 15. This list of event types is enormous. A sizable number of the event types are unique to IE4, IE5, and IE5.5/6, and in some cases, just the Windows version at that.

If you compose pages for both IE4+ and NN6+/W3C, however, you need to know which event types these browser families and generations have in common. Event types for

NN6+/Moz1+/Safari are based primarily on the W3C DOM Level 2 specification, although they also include keyboard events, which are not formally part of the Level 2 specification. Table 25-4 lists a common denominator of event types for modern browsers and the objects that support them. As you can see, many of these event types and corresponding objects go way back to the beginning. The biggest change is that mouse events are available for any visible element. While not as long as the IE event list, the event types in Table 25-4 are the basic set you should get to know for all browsers.

Table 25-4: IE4+ and W3C DOM Event Types in Common

Event type	*Applicable Elements*
abort	object
blur	window, button, text, password, label, select, textarea
change	text, password, textarea, select
click	All elements
error	window, frameset, object
focus	window, button, text, password, label, select, textarea
keydown	text, password, textarea
keypress	text, password, textarea
keyup	text, password, textarea
load	window, frameset, object
mousedown	All elements
mousemove	All elements
mouseout	All elements
mouseover	All elements
mouseup	All elements
reset	form
resize	window
scroll	window
select	text, password, textarea
submit	form
unload	window, frameset

NN4 event Object

Properties	*Methods*	*Event Handlers*
data		
layerX		
layerY		
modifiers		

Continued

(NN4) *eventObject*

Properties	*Methods*	*Event Handlers*
pageX		
pageY		
screenX		
screenY		
target		
type		
which		

Syntax

Accessing NN4 event object properties:

eventObject.property

Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

About this object

Most of the details about this object were covered in the comparative event object discussions earlier in this chapter. As the NN4 browser continues to dissipate from the user-installed base, this object and its details will become less important.

Properties

data

Value: Array of strings. Read-Only
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

A `dragdrop` event contains information about the URL string being dragged to the browser window. Because dragging multiple items to a window is possible (for example, many icons representing URLs on some operating systems), the value of the property is an array of strings, with each string containing a single URL (including `file://` URLs for computer files).

URL information such as this is deemed to be private data, so it is exposed only to signed scripts after the user has granted permission to read browser data. If you want your signed script to capture this information without loading the URL into the window, the event handler must evaluate to return `false`.

Example

The page in Listing 25-12 contains little more than a `textarea` in which the URLs of dragged items are listed. To run this script without signing the scripts, turn on codebase principals, as directed in Chapter 46 on the CD-ROM.

To experiment with this listing, load the page and drag any desktop icons that represent files, applications, or folders to the window. Select multiple items and drag them all at once. Because the `ondragdrop` event handler evaluates to `return false`, the files are not loaded into the window. If you want merely to look at the URL and allow only some to process, you

would generate an `if...else` construction to return `true` or `false` to the event handler as needed. A value of `return true` allows the normal processing of the `dragdrop` event to take place after your event handler function has completed its processing.

Listing 25-12: **Obtaining URLs of a dragdrop Event's data Property**

```
<html>
   <head>
      <title>Drag and Drop</title>
      <script type="text/javascript">
      function handleDrag(evt) {
         netscape.security.PrivilegeManager.enablePrivilege(
            "UniversalBrowserRead");
         var URLArray = evt.data;
         netscape.security.PrivilegeManager.disablePrivilege(
            "UniversalBrowserRead");
         if (URLArray) {
            document.forms[0].output.value = URLArray.join("\n");
         } else {
            document.forms[0].output.value = "Nothing found.";
         }
         return false;
      }
      </script>
   </head>
   <body ondragdrop="return handleDrag(event)">
      <b>Drag a URL to this window (NN4 only).</b>
      <hr />
      <form>
         URLs:<br />
         <textarea name="output" cols="70" rows="4"></textarea><br />
         <input type="reset" />
      </form>
   </body>
</html>
```

layerX
layerY
pageX
pageY
screenX
screenY

Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

For many (but not all) mouse-related events, the NN4 event object contains a lot of information about the coordinates of the pointer when the event occurred. In the most complex case, a click in a `layer` object has three distinct pairs of horizontal and vertical (`x` and `y`) coordinate values relative to the layer, the page, and the entire screen. If no layers are specified for a document, the layer and page coordinate systems are identical. Note that these values are

merely geographical in nature and do not, by themselves, contain any information about the object being clicked (information held by the *eventObject*.target property).

These mouse coordinate properties are set only with specific events. In the case of a link object, the click and all four mouse events pack these values into the event object. For buttons, however, only the mouse events (mousedown and mouseup) receive these coordinates.

Each of the two window event types (move and resize) uses one of these property pairs to convey the results of the user action involved. For example, when the user resizes a window, the resize event stuffs the *eventObject*.layerX and *eventObject*.layerY properties with the inner width and height (that is, the content area) of the browser window (you can also use the optional *eventObject*.width and *eventObject*.height property names if you prefer). When the user moves the window, the *eventObject*.screenX and *eventObject*.screenY properties contain the screen coordinates of the top-left corner of the entire browser application window.

Example

You can see the effects of the coordinate systems and associated properties with the page in Listing 25-13. Part of the page contains a three-field readout of the layer-, page-, and screen-level properties. Two clickable objects are provided so that you can see the differences between an object not in any layer and an object residing within a layer. The object not confined by a layer has the same coordinates for the layer and the page in the event object properties.

Additional readouts display the event object coordinates for resizing and moving a window. If you maximize the window under Windows, the Navigator browser's top-left corner is actually out of sight, four pixels up and to the left. That's why the screenX and screenY values are both -4.

Listing 25-13: **NN4 Event Coordinate Properties**

```
<html>
   <head>
      <title>X and Y Event Properties</title>
      <script type="text/javascript">
      function checkCoords(evt) {
         var form = document.forms[0];
         form.layerCoords.value = evt.layerX + "," + evt.layerY;
         form.pageCoords.value = evt.pageX + "," + evt.pageY;
         form.screenCoords.value = evt.screenX + "," + evt.screenY;
         return false;
      }
      function checkSize(evt) {
         document.forms[0].resizeCoords.value = evt.layerX + "," + evt.layerY;
      }
      function checkLoc(evt) {
         document.forms[0].moveCoords.value = evt.screenX + "," + evt.screenY;
      }
      </script>
   </head>
   <body onresize="checkSize(event)" onmove="checkLoc(event)">
      <h1>X and Y Event Properties (NN4)</h1>
      <hr />
```

```
      <p>Click on the button and in the layer/image to see the coordinate
         values for the event object.</p>
      <form name="output">
         <table>
            <tr>
               <td colspan="2">Mouse Event Coordinates:</td>
            </tr>
            <tr>
               <td align="right">layerX, layerY:</td>
               <td><input type="text" name="layerCoords" size="10" /></td>
            </tr>
            <tr>
               <td align="right">pageX, pageY:</td>
               <td><input type="text" name="pageCoords" size="10" /></td>
            </tr>
            <tr>
               <td align="right">screenX, screenY:</td>
               <td><input type="text" name="screenCoords" size="10" /></td>
            </tr>
            <tr>
               <td align="right"><input type="button" value="Click Here"
                  onmousedown="checkCoords(event)" /></td>
            </tr>
            <tr>
               <td colspan="2"><hr /></td>
            </tr>
            <tr>
               <td colspan="2">Window Resize Coordinates:</td>
            </tr>
            <tr>
               <td align="right">layerX, layerY:</td>
               <td><input type="text" name="resizeCoords" size="10" /></td>
            </tr>
            <tr>
               <td colspan="2"><hr /></td>
            </tr>
            <tr>
               <td colspan="2">Window Move Coordinates:</td>
            </tr>
            <tr>
               <td align="right">screenX, screenY:</td>
               <td><input type="text" name="moveCoords" size="10" /></td>
            </tr>
         </table>
      </form>
      <layer name="display" bgcolor="coral" top="140" left="300" height="250"
      width="330">
         <a href="javascript:void(0)" onclick="return checkCoords(event)"><img
         alt="image" src="nile.gif" width="320" height="240" border="0" /></a>
      </layer>
   </body>
</html>
```

Related Items: `window` and `layer` object move and resize methods.

modifiers

Value: Constant. Read-Only
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

The `modifiers` property of the NN4 event object refers to the modifier keys that can be pressed while clicking or typing. Modifier keys are Alt (also the Option key on the Macintosh keyboard), Ctrl, Shift, and what is known as a meta key (for example, the Command key, ⌘, on the Macintosh keyboard). You can use this property to find out if one or more modifier keys were pressed at the time the event occurred.

Values for these keys are integer values designed in such a way that any combination of keys generates a unique value. Fortunately, you don't have to know anything about these values, because the event model supplies some plain-language constants (properties of a global `Event` object always available behind the scenes) that a script can apply to the property value passed with the object. The constant names consist of the key name (all uppercase), followed by an underscore and the uppercase word `MASK`. For example, if the Alt key is pressed by itself or in concert with other modifier keys, you can use the bitwise `and` operator (&) and the `Event.ALT_MASK` constant to test for the presence of the Alt key in the property value:

```
function handleMyEvent(evt) {
   if (evt.modifiers & Event.ALT_MASK) {
      //statements for Alt key handling
   }
}
```

Modifiers are not available with every event. You can capture them with `mousedown` and `mouseup` events in buttons and links. The only `click` event offering modifiers is with button objects. Keyboard events in text objects also include these modifiers. But be aware that accelerated keyboard combinations (for example, Ctrl+Q/⌘-Q for Quit) are not trappable by JavaScript event mechanisms because they are reserved for the browser's own menu shortcuts.

Example

See Listing 25-10 earlier in this chapter to see (in a cross-browser way) how the modifier keys are read for NN4.

target

Value: Object reference. Read-Only
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

Every event has a property containing a reference to the object that was clicked, typed into, or otherwise acted upon. Most commonly, this property is examined when you set up a page to trap for events at the window, document, or layer level, as described earlier in this chapter. The `target` property lets you better identify the intended destination of the event while handling all processing for that type of event in one place. With a reference to the target object at hand in this property, your scripts can extract and/or set properties of the object directly.

(IE) event

type

Value: String. Read-Only
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

An event object's type is the name of the event that generated the event object. An event name is the same as the event handler's name, without the "on" prefix. Therefore, if a button's `onclick` event handler is triggered by a user's click, then the event type is `click` (all lowercase). If you create a multipurpose function for handling events, you can extract the *`eventObject`*`.type` property to help the function decide how to handle the current event. This sounds like a good job for the `switch` control structure (see Chapter 31).

which

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

The value of the `which` property depends on the event type: a mouse button indicator for mouse events and a character key code for keyboard events.

For a mouse-related event, the *`eventObject`*`.which` property contains either a 1 for the left (primary) mouse button or a 3 for the right (secondary) mouse button. Most Macintosh computers have only a one-button mouse, so exercise care in designing pages that rely on the second mouse button. Even on Windows and other platforms, you must program an object's `onmousedown` event handler to `return false` for the secondary button to be registered instead of a browser pop-up menu appearing onscreen.

Keyboard events generate the ISO-Latin character code for the key that has been pressed. This value is an integer between 0 and 255. If your script needs to look at the actual character being typed, rather than the key code, use the `String.fromCharCode()` method (see Chapter 27) to make the conversion. If you have difficulty obtaining character codes from keyboard events, try using the `onkeydown` and `onkeyup` events rather than `onkeypress`. In either case, the function keys do not present character codes.

Example

See Listing 25-10 for an example of using the *`eventObject`*`.which` property.

IE4+ event Object

Properties	*Methods*	*Event Handlers*
`altKey`		
`altLeft`		
`behaviorCookie`		
`behaviorPart`		
`bookmarks`		
`boundElements`		
`button`		

Continued

(IE) event

Properties	*Methods*	*Event Handlers*
cancelBubble		
clientX		
clientY		
contentOverflow		
ctrlKey		
ctrlLeft		
dataFld		
dataTransfer		
fromElement		
keyCode		
nextPage		
offsetX		
offsetY		
propertyName		
qualifier		
reason		
recordset		
repeat		
returnValue		
saveType		
screenX		
screenY		
shiftKey		
shiftLeft		
srcElement		
srcFilter		
srcUrn		
toElement		
type		
wheelData		
x		
y		

Syntax

Accessing IE4+ `event` object properties:

`[window.]event.`*`property`*

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

About this object

The IE4+ `event` object is a property of the `window` object. Its basic operation is covered earlier in this chapter.

You can see a little of what the `event` object is about with the help of The Evaluator (see Chapter 13). If you type `event` into the bottom text box, you can examine the properties of the `event` object for the event that triggers the function that displays the `event` object properties. If you press the Enter key in the text box, you see properties of the `keypress` event that caused the internal script to run; click the List Properties button to see the properties of the `click` event fired at the button. Hold down some of the modifier keys while clicking to see how this affects some of the properties.

As you review the properties for the `event` object, make special note of the compatibility rating for each property. The list of properties for this object has grown over the evolution of the IE4+ event object model. Also, most properties are listed here as being read-only, which they were in IE4. But for IE5+, these properties are also Read/Write if the event is created artificially via methods, such as IE5.5+'s `document.createEventObject()` method. Event objects that are created by user or system action have very few properties that can be modified on the fly (to prevent your scripts from altering user actions). Notice, too, that some properties are the same as for the W3C DOM event object, as revealed in the compatibility ratings.

Properties

`altKey`
`ctrlKey`
`shiftKey`

Value: Boolean. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

When an `event` object is created in response to a user or system action, these three properties are set based on whether their corresponding keys were being held down at the time — a Shift-click, for example. If the key was held down, the property is assigned a value of `true`; otherwise the value is `false`.

Most commonly, you use expressions consisting of this property as `if` construction condition statements. Because these are Boolean values, you can combine multiple properties in a single condition. For example, if you have a branch of a function that is to execute only if the event occurred with both the Shift and Control keys held down, the condition looks as the following:

```
if (event.shiftKey && event.ctrlKey) {
   // statements to execute
}
```

Conversely, you can take a more user-friendly approach to provide special processing if the user holds down any one of the three modifier keys:

```
if (event.shiftKey || event.ctrlKey || event.altKey) {
   // statements to execute
}
```

The rationale behind this approach is to offer perhaps some shortcut operation for users, but not force them to memorize a specific modifier key combination.

Example

See Listing 25-10, where the values of these three properties are used to set the `checked` properties of corresponding checkboxes for a variety of event types.

Related Items: `altLeft`, `ctrlLeft`, `shiftLeft` properties.

altLeft
ctrlLeft
shiftLeft

Value: Boolean. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

Some versions of Windows allow events to be modified by only the left-hand Alt, Ctrl, and Shift keys when using IE5.5+. For these modifiers to be recorded by the `event` object, focus must be on the document (body), and not in any form control. If the left-key version is `false` and the regular version is `true`, then your script knows that the right-hand key had been held down during the event.

Related Items: `altKey`, `ctrlKey`, `shiftKey` properties.

behaviorCookie
behaviorPart

Value: Integer. Read-Only
Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-

These two properties are related to a Windows technology that Microsoft calls *rendering behaviors*. Unlike the behaviors discussed under the `addBehavior()` method in Chapter 15, rendering behaviors are written in C++ and provide services for custom drawing on your Web page. For more details, consult the document "Implementing Rendering Behaviors" at `http://msdn.microsoft.com/workshop/browser/editing/imprendbehav.asp`.

bookmarks
boundElements
dataFld
qualifier
reason
recordset

Value: See text. Read-Only
Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-

This group of `event` object properties is tied to using Data Binding in Windows versions of IE4+. Extensive details of Data Binding lie outside the scope of this book, but Table 25-5 provides a summary of these `event` object properties within that context (much of the terminology is used in Data Binding, but doesn't affect other scripting). For more details, search for ActiveX Data Objects (ADO) at `http://msdn.microsoft.com/workshop/`.

Table 25-5: ADO-Related event Object Properties

Property	*Value*	*First Implemented*	*Description*
`bookmarks`	Array	IE4	Array of ADO bookmarks (saved positions) for records within a recordset associated with the object that received the event.
`boundElements`	Array	IE5	Array of element references for all elements bound to the same data set that was touched by the current event.
`dataFld`	String	IE5	Name of the data source column that is bound to a table cell that receives a `cellchange` event.
`qualifier`	String	IE5	Name of the data member associated with a data source that receives a data-related event. Available only if the data source object (DSO) allows multiple-named data members or a qualifier has been explicitly set via the `datasrc` attribute of the bound element. Read-write in IE5+.
`reason`	Integer	IE4	Set only from `onDataSetComplete` event, provides the result code of the data set loading (`0`=successful; `1`=transfer aborted; `2`=other error).
`recordset`	Object	IE4	Reference to the current recordset in a data source object.

button

Value: Integer. Read-Only

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `button` property reveals which button or buttons were pressed to activate a mouse event. If no mouse button is pressed to generate an event, this property is zero. But integers 1 through 7 reveal single and multiple button presses, including three-button mice when they are recognized by the operating system. Integer values correspond to buttons according to the following scheme:

Value	*Description*
`0`	No button
`1`	Left (primary) button
`2`	Right button
`3`	Left and right buttons together
`4`	Middle button

Continued

(IE) event.button

Value	*Description*
5	Left and middle buttons together
6	Right and middle buttons together
7	Left, middle, and right buttons together

Mouse buttons other than the primary one are easier to look for in `mousedown` or `mouseup` events rather than `onclick` events. Be aware that as the user works toward pressing multiple buttons, each press fires a `mousedown` event. Therefore, if the user presses the left button first, the `mousedown` event fires, with the `event.button` property bearing the `1` value; as soon as the right button is pressed, the `mousedown` event fires again, but this time with an `event.button` value of `3`. If your script intends to perform special action with both buttons pressed, it should ignore and not perform any action for a single mouse button, because that one-button event will very likely fire in the process, disturbing the intended action.

Exercise caution when scripting the `event.button` property for both IE4+ and NN6+/W3C. The W3C DOM event model defines different button values for mouse buttons (`0`, `1`, and `2` for left, middle, and right) and no values for multiple buttons.

Example

See Listing 25-11, where the `event.button` property is revealed in the status bar. Try pressing individual mouse buttons on, for example, the screen button. Then try combinations, watching the results very closely in the status bar.

Related Items: None.

`cancelBubble`

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `cancelBubble` property (which sounds more as if it should be a method name) determines whether the current `event` object bubbles up any higher in the element containment hierarchy of the document. By default, this property is `false`, meaning that if the event is supposed to bubble, it will do so automatically.

To prevent event bubbling for the current event, set the property to `true` anywhere within the event handler function. As an alternative, you can cancel bubbling directly in an element's event handler attribute, as in the following:

```
onclick="doButtonClick(this); event.cancelBubble = true"
```

Cancelling event bubbling works only for the current event. The very next event to fire will have bubbling enabled (provided the event bubbles).

Example

See Listing 25-6 to see the `cancelBubble` property in action. Even though that listing has some features that apply to IE5.5+, the bubble cancelling demonstration works all the way back to IE4.

Related Items: `returnValue` property.

clientX
clientY
offsetX
offsetY
screenX
screenY
x
y

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

An IE `event` object provides coordinates for an event in as many as four coordinate spaces: the element itself, the parent element of the event's target, the viewable area of the browser window, and the entire video screen. Unfortunately, misleading values can be returned by some of the properties that correspond to these coordinate spaces, as discussed in this section. Note that no properties provide the explicit position of an event relative to the entire page, in case the user has scrolled the window.

Starting with the innermost space — that of the element that is the target of the event — the `offsetX` and `offsetY` properties should provide pixel coordinates within the target element. This is how, for example, you could determine the click point on an image, regardless of whether the image is embedded in the `body` or floating around in a positioned `div`. Windows versions through IE6 produce the correct values in most cases. But for some elements that are child elements of the `body` element, the vertical (`y`) value may be relative to the viewable window, rather than just the element itself. You can see an example of this when you work with Listing 25-14 and click the `h1` or `p` elements near the top of the page. This problem does not affect MacIE, but there is another problem on Mac versions: If the page is scrolled away from its normal original position, the scrolled values are subtracted from the `clientX` and `clientY` values. This is an incompatibility bug, and you must take this error into account if you need click coordinates inside an element for a potentially scrolled page. This error correction must be done only for the Mac, because Windows works okay.

Extending scope to the offset parent element of the event's target, the `x` and `y` properties in IE5+ for Windows should return the coordinates for the event relative to the target's offset parent element (the element that can be found via the `offsetParent` property). For most non-positioned elements, these values are the same as the `clientX` and `clientY` properties because, as discussed in a moment, the offset parent element has a zero offset with *its* parent, the `body`. Observe an important caution about the `x` and `y` properties: In WinIE4 and through MacIE5, the properties do not take into account any offset parent locations other than the `body`. Even in WinIE5+, this property can give false readings in some circumstances. By and large, these two properties should not be used.

The next set of coordinates, `clientX` and `clientY`, are relative to the visible document area of the browser window. When the document is scrolled all the way to the top (or the document doesn't scroll at all), these coordinates are the same as the coordinates on the entire page. But because the page can scroll "underneath" the viewable window, the coordinates on the page can change if the page scrolls. Also, in the Windows versions of IE, you can actually register mouse events that are up to 2 pixels outside of the `body` element, which seems weird, but true. Therefore, in WinIE, if you click the background of the `body`, the event

fires on the body element, but the clientX/clientY values will be 2 pixels greater than offsetX/offsetY (they're equal in MacIE). Despite this slight discrepancy, you should rely on the clientX and clientY properties if you are trying to get the coordinates of an event that may be in a positioned element, but have those coordinates relative to the entire viewable window, rather than just the positioning context.

Taking the page's scrolling into account for an event coordinate is often important. After all, unless you generate a fixed-size window for a user, you don't know how the browser window will be oriented. If you're looking for a click within a specific region of the page, you must take page scrolling into account. The scrolling factor can be retrieved from the document.body.scrollLeft and document.body.scrollTop properties. When reading the clientX and clientY properties, be sure to add the corresponding scroll properties to get the position on the page:

```
var coordX = event.clientX + document.body.scrollLeft;
var coordY = event.clientY + document.body.scrollTop;
```

Do this in your production work without fail.

Finally, the screenX and screenY properties return the pixel coordinates of the event on the entire video screen. These properties may be more useful if IE provided more window dimension properties. In any case, because mouse events fire only when the cursor is somewhere in the content region of the browser window, don't expect to get screen values of anywhere outside this region.

If these descriptions seem confusing to you, you are not alone. Throw in a few bugs, and it may seem like quite a mess. But think how you may use event coordinates in scripts. By and large, you want to know one of two types of mouse event coordinates: within the element itself and within the page. Use the offsetX/offsetY properties for the former; use clientX/clientY (plus the scroll property values) for the latter.

Although the coordinate properties are used primarily for mouse events, there is a little quirk that may let you determine if the user has resized the window via the maximize icon in the title bar (on the Mac, this is called the zoom box) or the resize handle at the bottom-right corner of the screen. Mouse event coordinates are recorded in the event object for a resize event. In the case of the maximize icon, the clientY coordinate is a negative value (above the client space) and the clientX coordinate is within about 45 pixels of the previous width of the window (document.body.clientWidth). This, of course, happens after the window has resized, so it is not a way to prevent window resizing.

Example

Listing 25-14 provides readings of all event coordinate properties in an interactive way. An onmousedown event handler triggers all event handling, and you can click the mouse anywhere on the page to see what happens. You see the tag of the element targeted by the mouse event to help you visualize how some of the coordinate properties are determined. An image is encased inside a positioned div element to help you see what happens to some of the properties when the event is targeted inside a positioned element.

Listing 25-14: **IE4+ Event Coordinate Properties**

```
<html>
   <head>
      <title>X and Y Event Properties (IE4+ Syntax)</title>
      <script type="text/javascript">
```

(IE) event.clientX

```
      function checkCoords(evt) {
         evt = (evt) ? evt : ((window.event) ? window.event : null);
         if (evt) {
            var elem = (evt.target) ? evt.target : evt.srcElement;
            var form = document.forms[0];
            form.srcElemTag.value = "<" + elem.tagName + ">";
            form.clientCoords.value = evt.clientX + "," + evt.clientY;
            if (typeof document.body.scrollLeft != "undefined") {
               form.pageCoords.value = (evt.clientX + document.body.scrollLeft)
+
                  "," + (evt.clientY + document.body.scrollTop);
            }
            form.offsetCoords.value = evt.offsetX + "," + evt.offsetY;
            form.screenCoords.value = evt.screenX + "," + evt.screenY;
            form.xyCoords.value = evt.x + "," + evt.y;
            if (elem.offsetParent) {
               form.parElem.value = "<" + elem.offsetParent.tagName + ">";
            }
            return false;
         }
      }
      function handleSize(evt) {
         evt = (evt) ? evt : ((window.event) ? window.event : null);
         if (evt) {
            document.forms[0].resizeCoords.value = evt.clientX + "," +
evt.clientY;
         }
      }
      </script>
   </head>
   <body onmousedown="checkCoords(event)" onresize="handleSize(event)">
      <h1>X and Y Event Properties (IE4+ Syntax)</h1>
      <hr />
      <p>Click on any element to see the coordinate values
         for the event object.</p>
      <form name="output">
         <table>
            <tr>
               <td colspan="2">IE Mouse Event Coordinates:</td>
            </tr>
            <tr>
               <td align="right">srcElement:</td>
               <td><input type="text" name="srcElemTag" size="10" /></td>
            </tr>
            <tr>
               <td align="right">clientX, clientY:</td>
               <td><input type="text" name="clientCoords" size="10" /></td>
               <td align="right">...With scrolling:</td>
               <td><input type="text" name="pageCoords" size="10" /></td>
            </tr>
            <tr>
               <td align="right">offsetX, offsetY:</td>
               <td><input type="text" name="offsetCoords" size="10" /></td>
            </tr>
            <tr>
               <td align="right">screenX, screenY:</td>
               <td><input type="text" name="screenCoords" size="10" /></td>
            </tr>
```

Continued

Listing 25-14 *(continued)*

```
            <tr>
               <td align="right">x, y:</td>
               <td><input type="text" name="xyCoords" size="10" /></td>
               <td align="right">...Relative to:</td>
               <td><input type="text" name="parElem" size="10" /></td>
            </tr>
            <tr>
               <td align="right"><input type="button" value="Click Here" /></td>
            </tr>
            <tr>
               <td colspan="2"><hr /></td>
            </tr>
            <tr>
               <td colspan="2">Window Resize Coordinates:</td>
            </tr>
            <tr>
               <td align="right">clientX, clientY:</td>
               <td><input type="text" name="resizeCoords" size="10" /></td>
            </tr>
         </table>
      </form>
      <div id="display" style="position:relative; left:100">
         <img alt="image" src="nile.gif" width="320" height="240" border="0" />
      </div>
   </body>
</html>
```

Here are some tasks to try with the page that loads from Listing 25-14 to help you understand the relationships among the various pairs of coordinate properties:

1. Click the dot above the "i" on the "Click Here" button label. The target element is the button (`input`) element, whose `offsetParent` is a table cell element. The `offsetY` value is very low because you are near the top of the element's own coordinate space. The client coordinates (and `x` and `y`), however, are relative to the viewable area in the window. If your browser window is maximized in Windows, the `screenX` and `clientX` values will be the same; the difference between `screenY` and `clientY` is the height of all the window chrome above the content region. With the window not scrolled at all, the client coordinates are the same with and without scrolling taken into account.

2. Jot down the various coordinate values and then scroll the page down slightly (clicking the scrollbar fires an event) and click the dot on the button again. The `clientY` value shrinks because the page has moved upward relative to the viewable area, making the measure between the top of the area smaller with respect to the button. The Windows version does the right thing with the offset properties, by continuing to return values relative to the element's own coordinate space; the Mac, unfortunately, subtracts the scrolled amount from the offset properties.

3. Click the large image. The client properties perform as expected for both Windows and Mac, as do the screen properties. For Windows, the `x` and `y` properties correctly return the event coordinates relative to the `img` element's `offsetParent`, which is the `div` element that surrounds it. Note, however, that the browser "sees" the `div` as starting

10 pixels to the left of the image. In WinIE5.5+, you can click within those 10 transparent pixels to the left of the image to click the `div` element. This padding is inserted automatically and impacts the coordinates of the `x` and `y` properties. A more reliable measure of the event inside the image is the offset properties. The same is true in the Macintosh version, as long as the page isn't scrolled, in which case the scroll, just as in Step 2, affects the values above.

4. Click the top `hr` element under the heading. It may take a couple of tries to actually hit the element (you've made it when the `hr` element shows up in the `srcElement` box). This is to reinforce the way the client properties provide coordinates within the element itself (again, except on the Mac when the page is scrolled). Clicking at the very left end of the rule, you eventually find the 0,0 coordinate.

Finally, if you are a Windows user, here are two examples to try to see some of the unexpected behavior of coordinate properties.

1. With the page not scrolled, click anywhere along the right side of the page, away from any text so that the `body` element is `srcElement`. Because the `body` element theoretically fills the entire content region of the browser window, all coordinate pairs except for the screen coordinates should be the same. But offset properties are 2 pixels less than all the others. By and large, this difference won't matter in your scripts, but you should be aware of this potential discrepancy if precise positioning is important. For inexplicable reasons, the offset properties are measured in a space that is inset 2 pixels from the left and top of the window. This is not the case in the Macintosh version, where all value pairs are the same from the `body` perspective.

2. Click the text of the `h1` or `p` elements (just above and below the long horizontal rule at the top of the page). In theory, the offset properties should be relative to the rectangles occupied by these elements (they're block elements, after all). But instead, they're measured in the same space as the client properties (plus the 2 pixels). This unexpected behavior doesn't have anything to do with the cursor being a text cursor, because if you click inside any of the text box elements, their offset properties are properly relative to their own rectangles. This problem does not afflict the Macintosh version.

Many of these properties are also in the W3C DOM and are therefore supported in W3C DOM browsers. Unsupported properties display their values as `undefined` when you run Listing 25-14 in those browsers.

You can see further examples of important event coordinate properties in action in the discussion of dragging elements around the IE page in Chapter 39 on the CD-ROM.

Related Items: `fromElement`, `toElement` properties.

dataTransfer

Value: Object. Read-Only
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `dataTransfer` property is a reference to a WinIE-only object called the `dataTransfer` object. Use this object in drag-and-drop operations (that is, with drag-and-drop-related events) to control not only the data that gets transferred from the source to the target but also to control the look of the cursor along the way.

Table 25-6 lists the properties and methods of the `dataTransfer` object.

(IE) event.dataTransfer

Table 25-6: dataTransfer object Properties and Methods

Property/Method	*Returns*	*Description*
`dropEffect`	String	An element that is a potential recipient of a drop action can use the `ondragenter`, `ondragover`, or `ondrop` event handler to set the cursor style to be displayed when the cursor is atop the element. Before this can work, the source element's `ondragstart` event handler must assign a value to the `event.effectAllowed` property. Possible string values for both properties are `copy`, `link`, `move`, or `none`. These properties correspond to the Windows system cursors for the operations users typically do with files and in other documents. You must also cancel the default action (meaning set `event.returnValue` to `false`) for all of these drop element event handlers: `ondragenter`, `ondragover`, and `ondrop`.
`effectAllowed`	String	Set in response to an `ondragstart` event of the source element, this property determines which kind of drag-and-drop action will be taking place. Possible string values are `copy`, `link`, `move`, or `none`. This property value must match the `dropEffect` property value for the target element's `event` object. Also, cancel the default action (meaning, set `event.returnValue` to `false`) in the `ondragstart` event handler.
`clearData([`*format*`])`	Nothing	Removes data in the clipboard. If no format parameters are supplied, all data are cleared. Data formats can be one or more of the following strings: `Text`, `URL`, `File`, `HTML`, `Image`.
`getData(`*format*`)`	String	Retrieves data of the specified format from the clipboard. The format is one of the following strings: `Text`, `URL`, `File`, `HTML`, `Image`. The clipboard is not emptied after you get the data, so that it can be retrieved in several sequential operations.
`setData (`*format*`, `*data*`)`	Boolean	Stores string data in the clipboard. The format is one of the following strings: `Text`, `URL`, `File`, `HTML`, `Image`. For non-text data formats, the data must be a string that specifies the path or URL to the content. Returns `true` if the transfer to the clipboard is successful.

The `dataTransfer` object acts as a conduit and controller of data that your scripts need to transfer from one element to another in response to a user's drag-and-drop action. You need to adhere to a well-defined sequence of actions triggered by a handful of event handlers. This means that the object is invoked on different instances of the `event` object as different events fire in the process of dragging and dropping.

The sequence begins at the source element, where an `ondragstart` event handler typically assigns a value to the `dropEffect` property and uses the `getData()` method to explicitly

capture whatever data it is about the source object that gets transferred to the eventual target. For example, if you drag an image, the information being transferred may simply be the URL of the image — data that is extractable from the `event.srcElement.src` property of that event (the `src` property of the image, that is).

At the target element(s), three event handlers must be defined: `ondragenter`, `ondragover`, and `ondrop`. Most commonly, the first two event handlers do nothing more than mark the element for a particular `dropEffect` (which must match the `effectAllowed` set at the source during the drag's start) and set `event.returnValue` to false so that the cursor displays the desired cursor. These actions are also carried out in the `ondrop` event handler, but that is also the handler that does the processing of the destination action at the target element. This is when the `dataTransfer` object's `getData()` method is invoked to pick up the data that has been "stored" away by `getData()` at the start of the drag. If you also want to make sure that the data is not picked up accidentally by another event, invoke the `clearData()` method to remove that data from memory.

Note that the style of dragging being discussed here is not the kind in which you see the source element actually moving on the screen (although you could script it that way). The intention is to treat drag-and-drop operations just as Windows does in, say, the Windows Explorer window or on the Desktop. To the user, the draggable component becomes encapsulated in the cursor. That's why the properties of the `dataTransfer` object control the appearance of the cursor at the drop point as a way of conveying to the user the type of action that will occur with the impending drop.

Example

An extensive example of the `dataTransfer` property in action can be found in Listing 15-37 in the section for the `ondrag` event handler.

Related Items: `ondragend`, `ondragenter`, `ondragleave`, `ondragover`, `ondragstart`, `ondrop` event handlers.

`fromElement`
`toElement`

Value: Element object. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `fromElement` and `toElement` properties allow an element to uncover where the cursor rolled in from or has rolled out to. These properties extend the power of the `onmouseover` and `onmouseout` event handlers by expanding their scope to outside the current element (usually to an adjacent element).

When the `onmouseover` event fires on an element, the cursor had to be over some other element just beforehand. The `fromElement` property holds a reference to that element. Conversely, when the `onmouseout` event fires, the cursor is already over some other element. The `toElement` property holds a reference to that element.

Example

Listing 25-15 provides an example of how the `fromElement` and `toElement` properties can reveal the life of the cursor action before and after it rolls into an element. When you roll the cursor to the center box (a table cell), its `onmouseover` event handler displays the text from the table cell from which the cursor arrived.

(IE) event.fromElement

Listing 25-15: **Using the toElement and fromElement Properties**

```
<html>
   <head>
      <title>fromElement and toElement Properties</title>
      <style type="text/css">
      .direction {background-color:#00FFFF; width:100; height:50;
      text-align:center}
      #main {background-color:#FF6666; text-align:center}
      </style>
      <script type="text/javascript">
      function showArrival() {
         var direction = (event.fromElement.innerText) ?
            event.fromElement.innerText : "parts unknown";
         status = "Arrived from: " + direction;
      }
      function showDeparture() {
         var direction = (event.toElement.innerText) ?
            event.toElement.innerText : "parts unknown";
         status = "Departed to: " + direction;
      }
      </script>
   </head>
   <body>
      <h1>fromElement and toElement Properties</h1>
      <hr />
      <p>Roll the mouse to the center box and look for arrival information in
         the status bar. Roll the mouse away from the center box and look for
         departure information in the status bar.</p>
      <table cellspacing="0" cellpadding="5">
         <tr>
            <td></td>
            <td class="direction">North</td>
            <td></td></tr>
         <tr>
            <td class="direction">West</td>
            <td id="main" onmouseover="showArrival()"
            onmouseout="showDeparture()">Roll</td>
            <td class="direction">East</td>
         </tr>
         <tr>
            <td></td>
            <td class="direction">South</td>
            <td></td>
         </tr>
      </table>
   </body>
</html>
```

This is a good example to experiment with in the browser, because it also reveals a potential limitation. The element registered as the `toElement` or `fromElement` must fire a mouse event to register itself with the browser. If not, the next element in the sequence that registers itself is the one acknowledged by these properties. For example, if you roll the mouse into the center box and then extremely quickly roll the cursor to the bottom of the page, you may bypass the South box entirely. The text that appears in the status bar is actually the inner text of the

body element, which is the element that caught the first mouse event to register itself as the toElement for the center table cell.

Related Items: srcElement property.

keyCode

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

For keyboard events, the keyCode property returns an integer corresponding to the Unicode value of the character (for onkeypress events) or the keyboard character key (for onkeydown and onkeyup events). There is a significant distinction between these numbering code systems.

If you want the Unicode values (the same as ASCII values for the Latin character set) for the key that a user pressed, get the keyCode property from the onkeypress event handler. For example, a lowercase "a" returns 97, while an uppercase "A" returns 65. Non-character keys, such as arrows, page navigation, and function keys, return a null value for the keyCode property during onkeypress events. In other words, the keyCode property for onkeypress events is more like a character code than a key code.

To capture the exact keyboard key that the user presses, use either the onkeydown or onkeyup event handler. For these events, the event object captures a numeric code associated with a particular key on the keyboard. For the character keys, this varies with the language assigned as the system language. Importantly, there is no distinction between uppercase or lowercase: The "A" key on the Latin keyboard returns a value of 65, regardless of the state of the Shift key. At the same time, however, the press of the Shift key fired its own onkeydown and onkeyup events, setting the keyCode value to 16. Other non-character keys — arrows, page navigation, function, and similar — have their own codes as well. This gets very detailed, including special key codes for the numeric keyboard keys that are different from their corresponding numbers along the top row of the alphanumeric keyboard.

Be sure to see the extensive section on keyboard events in Chapter 15 for examples of how to apply the keyCode property in applications.

Example

Listing 25-16 provides an additional play area to view the keyCode property for all three keyboard events while you type into a textarea. You can use this page later as an authoring tool to grab the precise codes for keyboard keys you may not be familiar with.

Listing 25-16: **Displaying keyCode Property Values**

```
<html>
   <head>
      <title>keyCode Property</title>
      <style type="text/css">
      td {text-align:center}
      </style>
      <script type="text/javascript">
      function showCode(which) {
         document.forms[0].elements[which].value = event.keyCode
      }
```

Continued

Listing 25-16 *(continued)*

```
      function clearEm() {
         for (var i = 1; i < document.forms[0].elements.length; i++) {
            document.forms[0].elements[i].value = ""
         }
      }
      </script>
   </head>
   <body>
      <h1>keyCode Property</h1>
      <hr />
      <form>
         <p><textarea name="scratchpad" cols="40" rows="5" wrap="hard"
            onkeydown="clearEm(); showCode('down')" onkeyup="showCode('up')"
            onkeypress="showCode('press')">
            </textarea></p>
         <table cellpadding="5">
            <tr>
               <th>Event</th>
               <th>event.keyCode</th>
            </tr>
            <tr>
               <td>onKeyDown:</td>
               <td><input type="text" name="down" size="3" /></td>
            </tr>
            <tr>
               <td>onKeyPress:</td>
               <td><input type="text" name="press" size="3" /></td>
            </tr>
            <tr>
               <td>onKeyUp:</td>
               <td><input type="text" name="up" size="3" /></td>
            </tr>
         </table>
      </form>
   </body>
</html>
```

The following are some specific tasks to try with the page to examine key codes (if you are not using a browser set for English and a Latin-based keyboard, your results may vary):

1. Enter a lowercase "a". Notice how the `onkeypress` event handler shows the code to be 97, which is the Unicode (and ASCII) value for the first of the lowercase letters of the Latin alphabet. But the other two events record just the key's code: 65.
2. Type an uppercase "A" via the Shift key. If you watch closely, you see that the Shift key, itself, generates the code 16 for the `onkeydown` and `onkeyup` events. But the character key then shows the value 65 for all three events, because the ASCII value of the uppercase letter happens to match the keyboard key code for that letter.
3. Press and release the Down Arrow key (be sure the cursor still flashes in the `textarea`, because that's where the keyboard events are being monitored). As a non-character key, it does not fire an `onkeypress` event. But it does fire the other events, and assigns 40 as the code for this key.

4. Poke around with other non-character keys. Some may produce dialog boxes or menus, but their key codes are recorded nonetheless. Note that not all keys on a Macintosh keyboard register with MacIE.

Related Items: `onkeydown`, `onkeypress`, `onkeyup` event handlers.

nextPage

Value: String. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The `nextPage` property is applicable only if your WinIE5.5+ page uses a TemplatePrinter behavior. Values of this property are one of the following strings: `left`, `right`, or an empty string. For more information about the TemplatePrinter behavior for WinIE5.5+, see

```
http://msdn.microsoft.com/workshop/browser/hosting/printpreview/reference/
behaviors/TemplatePrinter.asp
```

propertyName

Value: String. Read-Only
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `propertyName` property is filled only after an `onpropertychange` event fires.

If a script modifies a property, the `onpropertychange` event handler fires, and the string name of the property is stuffed into the `event.propertyName` property. If the property happens to be a property of the `style` object associated with the element, the `propertyName` is the full property reference, as in `style.backgroundColor`.

Example

See Listing 15-46 in the section about the `onpropertychange` event handler for an example of the values returned by this property.

Related Items: `onpropertychange` event handler (Chapter 15).

repeat

Value: Boolean. Read-Only
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `repeat` property reveals for `onkeydown` events only whether the key is in repeat mode (as determined by the Keyboard control panel settings in the system). With this information, you can prevent the automatic triggering of repeat mode from causing multiple characters from being recognized by the browser. This property can come in handy if users may be physically challenged and may occasionally and accidentally hold down a key too long. The following script fragment in an `onkeydown` event handler for a text box or `textarea` prevents multiple characters from appearing even if the system goes into repeat mode:

```
if (event.repeat) {
   event.returnValue = false;
}
```

By disabling the default action while in repeat mode, no further characters reach the text box until repeat mode goes away (meaning, with the press of another key).

Related Items: `onkeydown` event handler.

(IE) event.returnValue

returnValue

Value: Boolean. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

While IE4+ continues to honor the original way of preventing default action for an event handler (that is, having the last statement of the event handler evaluate to `return false`), the IE4+ event model provides a property that lets the cancellation of default action take place entirely within a function invoked by an event handler. By default, the `returnValue` property of the `event` object is `true`, meaning that the element processes the event after the scripted handler completes its job, just as if the script weren't there. Normal processing, for example, is displaying a typed character, navigating to a link's `href` URL upon being clicked, or submitting a form after the Submit button is clicked.

But you don't always want the default action to occur. For example, consider a text box that is supposed to allow only numbers to be typed in it. The `onkeypress` event handler can invoke a function that inspects each typed character. If the character is not a numeric character, it should not reach the text box for display. The following validation function may be invoked from the `onkeypress` event handler of just such a text box:

```
function checkIt() {
   var charCode = event.keyCode;
   if (charCode < 48 || charCode > 57) {
      alert("Please make sure entries are numerals only.");
      event.returnValue = false;
   }
}
```

By using this event handler, the errant character won't appear in the text box.

Note that this property is not a substitute for the `return` statement of a function. If you need a value to be returned to the invoking statement, you can use a `return` statement in addition to setting the `event.returnValue` property.

Example

You can find several examples of the `returnValue` property at work in Chapter 15 and Chapter 1. Look at Listings 15-30, 15-33, 15-36, 15-37, 15-38, and 15-45. Moreover, many of the other examples in Chapter 15 can substitute the `returnValue` property way of cancelling the default action if the scripts were to be run exclusively on IE4+.

Related Items: `return` statement (Chapter 33).

saveType

Value: String. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The `saveType` property is assigned a value only when an `oncontentsave` event is bound to a WinIE DHTML behavior (`.htc`). For more information about behaviors, see

`http://msdn.microsoft.com/workshop/author/behaviors/overview.asp`

Related Items: `addBehavior()` method.

srcElement

Value: Element object reference. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `srcElement` property is a reference to the HTML element object that is the original target of the event. Because an event may bubble up through the element containment hierarchy and be processed at any level along the way, having a property that points back to the element from which the event originated is comforting. After you have a reference to that element, you can read or write any properties that belong to that element or invoke any of its methods.

Example

As a simplified demonstration of the power of the `srcElement` property, Listing 25-17 has but two event handlers defined for the `body` element, each invoking a single function. The idea is that the `onmousedown` and `onmouseup` events will bubble up from whatever their targets are, and the event handler functions will find out which element is the target and modify the color style of that element.

An extra flair is added to the script in that each function also checks the `className` property of the target element. If the `className` is `bold` — a class name shared by three `span` elements in the paragraph — the stylesheet rule for that class is modified so that all items share the same color. Your scripts can do even more in the way of filtering objects that arrive at the functions to perform special operations on certain objects or groups of objects.

Notice that the scripts don't have to know anything about the objects on the page to address each clicked one individually. That's because the `srcElement` property provides all of the specificity needed for acting on the target element.

Listing 25-17: **Using the srcElement Property**

```
<html>
   <head>
      <title>srcElement Property</title>
      <style type="text/css">
      .bold {font-weight:bold}
      .ital {font-style:italic}
      </style>
      <script type="text/javascript">
      function highlight() {
         var elem = event.srcElement;
         if (elem.className == "bold") {
            document.styleSheets[0].rules[0].style.color = "red";
         } else {
            elem.style.color = "#FFCC00";
         }
      }
      function restore() {
         var elem = event.srcElement;
         if (elem.className == "bold") {
            document.styleSheets[0].rules[0].style.color = "";
         } else {
            elem.style.color = "";
         }
      }
      </script>
   </head>
```

Continued

(IE) event.srcElement

Listing 25-17 *(continued)*

```
   <body onmousedown="highlight()" onmouseup="restore()">
      <h1>srcElement Property</h1>
      <hr />
      <p>One event handler...</p>
      <ul>
         <li>Can</li>
         <li>Cover</li>
         <li>Many</li>
         <li>Objects</li>
      </ul>
      <p>Lorem ipsum dolor sit amet, consectetaur adipisicing elit, <span
         class="bold">sed do</span> eiusmod tempor incididunt <span
         class="ital">ut labore et</span> dolore magna aliqua. Ut enim adminim
         veniam, <span class="bold">quis nostrud exercitation</span> ullamco
         laboris nisi ut aliquip ex ea <span class="bold">commodo
         consequat</span>.</p>
   </body>
</html>
```

Related Items: `fromElement`, `toElement` properties.

srcFilter

Value: String. Read-Only
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

According to Microsoft, the `srcFilter` property should return a string of the name of the filter that was applied to trigger an `onfilterchange` event handler. While the property exists in the `event` object, its value is always `null`, at least through WinIE6.

Related Items: `onfilterchange` event handler; `style.filter` object.

srcUrn

Value: String. Read-Only
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

If an event is fired in a WinIE behavior attached to an element, and the behavior has a URN identifier defined for it, the `srcUrn` property returns the string from the URN identifier. For more information about behaviors, see

`http://msdn.microsoft.com/workshop/author/behaviors/overview.asp`

Related Items: `addBehavior()` method.

toElement

(See `fromElement`)

type

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

You can find out what kind of event fired to create the current `event` object by way of the `type` property. The value is a string version of the event name — just the name of the event without the "on" prefix that is normally associated with event names in IE. This property can be helpful when you designate one event handler function to process different kinds of events. For example, both the `onmousedown` and `onclick` event handlers for an object can invoke one function. Inside the function, a branch is written for whether the type comes in as `mousedown` or `click`, with different processing for each event type. That is not to endorse such event handler function sharing, but for you to be aware of this power should your script constructions find the property helpful.

This property and its values are fully compatible with the NN4 and W3C/NN6+ event models.

Example

Use The Evaluator (Chapter 13) to see values returned by the `type` property. Enter the following object name into the bottom text box and press Enter/Return:

```
event
```

If necessary, scroll the Results box to view the `type` property, which should read `keypress`. Now click the List Properties button. The type changes to `click`. The reason for these types is that the `event` object whose properties are being shown here is the event that triggers the function to show the properties. From the text box, an `onkeypress` event handler triggers that process; from the button, an `onclick` event handler does the job.

Related Items: All event handlers (Chapter 15).

`wheelData`

Value: Integer. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The `wheelData` property returns an integer indicating which direction the mouse wheel was rolled for an `onmousewheel` event. The values returned are typically either 120 or –120, with a positive value indicating that the mouse wheel was rolled toward the screen and a negative value indicating that the wheel was rolled the opposite direction.

NN6+/Moz/Safari event Object

Properties	*Methods*	*Event Handlers*
`altKey`	`initEvent()*`	
`bubbles`	`initKeyEvent()*`	
`button`	`initMouseEvent()*`	
`cancelBubble`	`initMutationEvent()*`	
`cancelable`	`initUIEvent()*`	
`charCode`	`preventDefault()`	
`clientX`	`stopPropagation()`	
`clientY`		
`ctrlKey`		

Continued

(NN6/Moz/Safari) *eventObject*

Properties	*Methods*	*Event Handlers*
currentTarget		
detail		
eventPhase		
isChar		
keyCode		
layerX		
layerY		
metaKey		
originalTarget		
pageX		
pageY		
relatedTarget		
screenX		
screenY		
shiftKey		
target		
timeStamp		
type		
view		

Syntax

Accessing NN6+ event object properties and methods:

eventObject.property | method([*parameters*])

Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

About this object

Although it is based largely on the event object as defined by the W3C DOM Level 2, the NN6+/Moz event object also carries forward several characteristics from the NN4 event object. A few properties are continued primarily for backward compatibility. But because future Mozilla development will likely forego the peculiarities of the NN4 DOM and event models, you should ignore these items (as highlighted below). Wherever possible, look forward and embrace the W3C DOM aspects of the event model. Safari, for example, implements a lot of the W3C DOM event model, but excludes all old NN4 properties.

Although the NN6+ event model provides a bubbling event propagation model just as IE4+, the incompatibility of referencing event objects between the event models is still there. In the W3C DOM (as in NN4), an event object is explicitly passed as a parameter to event handler (or, rather, event listener) functions. But after you have a browser-specific event object assigned to a variable inside a function, a few important properties have the same names

between the IE4+ and W3C DOM event models. If Microsoft adopts more of the W3C DOM event model in future versions of IE, the compatibility situation should improve.

The event object discussed in this section is the instance of an event that is created as the result of a user or system event action. The W3C DOM includes an additional static `Event` object. Many of the properties of the static `Event` object are inherited by the event instances, so the detailed coverage of those shared properties is in this section because it is the event object you'll be scripting for the most part.

In many code fragments in the following detail sections, you will see references that begin with the `evt` reference. This assumes that the statement(s) resides inside a function that has assigned the incoming event object to the `evt` parameter variable:

```
function myFunction(evt) {...}
```

As shown earlier in this chapter, you can equalize W3C DOM and IE4+ event object references when it is practical to do so because the scripts work on identical (or similar) event object properties.

Properties

altKey
ctrlKey
metaKey
shiftKey

Value: Boolean. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

When an event object is created in response to a user or system action, these four properties are set based on whether their corresponding keys were being held down at the time — a Shift-click, for example. If the key was held down, the property is assigned a value of `true`; otherwise the value is `false`. The `metaKey` property corresponds to the Command key on the Macintosh keyboard but does not register for the Windows key on Wintel computers.

Most commonly, you use expressions consisting of this property as `if` construction condition statements. Because these are Boolean values, you can combine multiple properties in a single condition. For example, if you have a branch of a function that is to execute only if the event occurred with both the Shift and Control keys held down, the condition looks as the following:

```
if (evt.shiftKey && evt.ctrlKey) {
   // statements to execute
}
```

Conversely, you can take a more user-friendly approach to provide special processing if the user holds down any one of the four modifier keys:

```
if (evt.shiftKey || evt.ctrlKey || evt.metaKey || evt.altKey) {
   // statements to execute
}
```

The rationale behind this approach is to offer perhaps some shortcut operation for users, but not force them to memorize a specific modifier key combination.

Example

See Listing 25-10, where the values of these properties are used to set the `checked` properties of corresponding checkboxes for a variety of event types.

Related Items: None.

bubbles

Value: Boolean. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

Not every event bubbles. For example, an `onsubmit` event propagates no further than the form object with which the event is associated. Events that do not bubble have their event object's `bubbles` property set to `false`; all others have the property set to `true`. You use this property in the rare circumstance of a single event handler function processing a wide variety of events. You may want to perform special operations only on events that can bubble and handle the others without special treatment. For this branch, you can use the property in an `if` condition statement:

```
if (evt.bubbles) {
   // special processing for bubble-able events
}
```

You do not have to branch, however, just to cancel bubbling. A non-propagating event doesn't mind if you tell it not to propagate.

Related Items: `cancelBubble` property.

button

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `button` property reveals the button that was pressed to activate the mouse event. In the W3C DOM, the left (primary) button returns a value of `0`. If the mouse is a three-button mouse, the middle button returns `1`. The right button (on any multibutton mouse) returns a value of `2`. Note that Safari elects to follow the IE button-numbering convention, where the primary mouse button has a code of `1`.

Mouse buttons other than the primary one are easier to look for in `mousedown` or `mouseup` events, rather than `onclick` events. In the case of a user pressing multiple buttons, only the most recent button is registered.

Exercise caution when scripting the `button` property across browsers. The respective event models define different button values for mouse buttons.

Example

See Listing 25-11, where the `button` property is revealed in the status bar. Try pressing individual mouse buttons on, say, the screen button.

Related Items: None.

cancelable

Value: Boolean. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

If an event is cancelable, then its default action can be prevented from occurring with the help of a script. While most events are cancelable, some are not. The cancelable property lets you inquire about a particular event object to see if its event type is `cancelable`. Values for the property are Booleans. You may want to perform special operations only on events that are cancelable and handle the others without special treatment. For this branch, you can use the property in an `if` condition statement:

```
if (evt.cancelable) {
   // special processing for cancelable events
}
```

You do not have to branch, however, just to prevent an event's default action. A non-cancelable event doesn't mind if you tell it to prevent the default action.

Related Items: `preventDefault()` method.

cancelBubble

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `cancelBubble` property is a rare instance of an IE4+ event property being implemented in NN6+/Moz/Safari even though the property is not defined in the W3C DOM. The property operates the same as in IE4+ in that it determines whether the current event object bubbles up any higher in the element containment hierarchy of the document. By default, this property is `false`, meaning that if the event is supposed to bubble, it will do so automatically.

To prevent event bubbling for the current event, set the property to `true` anywhere within the event handler function. As an alternative, you can cancel bubbling directly in an element's event handler attribute, as in the following:

```
onclick="doButtonClick(this); event.cancelBubble = true"
```

Cancelling event bubbling works only for the current event. The very next event to fire will have bubbling enabled (provided the event bubbles).

If you are trying to migrate your code as much as possible to the W3C DOM, use the `stopPropagation()` method instead of `cancelBubble`. For cross-browser compatibility, however, `cancelBubble` is a safe bet.

Example

See Listing 25-6 to see the `cancelBubble` property in action in an IE environment. Even though that listing has some features that apply to WinIE5.5+, the bubble cancelling demonstration works all the way back to IE4.

Related Items: `stopPropagation()` method.

charCode
keyCode

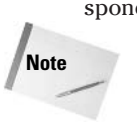
Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The W3C DOM event object model clearly distinguishes between the Unicode character attached to the alphanumeric keys of the keyboard and the code attached to each of the keyboard keys (regardless of its character). To inspect the character of a key, use the

(NN6/Moz/Safari) *eventObject*.charCode

onkeypress event to create the event object, and then look at the event object's charCode property. This is the property that returns 97 for "a" and 65 for "A" because it's concerned with the character associated with the key action. This property's value is zero for onkeydown and onkeyup events.

In contrast, the keyCode property is filled with a non-zero value only from onkeydown and onkeyup events (onkeypress sets the property to zero) when alphanumeric keys are pressed; for most other non-character keys, all three events fill the keyCode property. Through this property you can look for non-character keys, such as arrows, page navigation, and function keys. For the character keys, there is no distinction between uppercase or lowercase: The "A" key on the Latin keyboard returns a value of 65, regardless of the state of the Shift key. At the same time, however, the press of the Shift key fires its own onkeydown and onkeyup events, setting the keyCode value to 16. Other non-character keys — arrows, page navigation, function, and similar — have their own codes as well. This gets very detailed, including special key codes for the numeric keyboard keys that are different from their corresponding numbers along the top row of the alphanumeric keyboard.



Safari 1.0 does not support the charCode property. Instead, the keyCode property returns the ASCII values of alphanumeric keyboard keys — and no non-character keys for any keyboard event.

Be sure to see the extensive section on keyboard events in Chapter 15 for examples of how to apply the keyCode property in applications.

Example

Listing 25-18 provides a play area to view the charCode and keyCode properties for all three keyboard events while you type into a textarea. You can use this later as an authoring tool to grab the precise codes for keyboard keys you may not be familiar with.

Listing 25-18: **Displaying charCode and keyCode Property Values**

```
<html>
   <head>
      <title>charCode and keyCode Properties</title>
      <style type="text/css">
      td {text-align:center}
      </style>
      <script type="text/javascript">
      function showCode(which, evt) {
         document.forms[0].elements[which + "Char"].value = evt.charCode;
         document.forms[0].elements[which + "Key"].value = evt.keyCode;
      }
      function clearEm() {
         for (var i = 1; i < document.forms[0].elements.length; i++) {
            document.forms[0].elements[i].value = "";
         }
      }
      </script>
   </head>
   <body>
      <h1>charCode and keyCode Properties</h1>
      <hr />
```

```
      <form>
         <p><textarea name="scratchpad" cols="40" rows="5" wrap="hard"
            onkeydown="clearEm(); showCode('down', event)"
            onkeyup="showCode('up', event)"
            onkeypress="showCode('press', event)"></textarea></p>
         <table cellpadding="5">
            <tr>
               <th>Event</th>
               <th>event.charCode</th>
               <th>event.keyCode</th>
            </tr>
            <tr>
               <td>onKeyDown:</td>
               <td><input type="text" name="downChar" size="3" /></td>
               <td><input type="text" name="downKey" size="3" /></td>
            </tr>
            <tr>
               <td>onKeyPress:</td>
               <td><input type="text" name="pressChar" size="3" /></td>
               <td><input type="text" name="pressKey" size="3" /></td>
            </tr>
            <tr>
               <td>onKeyUp:</td>
               <td><input type="text" name="upChar" size="3" /></td>
               <td><input type="text" name="upKey" size="3" /></td>
            </tr>
         </table>
      </form>
   </body>
</html>
```

Here are some specific tasks to try with the page in NN6+/Moz1+ to examine key codes (if you are not using a browser set for English and a Latin-based keyboard, your results may vary):

1. Enter a lowercase "a". Notice how the `onkeypress` event handler shows the `charCode` to be 97, which is the Unicode (and ASCII) value for the first of the lowercase letters of the Latin alphabet. But the other two event types record just the key's code: 65.
2. Type an uppercase "A" via the Shift key. If you watch closely, you see that the Shift key, itself, generates the key code 16 for the `onkeydown` and `onkeyup` events. But the character key then shows the value 65 for all three events (until you release the Shift key), because the ASCII value of the uppercase letter happens to match the keyboard key code for that letter.
3. Press and release the Down Arrow key (be sure the cursor still flashes in the `textarea`, because that's where the keyboard events are being monitored). As a non-character key, all three events stuff a value into the `keyCode` property, but zero into `charCode`. The `keyCode` value for this key is 40.
4. Poke around with other non-character keys. Some may produce dialog boxes or menus, but their key codes are recorded nonetheless.

Related Items: `onkeydown`, `onkeypress`, `onkeyup` event handlers.

(NN6/Moz/Safari) *eventObject*.clientX

```
clientX
clientY
layerX
layerY
pageX
pageY
screenX
screenY
```

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The W3C DOM `event` object borrows mouse coordinate properties from both the NN4 and IE4+ event models. If you have worked with event coordinates in these other browsers, you have nothing new to learn for W3C DOM-compatible browsers.

Like the IE4+ `event` object, the W3C DOM `event` object's `clientX` and `clientY` properties are the coordinates within the viewable content region of the window. These values are relative to the window space, not the document. But unlike IE4+, you don't have to calculate the position of the coordinates within the document because another pair of NN/Moz/Safari properties, `pageX` and `pageY`, provide that information automatically. If the page has not scrolled, the values of the client and page coordinates are the same. Because it is usually more important to know an event's coordinates with respect to the document than the window, the `pageX` and `pageY` properties are used most often.

Another NN/Moz/Safari property pair, `layerX` and `layerY`, borrow terminology from the now defunct layer schemes of NN4, but the properties can still be quite valuable nonetheless. These coordinates are measured relative to the positioning context of the element that received the event. For regular, unpositioned elements in the `body` part of a document, that positioning context is the `body` element. Thus, for those elements, the values of the page and layer coordinates will be the same. But if you create a positioned element, the coordinate space is measured from the top-left corner of that space. Thus, if you are using the coordinates to assist in scripted dragging of positioned elements, you can confine your scope to just the positioned element.

One coordinate system missing from the NN6+/Moz repertoire, but present in Safari, is that of the target element itself (comparable to the `offsetX` and `offsetY` properties of IE4+). These values, however, can be calculated in NN/Moz by subtracting from the page coordinate properties the `offsetLeft` and `offsetTop` properties of both the target element and its positioning context. For example, if you want to get the coordinates of a mouse event inside an image, the event handler can calculate those values as follows:

```
var clickOffsetX = evt.pageX - evt.target.offsetLeft -
   document.body.offsetLeft;
var clickOffsetY = evt.pageY - evt.target.offsetTop -
   document.body.offsetTop;
```

The last set of coordinate properties, `screenX` and `screenY`, provide values relative to the entire video display. Of all these properties, only the client and screen coordinates are defined in the W3C DOM Level 2 standard.

Keep in mind that in most W3C DOM–compatible browsers, event targets include text nodes inside elements. Because nodes do not have all the properties of elements (for example, they have no offset properties signifying their location in the document), you may sometimes have to go to the target node's parent node to get an element object whose offset properties provide the necessary page geography. This matters, of course, only if your scripts need concern themselves with mouse events on text.

Example

You can see the effects of the coordinate systems and associated NN6+/Moz properties with the page in Listing 25-19. You can view coordinate values for all four measuring systems, as well as some calculated value. Two clickable objects are provided so that you can see the differences between an object not in any layer and an object residing within a layer (although anything you see is clickable, including text nodes). Figure 25-1 shows the results of a click inside the positioned layer.

One of the calculated fields applies window scrolling values to the client coordinates. But, as you will see, these calculated values are the same as the more convenient page coordinates. The other calculated field shows the coordinates relative to the rectangular space of the target element. Notice in the code that if the `nodeType` of the target indicates a text node, that node's parent node (an element) is used for the calculation.

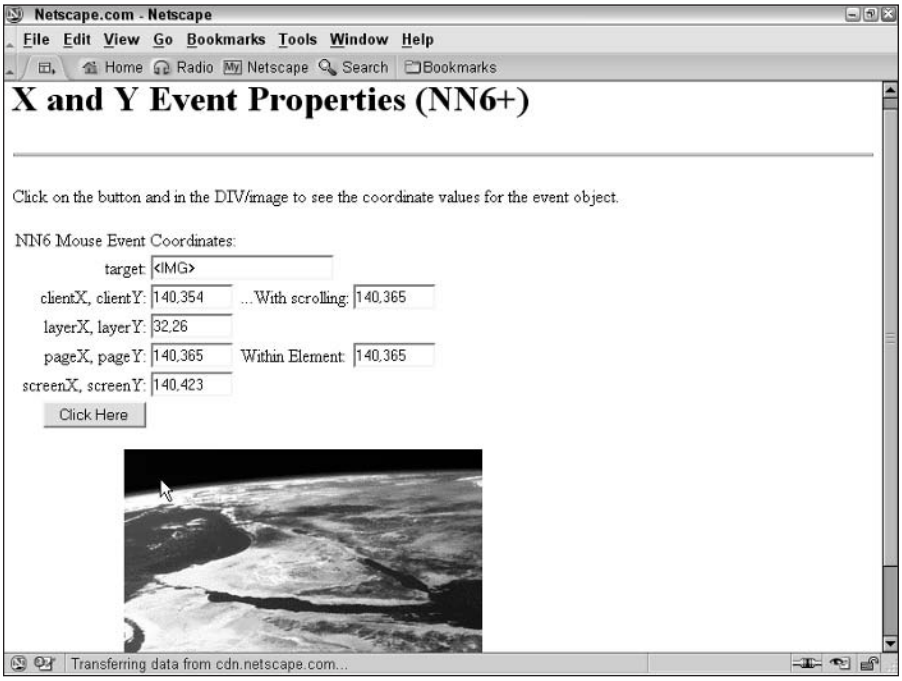


Figure 25-1: NN6+/Moz event coordinates for a click inside a positioned element.

Listing 25-19: NN6+/Moz/Safari Event Coordinate Properties

```
<html>
   <head>
      <title>X and Y Event Properties (NN6+)</title>
      <script type="text/javascript">
      function checkCoords(evt) {
         var form = document.forms["output"];
         var targText, targElem;
         if (evt.target.nodeType == 3) {
            targText = "[textnode] inside <" + evt.target.parentNode.tagName +
               ">";
            targElem = evt.target.parentNode;
         } else {
            targText = "<" + evt.target.tagName + ">";
            targElem = evt.target;
         }
         form.srcElemTag.value = targText;
         form.clientCoords.value = evt.clientX + "," + evt.clientY;
         form.clientScrollCoords.value = (evt.clientX + window.scrollX) +
            "," + (evt.clientY + window.scrollY);
         form.layerCoords.value = evt.layerX + "," + evt.layerY;
         form.pageCoords.value = evt.pageX + "," + evt.pageY;
         form.inElemCoords.value =
            (evt.pageX - targElem.offsetLeft - document.body.offsetLeft) +
            "," + (evt.pageY - targElem.offsetTop - document.body.offsetTop);
         form.screenCoords.value = evt.screenX + "," + evt.screenY;
         return false;
      }
      </script>
   </head>
   <body onmousedown="checkCoords(event)">
      <h1>X and Y Event Properties (NN6+)</h1>
      <hr />
      <p>Click on the button and in the div/image to see the coordinate values
         for the event object.</p>
      <form name="output">
         <table>
            <tr>
               <td colspan="2">NN6 Mouse Event Coordinates:</td>
            </tr>
            <tr>
               <td align="right">target:</td>
               <td colspan="3"><input type="text" name="srcElemTag"
               size="25" /></td>
            </tr>
            <tr>
               <td align="right">clientX, clientY:</td>
               <td><input type="text" name="clientCoords" size="10" /></td>
               <td align="right">...With scrolling:</td>
               <td><input type="text" name="clientScrollCoords"
               size="10" /></td>
            </tr>
            <tr>
               <td align="right">layerX, layerY:</td>
```

```
            <td><input type="text" name="layerCoords" size="10" /></td>
         </tr>
         <tr>
            <td align="right">pageX, pageY:</td>
            <td><input type="text" name="pageCoords" size="10" /></td>
            <td align="right">Within Element:</td>
            <td><input type="text" name="inElemCoords" size="10" /></td>
         </tr>
         <tr>
            <td align="right">screenX, screenY:</td>
            <td><input type="text" name="screenCoords" size="10" /></td>
         </tr>
         <tr>
            <td align="right"><input type="button" value="Click Here" />
            </td>
         </tr>
      </table>
   </form>
   <div id="display" style="position:relative; left:100">
      <img alt="image" src="nile.gif" width="320" height="240" border="0" />
   </div>
</body>
</html>
```

Related Items: `target` property.

currentTarget

Value: Element object reference. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

As an event courses its way through its propagation paths, an event listener may process that event along the way. Though the event knows what the target is, it can also be helpful for the event listener function to know which element's event listener is now processing the event. The `currentTarget` property provides a reference to the element object whose event listener is processing the event. This allows one listener function to potentially process the event from different levels, branching the code to accommodate different element levels that process the event.

A valuable companion piece of information about the event is the `eventPhase` property, which helps your event listener function determine if the event is in capture mode, bubble mode, or is at the target. This property is demonstrated in the next section.

Example

Listing 25-20 shows the power of the `currentTarget` property in revealing the element that is processing an event during event propagation. Similar to the code in Listing 25-7, this example is made simpler because it lets the event object's properties do more of the work to reveal the identity of each element that processes the event. Event listeners assigned for various propagation modes are assigned to a variety of nodes in the document. After you click the button, each listener in the propagation chain fires in sequence. The alert dialog shows which node is processing the event. And, as in Listing 25-7, the `eventPhase` property is used to help display the propagation mode in force at the time the event is processed by each node.

(NN6/Moz/Safari) *eventObject*.currentTarget

Listing 25-20: currentTarget and eventPhase Properties

```
<html>
   <head>
      <title>currentTarget and eventPhase Properties</title>
      <script type="text/javascript">
      function init() {
         // using old syntax to assign bubble-type event handlers
         document.onclick = processEvent;
         document.body.onclick = processEvent;
         // turn on click event capture for document and form
         document.addEventListener("click", processEvent, true);
         document.forms[0].addEventListener("click", processEvent, true);
         // set bubble event listener for form
         document.forms[0].addEventListener("click", processEvent, false);
      }
      function processEvent(evt) {
         var currTargTag, msg;
         if (evt.currentTarget.nodeType == 1) {
            currTargTag = "<" + evt.currentTarget.tagName + ">";
         } else {
            currTargTag = evt.currentTarget.nodeName;
         }
         msg = "Event is now at the " + currTargTag + " level ";
         msg += "(" + getPhase(evt) + ").";
         alert(msg);
      }
      // reveal event phase of current event object
      function getPhase(evt) {
         switch (evt.eventPhase) {
         case 1:
            return "CAPTURING";
            break;
         case 2:
            return "AT TARGET";
            break;
         case 3:
            return "BUBBLING";
            break;
         default:
            return "";
         }
      }
      </script>
   </head>
   <body onload="init()">
      <h1>currentTarget and eventPhase Properties</h1>
      <hr />
      <form>
         <input type="button" value="A Button" name="main1"
         onclick="processEvent(event)" />
      </form>
   </body>
</html>
```

You can also click other places on the page. For example, if you click to the right of the button, you will be clicking the `form` element. Event propagation and processing adjusts accordingly. Similarly, if you click the header text, the only event listeners that see the event are in the document and `body` levels.

Related Items: `eventPhase` property.

`detail`

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `detail` property is included in the W3C DOM specification as an extra property whose purpose can be determined by the browser maker. Mozilla-based browsers increment a numeric value for rapid instances of `click` events on an object; Safari appears to be holding this property in reserve.

Related Items: None.

`eventPhase`

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

An event fires in one of three possible event phases: event capture, at the target, or bubbling. Because the same event listener function may be processing an event in multiple phases, it can inspect the value of the `eventPhase` property of the event object to see in which phase the event was when the function was invoked. Values for this property are integers `1` (capture), `2` (at target), or `3` (bubbling).

Example

See Listing 25-20 earlier in this chapter for an example of how you can use a `switch` construction to branch function processing based on the event phase of the current event object.

Related Items: `currentTarget` property.

`isChar`

Value: Boolean. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

You can find out from each keyboard event whether the key being pressed is a character key by examining the `isChar` property. Most typically, however, you are already filtering for character or non-character keys by virtue of the event handlers used to capture keyboard actions: `onkeypress` for character keys; `onkeydown` or `onkeyup` for non-character keys. Be aware that the `isChar` property returns inconsistent values (even for the same key) in the first release of NN6.

Related Items: `charCode`, `keyCode` properties.

`originalTarget`

Value: Node object reference. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `originalTarget` property provides a reference to the node object that serves as the genuine first target of the event. This information is typically associated with the internal construction of certain elements, which makes it less useful for scripting purposes.

(NN6/Moz/Safari) *eventObject*.originalTarget

Additionally, in many cases the `originalTarget` property holds the same value as the `target` property.

`relatedTarget`

Value: Element object. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `relatedTarget` property allows an element to uncover where the cursor rolled in from or has rolled out to. This property extends the power of the `onmouseover` and `onmouseout` event handlers by expanding their scope to outside the current element (usually to an adjacent element). This one W3C DOM property does the same duty as the `fromElement` and `toElement` properties of the IE4+ `event` object.

When the `onmouseover` event fires on an element, the cursor had to be over some other element just beforehand. The `relatedTarget` property holds a reference to that element. Conversely, when the `onmouseout` event fires, the cursor is already over some other element. The `relatedTarget` property holds a reference to that element.

Example

Listing 25-21 provides an example of how the `relatedTarget` property can reveal the life of the cursor action before and after it rolls into an element. When you roll the cursor to the center box (a table cell), its `onmouseover` event handler displays the text from the table cell from which the cursor arrived (the `nodeValue` of the text node inside the table cell). If the cursor comes in from one of the corners (not easy to do), a different message is displayed.

The two functions that report the results employ a bit of filtering to make sure that they process the event object only if the event occurs on an element and if the `relatedTarget` element is anything other than a nested text node of the central table cell element. Because nodes respond to events in W3C DOM browsers, this extra filtering prevents processing whenever the cursor makes the transition from the central `td` element to its nested text node. Safari 1.0 reports values only on horizontal moves in this example — between cells of the same table row.

Listing 25-21: Using the relatedTarget Property

```
<html>
   <head>
      <title>relatedTarget Properties</title>
      <style type="text/css">
      .direction {background-color:#00FFFF; width:100; height:50;
      text-align:center}
      #main {background-color:#FF6666; text-align:center}
      </style>
      <script type="text/javascript">
      function showArrival(evt) {
         if (evt.target.nodeType == 1) {
            if (evt.relatedTarget != evt.target.firstChild) {
               var direction = (evt.relatedTarget.firstChild) ?
                  evt.relatedTarget.firstChild.nodeValue : "parts unknown";
               window.status = "Arrived from: " + direction;
            }
         }
      }
```

```
      function showDeparture(evt) {
         if (evt.target.nodeType == 1) {
            if (evt.relatedTarget != evt.target.firstChild) {
               var direction = (evt.relatedTarget.firstChild) ?
                  evt.relatedTarget.firstChild.nodeValue : "parts unknown";
               window.status = "Departed to: " + direction;
            }
         }
      }
      </script>
   </head>
   <body>
      <h1>relatedTarget Properties</h1>
      <hr />
      <p>Roll the mouse to the center box and look for arrival information in
         the status bar. Roll the mouse away from the center box and look for
         departure information in the status bar.</p>
      <table cellspacing="0" cellpadding="5">
         <tr>
            <td></td>
            <td class="direction">North</td>
            <td></td>
         </tr>
         <tr>
            <td class="direction">West</td>
            <td id="main" onmouseover="showArrival(event)"
            onmouseout="showDeparture(event)">Roll</td>
            <td class="direction">East</td>
         </tr>
         <tr>
            <td></td>
            <td class="direction">South</td>
            <td></td>
         </tr>
      </table>
   </body>
</html>
```

Related Items: `target` property.

target

Value: Element object reference. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `target` property is a reference to the HTML element object that is the original target of the event. Because an event may trickle down and bubble up through the element containment hierarchy and be processed at any level along the way, having a property that points back to the element from which the event originated is comforting. As soon as you have a reference to that element, you can read or write any properties that belong to that element or invoke any of its methods.

Example

As a simplified demonstration of the power of the `target` property, Listing 25-22 has but two event handlers defined for the `body` element, each invoking a single function. The idea is that

(NN6/Moz/Safari) *eventObject*.target

the `onmousedown` and `onmouseup` events will bubble up from whatever their targets are, and the event handler functions will find out which element is the target and modify the color style of that element.

An extra flair is added to the script in that each function also checks the `className` property of the target element. If the `className` is `bold` — a class name shared by three `span` elements in the paragraph — the stylesheet rule for that class is modified so that all items share the same color. Your scripts can do even more in the way of filtering objects that arrive at the functions to perform special operations on certain objects or groups of objects.

Notice that the scripts don't have to know anything about the objects on the page to address each clicked one individually. That's because the `target` property provides all of the specificity needed for acting on the target element.

Listing 25-22: **Using the target Property**

```
<html>
   <head>
      <title>target Property</title>
      <style type="text/css">
      .bold {font-weight:bold}
      .ital {font-style:italic}
      </style>
      <script type="text/javascript">
      function highlight(evt) {
         var elem = (evt.target.nodeType == 3) ? evt.target.parentNode :
            evt.target;
         if (elem.className == "bold") {
            document.styleSheets[0].cssRules[0].style.color = "red";
         } else {
            elem.style.color = "#FFCC00";
         }
      }
      function restore(evt) {
         var elem = (evt.target.nodeType == 3) ? evt.target.parentNode :
            evt.target;
         if (elem.className == "bold") {
            document.styleSheets[0].cssRules[0].style.color = "black";
         } else {
            elem.style.color = "black";
         }
      }
      </script>
   </head>
   <body onmousedown="highlight(event)" onmouseup="restore(event)">
      <h1>target Property</h1>
      <hr />
      <p>One event handler...</p>
      <ul>
         <li>Can</li>
         <li>Cover</li>
         <li>Many</li>
         <li>Objects</li>
      </ul>
```

```
      <p>Lorem ipsum dolor sit amet, consectetaur adipisicing elit, <span
         class="bold">sed do</span> eiusmod tempor incididunt <span
         class="ital">ut labore et</span> dolore magna aliqua. Ut enim adminim
         veniam, <span class="bold">quis nostrud exercitation</span> ullamco
         laboris nisi ut aliquip ex ea <span class="bold">commodo
         consequat</span>.</p>
   </body>
</html>
```

Related Items: `relatedTarget` property.

timeStamp

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

Each event receives a time stamp in milliseconds, based on the same date epoch as the `Date` object (1 January 1970). Just as with the `Date` object, accuracy is wholly dependent on the accuracy of the system clock of the client computer.

Although the precise time of an event may be of value in only some situations, the time between events can be useful for applications, such as timed exercises or action games. You can preserve the time of the most recent event in a global variable, and compare the time of the current time stamp against the stored value to determine the elapsed time between events.

Example

Listing 25-23 uses the `timeStamp` property to calculate the instantaneous typing speed when you type into a `textarea`. The calculations are pretty raw, and work only on intra-keystroke times without any averaging or smoothing that a more sophisticated typing tutor might perform. Calculated values are rounded to the nearest integer.

Listing 25-23: Using the timeStamp Property

```
<html>
   <head>
      <title>timeStamp Property</title>
      <script type="text/javascript">
      var stamp;
      function calcSpeed(evt) {
         if (stamp) {
            var gross = evt.timeStamp - stamp;
            var wpm = Math.round(6000/gross);
            document.getElementById("wpm").firstChild.nodeValue = wpm +
               " wpm.";
         }
         stamp = evt.timeStamp;
      }
      </script>
   </head>
```

Continued

Listing 25-23 *(continued)*

```
   <body>
      <h1>timeStamp Property</h1>
      <hr />
      <p>Start typing, and watch your instantaneous typing speed below:</p>
      <p><textarea cols="60" rows="10" wrap="hard"
         onkeypress="calcSpeed(event)"></textarea></p>
      <p>Typing Speed: <span id="wpm"> </span></p>
   </body>
</html>
```

Related Items: `Date` object.

type

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

You can find out what kind of event fired to create the current event object by way of the `type` property. The value is a string version of the event name — just the name of the event without the "on" prefix that is normally associated with event listener names in NN6+. This property can be helpful when you designate one event handler function to process different kinds of events. For example, both the `onmousedown` and `onclick` event listeners for an object can invoke one function. Inside the function, a branch is written for whether the type comes in as `mousedown` or `click`, with different processing for each event type. That is not to endorse such event handler function sharing, but be aware of this power should your script constructions find the property helpful.

This property and its values are fully compatible with the NN4 and IE4+ event models.

Caution Keyboard events in Safari 1.0 report their types as `khtml_keydown`, `khtml_keypress`, and `khtml_keyup`, using the prefix referring to the name of the rendering engine on which Safari is built. This is probably to avoid committing to an unfinished W3C DOM Level 3 keyboard event specification.

Related Items: All event handlers (Chapter 15).

view

Value: Window object reference. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The closest that the W3C DOM Level 2 specification comes to acknowledging the browser window is an abstract object called an *abstract view* (AbstractView class). The object's only property is a reference to the document that it contains — the root document node that you've come to know and love. User events always occur within the confines of one of these views, and this is reflected in the `event` object's `view` property. NN6+ returns a reference to the `window` object (which can be a frame) in which the event occurs. This reference allows an event object to be passed to scripts in other frames and those scripts can then gain access to the `document` object of the target element's window.

Related Items: `window` object.

Methods

```
initEvent("eventType", bubblesFlag, cancelableFlag)
initKeyEvent("eventType", bubblesFlag, cancelableFlag,
view, ctrlKeyFlag, altKeyFlag, shiftKeyFlag, metaKeyFlag,
keyCode, charCode)
initMouseEvent("eventType", bubblesFlag, cancelableFlag,
view, detailVal, screenX, screenY, clientX, clientY,
ctrlKeyFlag, altKeyFlag, shiftKeyFlag, metaKeyFlag,
buttonCode, relatedTargetNodeRef)
initMutationEvent("eventType", bubblesFlag,
cancelableFlag, relatedNodeRef, prevValue, newValue,
attrName, attrChangeCode)
initUIEvent("eventType", bubblesFlag, cancelableFlag,
view, detailVal)
```

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The W3C DOM `event` object initialization methods provide a means of initializing a newly created event with a complete set of property values associated with that particular event. The parameters to each of the initialization methods vary according to the type of event being initialized. However, all of the initialization methods share the first three parameters: `eventType`, `bubblesFlag`, and `cancelableFlag`. The `eventType` parameter is a string identifier for the event's type, such as `"mousedown"` or `"keypress"`. The `bubblesFlag` parameter is a Boolean value that specifies whether the event's default propagation behavior is to bubble (`true`) or not (`false`). The `cancelableFlag` parameter is also a Boolean value, and its job is to specify if the event's default action may be prevented with a call to the `preventDefault()` method (`true`) or not (`false`).

A few of the methods also include `view` and `detailVal` parameters, which correspond to the window or frame in which the event occurred and the integer code of detail data associated with the event, respectively. Additional parameters are specified for some of the methods, and are unique to the event being initialized.

You don't have to use the more detailed methods if you need a simple event. For example, if you want a simple `mouseup` event, you can initialize a generic event with `initEvent()`, and dispatch the event to the desired element, without having to fill in all of the coordinate, button, and other parameters of the `initMouseEvent()` method:

```
var evt = document.createEvent("MouseEvents");
evt.initEvent("mouseup", true, true);
document.getElementById("myButton").dispatchEvent(evt);
```

For more details about W3C DOM event types and the values expected for each of the more complex initialization methods, visit `http://www.w3.org/TR/DOM-Level-2-Events/events.html#Events-eventgroupings`.

Related Items: `document.createEvent()` method.

(NN6/Moz/Safari) *eventObject*.preventDefault()

preventDefault()

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

While NN6+ continues to honor the original way of preventing default action for an event handler (that is, having the last statement of the event handler evaluate to `return false`), the W3C DOM event model provides a method that lets the cancellation of default action take place entirely within a function invoked by an event handler. For example, consider a text box that is supposed to allow only numbers to be typed in it. The `onkeypress` event handler can invoke a function that inspects each typed character. If the character is not a numeric character, it does not reach the text box for display. The following validation function may be invoked from the `onkeypress` event handler of just such a text box:

```
function checkIt(evt) {
   var charCode = evt.charCode;
   if (charCode < 48 || charCode > 57) {
      alert("Please make sure entries are numbers only.");
      evt.preventDefault();
   }
}
```

This way, the errant character won't appear in the text box.

Invoking the `preventDefault()` method in NN6+/Moz/Safari is the equivalent of assigning `true` to `event.returnValue` in IE5+.

Related Items: `cancelable` property.

stopPropagation()

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

Use the `stopPropagation()` method to stop events from trickling down or bubbling up further through the element containment hierarchy. A statement in the event listener function that invokes

```
evt.stopPropagation();
```

is all that is needed. As an alternative, you can cancel bubbling directly in an element's event handler attribute, as in the following:

```
onclick="doButtonClick(this); event.stopPropagation()"
```

If you are writing cross-browser scripts, you also have the option of using the `cancelBubble` property, which is compatible with IE4+.

Related Items: `bubbles`, `cancelBubble` properties.

✦ ✦ ✦

CHAPTER 26

Style Sheet and Style Objects

In This Chapter

Managing stylesheets by script

Changing element styles on the fly

Distinguishing among `style`, `styleSheet`, and style objects

Version 4 browsers from Microsoft and Netscape were the first to offer full-scale support for the concept of stylesheets. Stylesheets promote a concept that makes excellent sense in the fast-paced, high-volume content creation environment that is today's World Wide Web: separating content from the rendering details of the content. Textual content may come from any number of electronic sources, but it may need to be dropped into different contexts — just like an online news feed that becomes amalgamated into dozens of Web portal sites, each with its own look and feel. All the content cares about is the text and its meaning; the Web page designer then decides how that content should be rendered on the page.

The concept has other advantages. Consider the large corporate Web site that wants to promote its identity through a distinct style. A family of stylesheets can dictate the font face, font size, the look of emphasized text, and the margin width of all body text. To apply these styles on an element-by-element basis would not only be a tedious page-authoring task, it is fraught with peril. If the style is omitted from the tags of one page, the uniformity of the look is destroyed. Worse yet, if the corporate design changes to use a different font face, the task of changing every style in every tag — even with a highly powered search-and-replace operation — is risky. But if a single external stylesheet file dictates the styles, then the designer need make only one change in that one file to cause the new look to ripple through the entire Web site.

Learning how to create and apply stylesheets is beyond the scope of this book, and this chapter assumes you already are familiar with stylesheet terminology, such as a stylesheet rule and a selector. If these terms are not in your vocabulary, you can find numerous tutorials on the subject both online and in books. Although IE, NN, and other recent browsers adhere fairly closely to W3C standards for stylesheets (called Cascading Style Sheets, or CSS for short), your first learning experience should come from sources that focus on standards, rather than browser-specific features. Microsoft includes some extras in the stylesheet vocabulary that work only on IE4+ for Windows; Netscape 7+ and other Mozilla-based browsers have specially named, preliminary properties that offer future CSS3 capabilities in advance of the final standards. Unless you develop for a single target browser brand and client operating system, learning the common denominator of stylesheet features is the right way to go. Details in this chapter cover all versions, so pay close attention to compatibility listings for each item.

One last compatibility note: Although NN4 implements a fair amount of CSS, it does not expose stylesheets or style rules to the object model. Part of this is linked to the static nature of an NN4 page. Because modifying a style may alter the physical layout of body elements, and because that browser does not reflow the page in response to such changes, altering styles of content that is already loaded is simply not possible. In NN6+/Moz1+, however, the page reflows, and everything relating to styles is exposed to the scriptable object model.

Making Sense of the Object Names

The first task in this chapter is to clarify the seemingly overlapping terminology for the stylesheet-related objects that you will be scripting. Some objects are more abstract than others, but they are all important. The objects in question are

- `style` element object
- `styleSheet` object (a member of the `styleSheets` array)
- rule or `cssRule` object (a member of the `rules` or `cssRules` array)
- `style` object

A `style` element object is the object that represents the `<style>` tag in your document. Most of its properties are inherited from the basic HTML element objects you see detailed in Chapter 15. By and large, you won't be reading or writing stylesheet properties via the `style` element object.

A stylesheet can be embedded in a document via the `<style>` tag or it may be linked in via a `<link>` tag. One property of the `document` object, the `styleSheets` property, returns an array (collection) of all `styleSheet` objects that are currently "visible" to the document, whether or not they are disabled. Included in the collection are stylesheets defined by a `<style>` tag or linked in via a `<link>` tag. Even though the `<style>` tag, for example, contains lines of code that make up the rules for a stylesheet, the `style` element object is not the path to reach the individual rules. The `styleSheet` object is. It is through the `styleSheet` object that you can enable or disable an entire sheet, access individual rules (via the `rules` or `cssRules` property array), and add or delete rules for that stylesheet.

The meat of any stylesheet is the rules that define how elements are to be rendered. At this object level, the terminology forks in IE4 and NN6. The IE4+ object model calls each stylesheet rule a rule object; the W3C DOM Level 2 model (in NN6+/Moz), calls each rule a `cssRule` object. MacIE5 and Safari support both references to the same object. Despite the incompatible object names, the two objects share key property names. Assembling a reference to a rule requires array references. For example, the reference to the first rule of the first `styleSheet` object in the document is as follows for various browsers:

```
var oneRule = document.styleSheets[0].rules[0];       // IE4+, MacIE5, Safari
var oneRule = document.styleSheets[0].cssRules[0];    // MacIE5, NN6+/Moz,
Safari
```

The last object of this quartet of style-related objects is the `style` object. This object is the mother lode, where actual style definitions take place. In earlier chapters, you have seen countless examples of modifying one or more `style` properties of an element. Most typically, this modification is accomplished through the `style` property of the HTML element. For example, you would set the font color of a `span` element whose ID is "hot" as follows:

```
document.all.hot.style.color = "red";                  // IE4+
document.getElementById("hot").style.color = "red";    // IE5+, W3C
```

The style object is also a property of a rule/cssRule object. Thus, if you need to modify the style of elements affected by an existing stylesheet rule, you approach the style object through a different reference path, but the style object is treated just as it is for elements:

```
document.styleSheets[0].rules[0].style.color = "red";       // IE4+, MacIE5,
Safari
document.styleSheets[0].cssRules[0].style.color = "red";    // MacIE5, NN6+/Moz,
Safari
```

Many scripters concern themselves solely with the style object, and at that, a style object associated with a particular element object. Rare are instances that require manipulation of styleSheet objects beyond perhaps enabling and disabling them under script control. Therefore, if you are learning about these objects for the first time, pay closest attention to the style object details rather than to the other related objects.

Imported Stylesheets

Stylesheets embedded in a document via the style element can import additional stylesheets via the @import selector:

```
<style type="text/css">
@import url(externalStyle.css);
p {font-size:16pt}
</style>
```

In this example scenario, the document sees just one styleSheet object. But that object has a stylesheet nested inside — the stylesheet defined by the external file. IE4+ calls one of these imported stylesheets an import object. An import object has all the properties of any styleSheet object, but its parentStyle property is a reference to the styleSheet that "owns" the @import rule. In fact, the @import statement does not even appear among the rules collection of the IE styleSheet object. Therefore, to access the first rule of the imported stylesheet, the reference is as the following:

```
document.styleSheets[0].imports[0].rules[0]
```

The W3C DOM and NN6+/Moz1+ treat import rule objects differently from the IE model. To the W3C DOM, even an at-rule is considered one of the cssRules collection of a styleSheet object. One of the properties of a cssRule object is type, which conveys an integer code value revealing whether the rule is a plain CSS rule or one of several other types, including an import rule. Of course, an imported rule object then has as one of its properties the styleSheet object that, in turn, contains the rules defined in the external stylesheet file. The parent-child relationship exists here, as well, whereby the stylesheet that contains the @import rule is referenced by the imported styleSheet object's parentStyle property (just as in IE4+).

Reading Style Properties

Both the IE4+ and W3C object models exhibit a behavior that at first glance may seem disconcerting. On the one hand, the W3C and good HTML practice encourage defining styles remotely (that is, embedded via <style> or <link> tags) rather than as values assigned to the style attribute of individual element tags throughout the document. This more closely adheres to the notion of separating style from content.

style

On the other hand, object models can be very literal beasts. Strictly speaking, if an element object presents a scriptable property that reflects an attribute for that element's tag, the first time a script tries to read that property, a value will be associated with that property *only* if the attribute is explicitly assigned in the HTML code. But if you assign stylesheet settings via remote stylesheets, the values are not explicitly set in the tag. Therefore, the `style` property of such an element comes up empty, even though the element is under the stylistic control of the remote stylesheet. If all you want to do is assign a new value to a style property, that's not a problem, because your assignment to the element object's `style` property overrides whatever style is assigned to that property in the remote stylesheet (and then that new value is subsequently readable from the `style` property). But if you want to see what the current setting is, the initial value won't be in the element's `style` object.

Microsoft (in IE5+) and the W3C DOM provide competing (and incompatible) solutions so this problem.

IE5+ provides an extra, read-only property — `currentStyle` — that reveals the stylesheet values that are currently being applied to the element, regardless of where the stylesheet definitions are. The `currentStyle` property returns an object that is in the same format and has most of the same properties as the regular `style` property. If your audience runs browsers no earlier than IE5, you should make a habit of reading styles for an element via its `currentStyle` property. If you want a change to a `style` object's property to apply to only one element, use the element's `style` property to set that value; but if the change is to apply to all elements covered by the same remote stylesheet rule, modify the `style` property of the rule object.

The W3C DOM solution is the `getComputedStyle()` method. Although the W3C DOM doesn't (yet) talk about a `window` object, it does describe an object (called the `defaultView`) which Mozilla-based browsers channel through the `window` object. To read the value of a particular style property being applied to an element, you first retrieve a *computed style* value for the element, and then read the desired CSS style property. For example, to read the `font-family` currently applied to an element whose ID is `myP`, use the following sequence:

```
var elem = document.getElementById("myP");
var computedStyle = window.getComputedStyle(elem, "");
var fontFam = computedStyle.getPropertyValue("font-family");
```

Note that you must use the CSS property name (for example, `font-family`) and not the scripted equivalent of that property (for example, `fontFamily`). Unfortunately, Safari 1.0 does not support the `getComputedStyle()` method and does not allow scripts to read stylesheet values set outside of a rendered element's style attribute.

style Element Object

See Chapter 15 for items shared by all HTML elements.

Properties	*Methods*	*Event Handlers*
`media`		
`type`		

Syntax

Accessing `style` element object properties and methods:

```
(IE4+)        document.all.objectID.property | method([parameters])
(IE5+/W3C)    document.getElementById(objectID).property | method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About this object

The `style` element is among the classification of HTML directive elements (see Chapter 36 on the CD-ROM) in that it goes in the `head` portion of a document and does not have any of its own content rendered in the page. But the contents obviously have a great amount of control over the rendering of other elements. Most of the properties, methods, and event handlers that the `style` element inherits from all HTML elements are irrelevant.

One exception is the Boolean `disabled` property. Although there are additional ways to disable a stylesheet (the `disabled` property of the `styleSheet` object), it may be easier to disable or enable a stylesheet by way of the `style` element object. Because you can assign an ID to this element and reference it explicitly, doing so may be more convenient than trying to identify which `styleSheet` object among the document's `styleSheets` collection you intend to enable or disable.

Properties

media

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `media` property represents the `media` attribute of a `style` element. This attribute can define what kind of output device is governed by the stylesheet. The HTML 4.0 specification has lofty goals for this attribute, but at best, computer browsers are limited to the following values: `screen`, `print`, and `all`. Thus, you can design one set of styles to apply when the page is viewed on the computer screen and a different set for when it's printed.

type

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+ , Safari1+

The `type` property represents the `type` attribute of the `style` element. For Cascading Style Sheets, this property is always set to `text/css`. If your scripts assign some other value to this property and the browser does not support that stylesheet type, the stylesheet no longer functions as a Cascading Style Sheet, and any styles it controls revert to their default styles.

styleSheet Object

Properties	*Methods*	*Event Handlers*
cssRules	addImport()	
cssText	addRule()	
disabled	deleteRule()	
href	insertRule()	
id	removeRule()	
imports		
media		
ownerNode		
ownerRule		
owningElement		
pages		
parentStyleSheet		
readOnly		
rules		
title		
type		

Syntax

Accessing `styleSheet` object properties and methods:

```
(IE4+/W3C)    document.styleSheets[index].property | method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About this object

If the `style` element object is the concrete incarnation of a stylesheet, then the `styleSheet` object is its abstract equivalent. A `styleSheet` object exists by virtue of a stylesheet definition being embedded in the current document either by way of the `<style>` tag or linked in from an external file via the `<link>` tag. Each element that introduces a stylesheet into a document creates a separate `styleSheet` object. Access to a `styleSheet` object is via the `document.styleSheets` array. If the document contains no stylesheet definitions, then the array has a length of zero. Styles that are introduced into a document by way of an element's `style` attribute are not considered `styleSheet` objects.

Although both IE4+ and W3C DOM browsers present `styleSheet` objects — and the object represents the same "thing" in both browser families — the set of properties and methods diverges widely between browsers. In many cases, the object provides the same information but through differently named properties in the two families. Interestingly, on some important properties, such as the ones that return the array of style rules and a reference to the HTML element that is responsible for the stylesheet's being in the document, MacIE5 and Safari provide both the Microsoft and W3C terminology. Methods for this object focus on adding rules to and deleting rules from the stylesheet. For the most part, however, your use of the `styleSheet` object will be as a reference gateway to individual rules (via the `rules` or `cssRules` array).

Properties

cssRules

Value: Array of rule objects. Read-Only
Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari1+

The `cssRules` property returns an array of stylesheet rule objects. Strictly speaking, the objects are called `cssRule` objects in the W3C DOM terminology. This property is implemented in MacIE5, but not in the Windows version as of IE6. The list of rule objects is in source code order. The corresponding WinIE4+ property is `rules`.

Example

Use The Evaluator (Chapter 13) to look at the `cssRules` property in W3C DOM browsers or MacIE5. First, view how many rules are in the first `styleSheet` object of the page by entering the following statement into the top text box:

```
document.styleSheets[0].cssRules.length
```

Now use the array with an index value to access one of the rule objects to view the rule object's properties list. Enter the following statement into the bottom text box:

```
document.styleSheets[0].cssRules[1]
```

You use this syntax to modify the style details of an individual rule belonging to the `styleSheet` object.

Related Items: `rules` property, `cssRule`, `rule` objects.

cssText

Value: String. Read/Write
Compatibility: WinIE5+, MacIE5, NN-, Moz-, Safari-

The `cssText` property contains a string of all stylesheet rules contained by the `styleSheet` object. Parsing this text in search of particular strings is not wise because the text returned by this property can have carriage returns and other formatting that is not obvious from the text that is assigned to the rules in the stylesheet. But you can use this property as a way to completely rewrite the rules of a stylesheet in a rather brute-force manner: Assemble a string consisting of all the new rules and assign that string to the `cssText` property. The more formal way of modifying rules (adding and removing them) is perhaps better form, but there is no penalty for using the `cssText` property if your audience is strictly IE5+.

Example

Use The Evaluator (Chapter 13) to replace the style rules in one blast via the `cssText` property. Begin by examining the value returned from the property for the initially disabled stylesheet by entering the following statement into the top text box:

```
document.styleSheets[0].cssText
```

Next, enable the stylesheet so that its rules are applied to the document:

```
document.styleSheets[0].disabled = false
```

Finally, enter the following statement into the top text box to overwrite the stylesheet with entirely new rules.

```
document.styleSheets[0].cssText = "p {color:red}"
```

Reload the page after you are finished to restore the original state.

Related Items: `addRule()`, `deleteRule()`, `insertRule()`, `removeRule()` methods.

disabled

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

While the `disabled` property of the `style` element object works with that element only, the `styleSheet` object's `disabled` property works with a `styleSheet` object that comes into the document by a `link` element as well.

Enabling and disabling stylesheets is one way to swap different appearance styles for a page, allowing the user to select the preferred style. The page can contain multiple stylesheets that control the same selectors, but your script can enable one and disable another to change the overall style. You can even perform this action via the `onload` event handler. For example, if you have separate stylesheets for Windows and Mac browsers, you can put both of them in the document, initially both disabled. An `onload` event handler determines the operating system and enables the stylesheet tailored for that OS. Unless your stylesheets are very extensive, there is little download performance penalty for having both stylesheets in the document.

Example

Use The Evaluator (Chapter 13) to toggle between the enabled and disabled state of the first `styleSheet` object on the page. Enter the following statement into the top text box:

```
document.styleSheets[0].disabled = (!document.styleSheets[0].disabled)
```

The inclusion of the NOT operator (`!`) forces the state to change from `true` to `false` or `false` to `true` with each click of the Evaluate button.

Related Items: `disabled` property of the `style` element object.

href

Value: String. Read/Write (See Text)
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

When a stylesheet is linked into a document via a `link` element, the `href` property of the `styleSheet` object contains a string with the URL to that file. Essentially, the `href` property of the `link` element is passed along to the `styleSheet` object that loads as a result. In WinIE4+ only, this property is read/write, allowing you to dynamically link in an external stylesheet file after the page has loaded. In MacIE and NN6+/Moz, this property is read-only.

Related Items: `link` element object.

id

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `id` property of a `styleSheet` object inherits the `id` property of its containing element (`style` or `link` element). This can get confusing, because it may appear as though two objects in the document have the same ID. The `id` string, however, can be used as an index to the `document.styleSheets` array in IE4+ (for example, `document.styleSheets["winLINK"]`). NN does not provide a comparable identifier associated with a `styleSheet` object.

Related Items: `id` property of all element objects.

imports

Value: Array of `styleSheet` objects. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

A stylesheet can contain one or more `@import` rules to import an external stylesheet file into the document. Each imported `styleSheet` object is treated as an `import` object. The `imports` property is a collection of all imported `styleSheet` objects that belong to the current `styleSheet` object. Imported stylesheets are not added to the `document.styleSheets` collection, so that references to an imported `styleSheet` object must be through the `document.styleSheets[i].imports[i]` array.

An `import` object is, itself, a `styleSheet` object. All properties and methods applicable to a `styleSheet` object also apply to an `import` object. Therefore, if you want to load a different external stylesheet into the page, you can assign the new URL to the imported `styleSheet` object's `href` property:

```
document.styleSheets[0].imports[0].href = "alternate.css";
```

Modifications of this nature work in IE for Windows, but not in MacIE.

Related Items: `styleSheet` object.

media

Value: See text. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Cascading Style Sheets can be defined to apply to specific output media, such as the video display screen, printer, and, in the future, devices such as speech synthesizers or Braille generators. A stylesheet gets this direction from the `media` attribute of a `style` or `link` element. That value is represented in the `media` property of the `styleSheet` object.

In IE4+, the `media` property value is a string with one of three possible values: `screen`, `printer`, `all`. The W3C DOM and NN6+ take this one step further by allowing for potentially multiple values being assigned to the `media` attribute. The NN6+/Moz/Safari value is an array of string media names (returned in an object called a `mediaList`).

Related Items: None.

ownerNode

Value: Node reference. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `ownerNode` property is a reference to the document node in which the `styleSheet` object is defined. For `styleSheet` objects defined inside `style` and `link` elements, the `ownerNode` property is a reference to that element. The corresponding property in IE4+ is `owningElement`. Oddly, MacIE5 has an additional, misnamed property called `owningNode`, whose value equals that of the `owningElement` property.

Example

Use The Evaluator (Chapter 13) with NN6+/Moz1+/Safari to inspect the `ownerNode` of the first `styleSheet` object in the document. Enter the following statement into the top text box:

```
document.styleSheets[0].ownerNode.tagName
```

The returned value is the `style` element tag name.

Related Items: `ownerRule`, `owningElement` property.

ownerRule

Value: cssRule object. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The ownerRule property applies to a styleSheet object that has been imported into a document via the @import rule. The property returns a reference to the @import rule responsible for loading the external stylesheet. There is an interaction between the ownerRule and ownerNode properties in that an imported rule has an ownerRule but its ownerNode property is null; conversely, a regular styleSheet has an ownerNode, but its ownerRule property is null.

Related Items: ownerNode property.

owningElement

Value: Element reference. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The owningElement property is a reference to the element object in which the styleSheet object is defined. For styleSheet objects defined inside style and link elements, the owningElement property is a reference to that element. The corresponding property in NN6+ is ownerNode. Oddly, MacIE5 has an additional, misnamed property called owningNode, whose value equals that of the owningElement property.

Example

Use The Evaluator (Chapter 13) with IE4+ to inspect the owningElement of the first styleSheet object in the document. Enter the following statement into the top text box:

```
document.styleSheets[0].owningElement.tagName
```

The returned value is the style element tag name.

Related Items: ownerNode property.

pages

Value: Array of @page rules. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

An @page style rule defines the dimensions and margins for printed versions of a Web page. The pages property returns a collection of @page rules contained by the current styleSheet object. If no @page rules are defined in the stylesheet, the array has a length of zero.

While an @page rule has the same properties as any rule object, it has one more read-only property, the pseudoClass property, which returns any pseudo-class definitions in the rule. For example, the following @page rules define different rectangle specifications for the left and right printed pages:

```
@page :left {margin-left:4cm; margin-right:3cm;}
@page :right {margin-left:3cm; margin-right:4cm;}
```

Values for the pseudoClass property of these two page rules are :left and :right, respectively.

To the W3C DOM, an @page rule is just another rule object, but one whose type property returns page.

For more information about the paged media specification, see `http://www.w3.org/TR/REC-CSS2/page.html`.

Related Items: None.

parentStyleSheet

Value: `styleSheet` object. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

An imported stylesheet is present thanks to the hosting of a `styleSheet` object created by a `style` or `link` element. That host `styleSheet` object is referenced by the `parentStyleSheet` property. For most `styleSheet` objects (that is, those not imported via the `@import` rule), the `parentStyleSheet` property is `null`. Take note of the distinction between the `parentStyleSheet` property, which points to a `styleSheet` object, and the various properties that refer to the HTML element that "owns" the `styleSheet` object.

Related Items: None.

readOnly

Value: Boolean. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `readOnly` property's name is a bit misleading. Its Boolean value lets your script know whether the current stylesheet was embedded in the document by way of the `style` element or brought in from an external file via the `link` element or `@import` rule. When embedded by a `style` element, the `readOnly` property is `false`; for stylesheets defined outside the page, the property is `true`. But a value of `true` doesn't mean that your scripts cannot modify the style properties. Style properties can still be modified on the fly, but of course the changes will not be reflected in the external file from which the initial settings came.

Related Items: `owningElement` property.

rules

Value: Array of `rule` objects. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari1+

The `rules` property returns an array of all rule objects (other than @ rules) defined in the current stylesheet. The order of rule objects in the array is based on source code order of the rules defined in the `style` element or in the external file.

Use the `rules` array as the primary way to reference an individual rule inside a stylesheet. If you use a `for` loop to iterate through all rules in search of a particular rule, you will most likely be looking for a match of the rule object's `selectorText` property. This assumes, of course, that each selector is unique within the stylesheet. Using unique selectors is good practice, but no restrictions prevent you from reusing a selector name in a stylesheet for additional style information applied to the same selector elements.

The corresponding property name for NN6+/Moz is `cssRules`. MacIE5 and Safari respond to both the `rules` and `cssRules` properties.

Example

Use The Evaluator (Chapter 13) with IE4+ to examine the `rules` property of the first `styleSheet` object in the page. First, find out how many rules are in the first `styleSheet` object by entering the following statement into the top text box:

```
document.styleSheets[0].rules.length
```

Next, examine the properties of one of the rules by entering the following statement into the bottom text box:

```
document.styleSheets[0].rules[1]
```

You now see the all the properties that IE4+ exposes for a rule object.

Related Items: `rule` object; `cssRules` property.

`title`

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

If you assign a value to the `title` attribute of a `style` element or a `link` element that loads a stylesheet, that string value filters down to the `title` property of the `styleSheet` object. You can use the string value as a kind of identifier, but it is not usable as a true identifier that you can use as an index to the `styleSheets` array. In visible HTML elements, the `title` attribute usually sets the text that displays with the tooltip over the element. But for the unseen `style` and `link` elements, the attribute has no impact on the rendered display of the page. Therefore, you can use this attribute and corresponding property to convey any string value you want.

Related Items: `title` property of all HTML elements.

`type`

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `type` property of a `styleSheet` object picks up the `type` attribute of the `style` or `link` element that embeds a stylesheet into the page. Unless you are experimenting with some new types of stylesheet language (assuming it is even supported in the browser), the value of the type property is `text/css`.

Related Items: None.

Methods

`addImport("`*URL*`"[, `*index*`])`

Returns: Integer.
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `addImport()` method lets you add an `@import` rule to a `styleSheet` object. A required first parameter is the URL of the external `.css` file that contains one or more stylesheet rules. If you omit the second parameter, the `@import` rule is appended to the end of rules in the

`styleSheet` object. Or you can specify an integer as the index of the position within the rules collection where the rule should be inserted. The order of rules in a `styleSheet` object can influence the cascading order of overlapping stylesheet rules (that is, multiple rules that apply to the same elements).

The value returned by the method is an integer representing the index position of the new rule within the rules collection of the `styleSheet`. If you need subsequent access to the new rule, you can preserve the value returned by the `addImport()` method and use it as the index to the `rules` collection.

Related Items: `addRule()` method.

`addRule("selector", "styleSpec"[, index])`
`removeRule(index)`

Returns: Integer (for `addRule()`).
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `addRule()` method appends or inserts a stylesheet rule into the current `styleSheet` object. The first two parameters are strings for the two components of every rule: the selector and the style specification. Any valid selector, including multiple, space-delimited selectors, is permitted. For the style specification, the string should contain the semicolon-delimited list of style *`attribute:value`* pairs, but without the curly braces that surround the specification in a regular stylesheet rule.

If you omit the last parameter, the rule is appended to the end of the `rules` collection for the stylesheet. Or, you can specify an integer index value signifying the position within the `rules` collection where the new rule should go. The order of rules in a `styleSheet` object can influence the cascading order of overlapping stylesheet rules (meaning multiple rules that apply to the same elements).

The return value conveys no meaningful information.

To remove a rule from a `styleSheet` object's `rules` collection, invoke the `removeRule()` method. Exercise some care here, because you must have the correct index value for the rule that you want to remove. Your script can use a `for` loop to iterate through the rules collection, looking for a match of the `selectorText` property (assuming that you have unique selectors). The index for the matching rule can then be used as the parameter to `removeRule()`. This method returns no value.

For NN6+, the corresponding methods are called `insertRule()` and `deleteRule()`.

Example

Use The Evaluator (Chapter 13) with IE4+ to add a stylesheet rule to the first `styleSheet` object of the page. First, make sure the stylesheet is enabled by entering the following statement into the top text box:

```
document.styleSheets[0].disabled = false
```

Next, append a style that sets the color of the `textarea` element:

```
document.styleSheets[0].addRule("textarea", "color:red")
```

Enter any valid object (such as `document.body`) into the bottom text box to see how the style has been applied to the `textarea` element on the page.

Now remove the style, using the index of the last item of the `rules` collection as the index:

```
document.styleSheets[0].removeRule(document.styleSheets[0].rules.length - 1)
```

The text in the `textarea` returns to its default color.

Related Items: `deleteRule()`, `insertRule()` methods.

deleteRule(*index*)
insertRule("*rule*", *index*)

Returns: Integer (for `insertRule()`).
Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari1+

The `insertRule()` method appends or inserts a stylesheet rule into the current `styleSheet` object. The first parameter is a string containing the style rule as it would normally appear in a stylesheet, including the selector and curly braces surrounding the semicolon-delimited list of style *attribute*:*value* pairs.

You must supply an index location within the `cssRules` array where the new rule is to be inserted. If you want to append the rule to the end of the list, use the `length` property of the `cssRules` collection for the parameter. The order of rules in a `styleSheet` object can influence the cascading order of overlapping stylesheet rules (meaning multiple rules that apply to the same elements).

The return value is an index for the position of the inserted rule.

Safari 1.0 executes the method without complaint, but the inserted rule does not alter any of the page's rendering.

To remove a rule from a `styleSheet` object's `cssRules` collection, invoke the `deleteRule()` method. Exercise some care here, because you must have the correct index value for the rule that you want to remove. Your script could use a `for` loop to iterate through the `cssRules` collection, looking for a match of the `selectorText` property (assuming that you have unique selectors). The index for the matching rule can then be used as the parameter to `deleteRule()`. This method returns no value.

For IE4+, the corresponding methods are called `addRule()` and `removeRule()`.

Example

Use The Evaluator (Chapter 13) with NN6+/Moz to add a stylesheet rule to the first `styleSheet` object of the page. First, make sure the stylesheet is enabled by entering the following statement into the top text box:

```
document.styleSheets[0].disabled = false
```

Next, append a style that sets the color of the `textarea` element:

```
document.styleSheets[0].insertRule("textarea {color:red}",
document.styleSheets[0].cssRules.length)
```

Enter any valid object (such as `document.body`) into the bottom text box to see how the style has been applied to the `textarea` element on the page.

Now remove the style, using the index of the last item of the rules collection as the index:

```
document.styleSheets[0].deleteRule(document.styleSheets[0].cssRules.length - 1)
```

Related Items: `addRule()`, `removeRule()` methods.

cssRule and rule Objects

Properties	*Methods*	*Event Handlers*
`cssText`		
`parentStyleSheet`		
`readOnly`		
`selectorText`		
`style`		
`type`		

Syntax

Accessing rule or `cssRule` object properties:

```
(IE4+)          document.styleSheets[index].rules[index].property
(MacIE5/W3C)    document.styleSheets[index].cssRules[index].property
```

About these objects

The `rule` and `cssRule` objects are different object model names for the same objects. For IE4+, the object is known as a *rule* (and a collection of them the `rules` collection); for NN6+/Moz/Safari (and MacIE5), the object follows the W3C DOM recommendation, calling the object a *cssRule* (and a collection of them the `cssRules` collection). For the remainder of this section, they will be referred to generically as the rule object.

A rule object has two major components. The first is the selector text, which governs which element(s) are to be influenced by the style rule. The second component is the style definition, with its set of semicolon-delimited *attribute*:*value* pairs. In both the IE4+ and NN6+/W3C object models, the style definition is treated as an object: the `style` object, which has tons of properties representing the style attributes available in the browser. The `style` object that belongs to a rule object is precisely the same `style` object that is associated with every HTML element object. Accessing `style` properties of a stylesheet rule requires a fairly long reference, as in

```
document.styleSheets[0].rules[0].style.color = "red";
```

but the format follows the logic of JavaScript's dot-syntax to the letter.

Properties

cssText

Value: String. Read/Write
Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari1+

The `cssText` property returns the full text of the current `cssRule` object. While the text returned from this property can be parsed to locate particular strings, it is easier and more reliable to access individual style properties and their values via the `style` property of a `cssRule` object. Safari 1.0 returns an empty string.

Related Items: `style` property.

parentStyleSheet

Value: `styleSheet` object. Read-Only
Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari1+

The `parentStyleSheet` property is a reference to the `styleSheet` object that contains the current `cssRule` object. The return value is a reference to a `styleSheet` object, from which scripts can read and write properties related to the entire stylesheet.

Related Items: `parentRule` property.

readOnly

Value: Boolean. Read-Only
Compatibility: WinIE5+, MacIE5, NN-, Moz-, Safari-

The `readOnly` property's name is a bit misleading. Its Boolean value lets your script know whether the current rule's `styleSheet` was embedded in the document by way of the `style` element or brought in from an external file via the `link` element or `@import` rule. When embedded by a `style` element, the `readOnly` property is false; for stylesheets defined outside the page, the property is `true`. But a value of `true` doesn't mean that your scripts cannot modify the style properties. Style properties can still be modified on the fly, but of course the changes are not reflected in the external file from which the initial settings came.

Related Items: `styleSheet.readOnly` property.

selectorText

Value: String. Read-Only
Compatibility: WinIE5+, MacIE5, NN6+, Moz1+, Safari1+

The `selectorText` property returns only the selector portion of a stylesheet rule. The value is a string, and if the selector contains multiple, space-delimited items, the `selectorText` value returns the same space-delimited string. For selectors that are applied to classes (preceded by a period) or ids (preceded by a crosshatch), those leading characters are returned as part of the string as well.

If you want to change the selector for a rule, removing the original rule and adding a new one in its place is better. You can always preserve the `style` property of the original rule and assign the style to the new rule.

Example

Use The Evaluator (Chapter 13) to examine the `selectorText` property of rules in the first `styleSheet` object of the page. Enter each of the following statements in the top text box:

```
document.styleSheets[0].rules[0].selectorText
document.styleSheets[0].rules[1].selectorText
```

Compare these values against the source code view for the `style` element in the page.

Related Items: `style` property.

style

Value: `style` object. Read/Write
Compatibility: WinIE5+, MacIE5, NN6+, Moz1+, Safari1+

The `style` property of a rule (or `cssRule`) is, itself, an object whose properties consist of the CSS style properties supported by the browser. Modifying a property of the `style` object requires a fairly long reference, as in

```
document.styleSheets[0].rules[0].style.color = "red";
```

Any change you make to the rule's `style` properties is reflected in the rendered style of whatever elements are denoted by the rule's selector. If you want to change the style of just one element, access the `style` property of just that element. Style values applied directly to an element override whatever stylesheet style values are associated with the element.

Example

Use The Evaluator (Chapter 13) to modify a `style` property of one of the `styleSheet` rules in the page. The syntax shown here is for IE4+, but you can substitute the `cssRules` reference for the `rules` collection reference in NN6+, MacIE5, and W3C browsers if you like.

Begin by reloading the page and making sure the stylesheet is enabled. Enter the following statement into the top text box:

```
document.styleSheets[0].disabled = false
```

The first rule is for the `myP` element on the page. Change the rule's `font-size` style:

```
document.styleSheets[0].rules[0].style.fontSize = "20pt"
```

Look over the `style` object properties in the discussion of the `style` object later in this chapter and have fun experimenting with different style properties. After you are finished, reload the page to restore the styles to their default states.

Related Items: `style` object.

type

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The W3C DOM defines several classes of stylesheet rules. To make it easier for a script to identify the kind of `cssRule` it is working with, the `type` property returns an integer whose value is associated with one of the known `cssRule` types. While not all of these rule types may be implemented in current browsers, the complete W3C DOM list is as follows:

***ruleObject*.type**

Type	*Description*
0	Unknown type
1	Regular style rule
2	@charset rule
3	@import rule
4	@media rule
5	@font-face rule
6	@page rule

Most of the stylesheet rules you work with are type 1. To learn more about these rule types, consult the W3C specification for CSS at `http://www.w3.org/TR/REC-CSS2`.

Related Items: None.

currentStyle , runtimeStyle , and style Objects

Properties	*Methods*	*Event Handlers*
(See below)		

Syntax

Accessing `currentStyle`, `runtimeStyle`, or `style` object properties:

```
(IE4+/W3C)      elementReference.style.property
(IE4+/W3C)      document.styleSheets[index].style.property
(IE5+)           elementReference.currentStyle.property
(IE5.5)          elementReference.runtimeStyle.property
```

About these objects

All three of these objects — `currentStyle`, `runtimeStyle`, and `style` — return an object that contains dozens of properties related to stylesheet specifications associated either with a `styleSheet` object (for the `style` object only) or any rendered HTML element object. With the browser page reflow facilities of modern browsers, changes made to the properties of the `style` and IE-specific `runtimeStyle` objects are reflected immediately by the rendered content on the page.

The primary object, the `style` object, is accessed as a property of either a `styleSheet` object or an HTML element object. It is vital to remember that style properties of an HTML element are reflected by the `style` object only if the specifications are made via the `style` attribute inside the element's tag. If your coding style requires that stylesheets be applied via `style` or `link` tags, and if your scripts need to access the `style` property values as set by those stylesheets, then you must read the properties of the effective stylesheet through the read-only `currentStyle` property (available in IE5+) or the W3C DOM `window.getComputedStyle()` method (NN6+/Moz).

IE's currentStyle object does not have precisely the same properties as its style object. Missing from the currentStyle object are the properties that contain combination values, such as border or borderBottom. On the other hand, currentStyle provides separate properties for each of the sides of a clipping rectangle (clipTop, clipRight, clipBottom, and clipLeft), which the clip property does not provide.

Microsoft introduced one more flavor of style object — the runtimeStyle object — in IE5.5. This object lets scripts override any style property that is set in a stylesheet or via the "style attribute. In other words, the runtimeStyle object is like a read/write version of currentStyle except that assigning a new value to one of its properties does not modify the stylesheet definition or the value assigned in a style attribute. By and large, however, your scripts will modify the style property of an element to make changes, unless you modify styles by enabling and disabling stylesheets (or changing the className property of an element so that it is under the control of a different selector).

Style properties

If you add up all the style object properties available in browsers starting with IE4 and NN6, you have a list approximately 180 properties long. A sizable percentage are in common among all browsers and are scriptable versions of W3C Cascading Style Sheet properties. The actual CSS property names are frequently script-unfriendly in that multiple-worded properties have hyphens in them, such as font-size. JavaScript identifiers do not allow hyphens, so multiple-worded properties are converted to interCap versions, such as fontSize.

Not all style properties are supported by all browsers that have the style object in their object models. Microsoft, in particular, has added many properties that are sometimes unique to IE and sometimes unique to just IE for Windows. On the Netscape side, you find some properties that appear to be supported by the style object, but the browser doesn't genuinely support the attributes. For example, the CSS specification defines several attributes that enhance the delivery of content that is rendered through a speech synthesizer. Although NN6 and NN7 don't qualify, the Gecko browser engine at their core could be adapted to such a browser. Therefore, if you see a property in the following listings that doesn't make sense to you, test it out in the compatible browsers to verify that it works as you need it. You will also find some properties that are proprietary to Mozilla-based browsers — properties that begin with moz. These properties are preliminary implementations of as yet unreleased CSS Level 3 properties. The moz prefix lets you use these properties today without conflicting with future, sanctioned implementations of the properties (which won't have the moz prefix). When specifying these properties in CSS syntax for your stylesheets, the properties begin with the special prefix -moz-, as in -moz-opacity (and the scripted equivalent, mozOpacity).

Some browsers also expose advanced style object properties to scripters, when, in fact, they are not genuinely supported in the browser. For example, an inspection of the style object for MacIE5 and NN6+ shows a quotes property, which matches the quotes style property in the W3C CSS2 specification. But in truth, the quotes style property cannot be set by script in these browsers. When you see that a property is supported by MacIE5 and NN6+ but none others, testing out the style property (and the stylesheet attribute as well) in The Evaluator is a good idea before attempting to employ the property in your application.

With so many properties associated with an object, it may be difficult to locate the specific property you need for a particular style effect. To help you locate properties, the listings that follow are divided into functional categories, ordered by popularity:

Category	*Description*
Text & Fonts	Font specifications, text rendering, text alignment
Inline Display & Layout	Element flow, alignment, and display
Positioning	Explicit positioning of "layers"
Background	Background images and colors
Borders & Edges	Borders, padding, and margins around elements
Lists	Details for `ul` and `ol` elements
Scroll bars	Scroll bar colors (WinIE5.5+ only)
Tables	Details for `table` elements and components
Printing	Page breaks and alignment for printed pages
Miscellaneous	Odds and ends
Aural	For rendering via speech-synthesis

Property values

All `style` object property values are strings. Moreover, many groups of style properties share the same format for their values. Knowing the formats for the frequently used values is helpful. The purpose of this chapter is not to teach you about stylesheets but to show you how to script them. Therefore, if you see unfamiliar terminology here, consult online or print instructional material about Cascading Style Sheets.

Length

Values for length cover a wide range, but they all define an amount of physical space in the document. Because content can be displayed on a video monitor or printed on a sheet of paper, any kind of length value should include a unit of measure as well as the quantity. One group of units (`px`, `em`, `ex`) are considered *relative* units, because the precise size depends on factors beyond the control of the stylesheet (for example, the pixel density of the display) or units set by elements with more global scope (for example, a `p` element's margin `em` length dependent upon the `body` element's font-size setting). *Absolute* units (`in`, `cm`, `mm`, `pi`, `pt`) are more appropriate for printed output. Length units are referred in script according to the following table:

Unit	*Script Version*	*Example*
pixel	`px`	`14px`
em	`em`	`1.5em`
ex	`ex`	`1.5ex`
inch	`in`	`3.0in`
centimeter	`cm`	`4.0cm`
millimeter	`mm`	`40mm`
pica	`pi`	`72pi`
point	`pt`	`14pt`

A length value can also be represented by a percentage as a string. For example, the `lineHeight` style for a paragraph would be set to `120%` of the font size established for the paragraph by the following statement:

```
document.getElementById("myP").style.lineHeight = "120%";
```

Style inheritance — an important CSS concept — often has significant impact on style properties whose values are lengths.

Color

Values for colors can be one of three types:

- ✦ RGB values (in a few different formats)
- ✦ plain-language versions of the color names
- ✦ plain-language names of system user interface items

RGB values can be expressed as hexadecimal values. The most common way is with a crosshatch character followed by six hex numbers, as in `#ff00ff` (letters can be uppercase or lowercase). A special shortcut is also available to let you specify three numbers with the assumption that they will be expanded to pairs of numbers. For example, a color of `#f0f` is expanded internally to be `#ff00ff`.

An alternative RGB expression is with the `rgb()` prefix and three numbers (from 0 to 255) or percentages corresponding to the red, green, and blue components of the color. Here are a couple of examples:

```
document.styleSheets[0].rules[0].style.color = "rgb(0, 255, 0)";
document.styleSheets[0].rules[0].style.color = "rgb(0%, 100%, 0%)";
```

Browsers also respond to a long list of plain-language color names originally devised by Netscape. You can see the list with sample colors at `http://developer.netscape.com/library/manuals/1998/htmlguide/colortab.htm`. Not all of those colors are necessarily part of what are known as "Web safe" colors. For a demonstration of Web safe colors, visit `http://www.lynda.com/hexh.html`.

The last category of color values references user interface pieces, many of which are determined by the user's control panel for video display. The string values correspond to recognizable UI components (also called system colors), as follows:

```
activeborder        highlight              scrollbar
activecaption       highlighttext          threeddarkshadow
appworkspace        inactiveborder         threedface
background          inactivecaption        threedhighlight
buttonface          inactivecaptiontext    threedlightshadow
buttonhighlight     infobackground         threedshadow
buttonshadow        infotext               window
buttontext          menu                   windowframe
captiontext         menutext               windowtext
graytext
```

Using these color settings may be risky for public sites, because you are at the mercy of the color settings the user has chosen. For a corporate environment where system installations and preferences are strictly controlled, these values could help define a safe color scheme for your pages.

Rectangle sides

Many style properties control the look of sides of rectangles (for example, thickness of a border around a block element). In most cases, the style values can be applied to individual sides or combinations of sides, depending on the number of values supplied to the property. The number of values affects the four sides of the rectangle according to the following matrix:

Number of Values	*Impact*
1	All four sides set to the one value
2	Top and bottom sides set to first value; left and right sides set to second value
3	Top side set to first value; left and right sides set to second value; bottom side set to third value
4	Top, right, bottom, and left sides set to individual values in that order

For example, to set the border color of an element so that all sides are red, the syntax is

```
elementRef.style.borderColor = "red";
```

To set the top and bottom to red but the left and right to green, the syntax is

```
elementRef.style.borderColor = "red green";
```

Properties that accept these multiple values cover a wide range of styles. Values may be colors, lengths, or selections from a fixed list of possible values.

Combination values

Another category of style values includes properties that act as shortcuts for several related properties. For example, the `border` property encompasses the `borderWidth`, `borderStyle`, and `borderColor` properties. This is possible because very different classes of values represent the three component properties: `borderWidth` is a length; `borderStyle` is based on a fixed list of values; and `borderColor` is a color value. Therefore, you can specify one or more of these property values (in any order), and the browser knows how to apply the values to the detailed subproperty. Only one value is permitted for any one of these subproperties, which means that if the property is one of the four-sided styles described in the previous section, the value is applied to all four sides equally.

For example, setting the `border` property to a single value, as in

```
elementRef.style.border = "blue";
```

is the same as setting

```
elementRef.style.borderColor = "blue";
```

But if you set multiple items, as in

```
elementRef.style.border = "groove blue 3px";
```

then you have set the equivalent of the following three statements:

```
elementRef.style.borderStyle = "groove"
elementRef.style.borderColor = "blue";
elementRef.style.borderWidth = "3px";
```

In the property descriptions that follow, these combination values are denoted by their scripted property names and the OR (||) operator, as in

```
border = "borderStyle || borderColor || borderWidth";
```

URLs

Unlike other property values containing URLs, a `style` property requires a slightly different format. This format includes the `url()` prefix, with the actual URL (relative or absolute) located inside the parentheses. The URL itself is not quoted, but the entire property value is, as in

```
elementRef.style.backgroundImage = "url(chainlink.jpg)";
```

URLs should not have any spaces in them, but if they do, use the URL-encoded version for the file specification: convert spaces to `%20`. This format distinguishes a URL value from some other string value for shortcut properties.

Text and font properties

`color`

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Foreground color of an element, primarily used to assign color to text. May also affect edges and highlights of other elements in some browsers.
Value: Color specification.
Example: `elementRef.style.color = "rgb(#22FF00)";`

`font`

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Up to six font-related style properties.
Value: Combination values: `fontStyle || fontVariant || fontWeight || fontSize || lineHeight || fontFamily`. See individual properties for their value formats.
Example: `elementRef.style.font = "bold sans-serif 16px";`

`fontFamily`

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Font family to be applied to an element in order of priority.
Value: Comma-delimited list of font families to be applied to element, starting with the most preferred font family name. You can also use one of several generic family names that rely on the browser to choose the optimal font to match the class: `serif | sans-serif | cursive | fantasy | monospace`. Not all browsers support all constants, but `serif`, `sans-serif`, and `monospace` are commonly implemented.
Example: `elementRef.style.fontFamily = "Bauhaus 93, Arial, monospace";`

fontSize

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Size of the characters of the current font family.
Value: Lengths (generally px or pt values); relative size constants: `larger | smaller`; absolute size constants: `xx-small | x-small | small | medium | large | x-large | xx-large`
Examples: `elementRef.style.fontSize = "16px"; elementRef.style.fontSize = "small";`

fontSizeAdjust

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari-
Controls: Aspect value of a secondary font family so that it maintains a similar character height as the primary font family.
Value: Number (including floating-point value) or `none`
Example: `elementRef.style.fontSizeAdjust = "1.05";`

fontStretch

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari-
Controls: Rendered width of a font's characters.
Value: Constant `ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded` or `wider | narrower | inherit | normal`
Example: `elementRef.style.fontStretch = "expanded";`

fontStyle

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Italic style of characters.
Value: Constant `normal | italic | oblique | inherit`
Example: `elementRef.style.fontStyle = "italic";`

fontVariant

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari-
Controls: Rendering characters as small caps.
Value: Constant `normal | small-caps | inherit`
Example: `elementRef.style.fontVariant = "small-caps";`

fontWeight

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Rendering characters in bold or light weights. Fonts that support numbered gradations can be controlled by those numbers. Normal = 400; Bold = 700.
Value: Constant `bold | bolder | lighter | normal | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | inherit`
Example: `elementRef.style.fontWeight = "bold";`

letterSpacing

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Spacing between characters. Used to override a font family's own characteristics.
Value: Length (usually em units, relative to current font size); Constant `normal | inherit`
Example: `elementRef.style.letterSpacing = "1.2em";`

lineBreak

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-
Controls: Line-break rules for Japanese text content.
Value: Constant `normal | strict`
Example: `elementRef.style.lineBreak = "strict";`

lineHeight

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Height of the rectangular space that holds a line of text characters.
Value: Length (usually em units, relative to current font size); number (a multiplier on the inherited line height); percentage (relative to inherited line height); constant `normal | inherit`
Example: `elementRef.style.lineHeight = "1.1";`

quotes

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari-
Controls: Characters to be used for quotation marks.
Value: Space-delimited pairs of open and close quotation symbols; Constant `none | inherit`
Example: `elementRef.style.quotes = "« »";`

rubyAlign

Compatibility: WinIE5+, MacIE5, NN-, Moz-, Safari-
Controls: Alignment of ruby text within a `ruby` element.
Value: Constant `auto | left | center | right | distribute-letter | distribute-space | line-edge`
Example: `RUBYelementRef.style.rubyAlign = "distribute=letter";`

rubyOverhang

Compatibility: WinIE5+, MacIE5, NN-, Moz-, Safari-
Controls: Overhang of ruby text within a `ruby` element.
Value: Constant `auto | whitespace | none`
Example: `RUBYelementRef.style.rubyOverhang = "whitespace";`

rubyPosition

Compatibility: WinIE5+, MacIE5, NN-, Moz-, Safari-
Controls: Placement of ruby text with respect to the `ruby` element's base text.
Value: Constant `above | inline`
Example: `RUBYelementRef.style.rubyPosition = "inline";`

textAlign

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Horizontal alignment of text with respect to its containing element.
Value: Constant `center | justify | left | right`
Example: `elementRef.style.textAlign = "center";`

textAlignLast

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-
Controls: Horizontal alignment of last line of text in a paragraph.
Value: Constant `auto | center | justify | left | right`
Example: `elementRef.style.textAlignLast = "justify";`

textAutospace

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-
Controls: Extra spacing between ideographic and non-ideographic text.
Value: Constant `none | ideograph-alpha | ideograph-numeric | ideograph-parenthesis | ideograph-space`
Example: `elementRef.style.textAutospace = "ideograph=alpha";`

textDecoration

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Display of underline, overline, or line-through with text.
Value: Constant `none | blink | line-through | overline | underline`
Example: `elementRef.style.textDecoration = "underline";`

textDecorationBlink
textDecorationLineThrough
textDecorationNone
textDecorationOverline
textDecorationUnderline

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-
Controls: Individual text decoration characteristics for text, allowing for multiple decorations to be applied to the same text.
Value: Boolean (not strings) `true | false`
Example: `elementRef.style.textDecorationUnderline = true;`

textIndent

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Amount of indentation for the first line of a block text element (for example, `p`).
Value: Length (negative values for outdenting); percentage (relative to inherited value)
Example: `elementRef.style.textIndent = "2.5em";`

textJustify

Compatibility: WinIE5+, MacIE5, NN-, Moz-, Safari-
Controls: Additional detailed specifications for an element whose `textAlign` property is set to `justify`.
Value: Constant `auto | distribute | distribute-all-lines | distribute-center-last | inter-cluster | inter-ideograph | inter-word | kashida | newspaper`
Example: `elementRef.style.textJustify = "distribute";`

textJustifyTrim

Compatibility: WinIE5+, MacIE5, NN-, Moz-, Safari-
Reserved for future use.

textKashidaSpace

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-
Controls: Ratio of kashida expansion to white space expansion for Arabic writing systems.
Value: Percentage
Example: `elementRef.style.textKashidaSpace = "90%";`

TextOverflow

Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-
Controls: Whether an ellipsis (. . .) is displayed at the end of a line of overflowed text to indicate that more text is available.
Value: Constant `clip | ellipsis`

textShadow

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari-
Controls: Shadow rendering around text characters. Note: The `style` attribute for this property is not implemented in MacIE5 or NN6/7, but the property is listed as valid for a `style` object.
Value: Each shadow specification consists of an optional color and three space-delimited length values (horizontal shadow offset, vertical shadow offset, blur radius length). Multiple shadow specifications are comma-delimited.

textTransform

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Case rendering of the text (meaning without altering the case of the original text).
Value: Constant `none | capitalize | lowercase | uppercase`
Example: `elementRef.style.textTransform = "uppercase";`

textUnderlinePosition

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-
Controls: Whether an underline text decoration is displayed above or below the text. Seems redundant with `textDecorationUnderline` and `textDecorationOverline`.
Value: Constant `above | below`
Example: `elementRef.style.textUnderlinePosition = "above";`

unicodeBidi

Compatibility: WinIE5+, MacIE5, NN6+, Moz1+, Safari1+
Controls: Within bidirectional text (for example, English and Arabic), to what extent an alternate direction text block is embedded within the outer element.
Value: Constant `normal | embed | bidi-override`
Example: `elementRef.style.unicodeBidi = "embed";`

whiteSpace

Compatibility: WinIE5.5+, MacIE5, NN6+, Moz1+, Safari1+
Controls: Treatment of white space characters within an element's source code.
Value: Constant `normal | nowrap | pre`
Example: `elementRef.style.whiteSpace = "nowrap";`

wordBreak

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-
Controls: Word breaking characteristics, primarily for Asian-language text or text containing a mixture of Asian and Latin characters.
Value: Constant `normal | break-all | keep-all`
Example: `elementRef.style.wordBreak = "break-all";`

wordSpacing

Compatibility: WinIE6+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Spacing between words.
Value: Length (usually in em units); Constant `normal`
Example: `elementRef.style.wordSpacing = "1em";`

wordWrap

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-
Controls: Word wrapping characteristics of text in a block element, explicitly sized inline element, or positioned element.
Value: Constant `normal | break-word`
Example: `elementRef.style.wordWrap = "break-word";`

writingMode

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-
Controls: Direction of content flow (left-to-right/top-to-bottom or top-to-bottom/right-to-left, as in some Asian languages).
Value: Constant `lr-tb | tb-rl`
Example: `elementRef.style.writingMode = "tb-rl";`

Inline display and layout properties

clear

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Layout orientation of an element with respect to a neighboring floating element.
Value: Constant `both | left | none | right`
Example: `elementRef.style.clear = "right";`

clip

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: The clipping rectangle of an element (that is, the position of the rectangle through which the user sees an element's content).
Value: `rect(topLength, rightLength, bottomLength, leftLength) | auto`
Example: `elementRef.style.clip = "rect(10px, 300px, 200px, 0px)";`

clipBottom
clipLeft
clipRight
clipTop

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-
Controls: Individual edges of the clipping rectangle of an element. These properties are read-only properties of the `currentStyle` object.
Value: Length `| auto`
Example: `var leftEdge = elementRef.currentStyle.clipLeft;`

content

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari-
Controls: The content rendered in addition to the element, usually to be applied with a `:before` or `:after` pseudo-class. This feature will become more useful when CSS counters are implemented in browsers. They'll provide automatic section or paragraph numbering. While the CSS equivalent is implemented in NN7/Moz1/Safari, changes to the scripted property are not rendered.
Value: See `http://www.w3.org/TR/REC-CSS2/generate.html#propdef-content`.

counterIncrement

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari-
Controls: The jumps in counter values to be displayed via the `content` style property. Note: The CSS property is not implemented in MacIE5 or NN6/7, but the property is listed as valid for a `style` object.
Value: One or more pairs of counter identifier and integers.

counterReset

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari-
Controls: Resets a named counter for content to be displayed via the `content` style property. Note: The CSS property is not implemented in MacIE5 or NN6/7, but the property is listed as valid for a `style` object.
Value: One or more pairs of counter identifier and integers.

cssFloat

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari1+
Controls: Horizontal alignment of an element that allows other content to wrap around the element (usually text wrapping around an image). Corresponds to the CSS `float` style attribute. See also the `floatStyle` property, below. Floating (non-positioned) elements follow a long sequence of rules for their behavior, detailed at `http://www.w3.org/TR/REC-CSS2/visuren.html#propdef-float`.
Value: Constant `left | right | none`
Example: `elementRef.style.cssFloat = "right";`

cursor

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari-
Controls: The icon used for the cursor on the screen from a library of system-generated cursors. The CSS2 specification defines syntax for downloadable cursors, but this feature is not implemented as of IE5.5 or NN6/7. You can change this style property only if a `:hover` pseudo-class is initially defined for the element.
Value: Constant `auto | crosshair | default | e-resize | help | move | n-resize | ne-resize | nw-resize | pointer | s-resize | se-resize | sw-resize | text | w-resize | wait`. New values for IE6 are: `all-scroll | col-resize | no-drop | not-allowed | progress | row-resize | url | vertical-text`. Mozilla-based browsers include: `alias | cell | context-menu | copy | count-down | count-up | count-up-down | grab | grabbing | spinning`.
Example: `elementRef.style.cursor = "hand";`

direction

Compatibility: WinIE5+, MacIE5, NN6+, Moz1+, Safari1+
Controls: Layout direction (left-to-right or right-to-left) of inline text (same as `dir` attribute of an element).
Value: Constant `ltr | rtl`
Example: `elementRef.style.direction = "rtl";`

display

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Whether an element is displayed on the page and in which display mode. Content surrounding an undisplayed element cinches up to occupy the undisplayed element's space—as if the element didn't exist for rendering purposes (see the `visibility` property for a different approach). Commonly used to hide or show segments of a graphical tree structure.

Also used to direct the browser to display an element as inline or block-level element. Some special-purpose values are associated with specific element types (for example, lists, table cells, and so on).
Value: Constant `block | compact | inline | inline-table | list-item | none | run-in | table | table-caption | table-cell | table-column-group | table-footer-group | table-header-group | table-row | table-row-group`
Example: `elementRef.style.display = "none";// removes element from page`

filter

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-
Controls: Rendering effects on static content and on transitions between hiding and showing elements. Microsoft made a massive overhaul of the `filter` stylesheet syntax in WinIE5.5 (using the `DXImageTransform` ActiveX control). Scripting transitions require several steps to load the transition and actions before playing the transition. Use `style.filter` to read or write the entire filter specification string; use the `elem.styles[i]` object to access individual filter properties. See the discussion of the `filter` object later in this chapter.
Value: Filter specification as string.
Example: `var filterSpec = elementRef.style.filter = "alpha(opacity=50) flipH()";`

layoutGrid

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-
Controls: Page grid properties (primarily for Asian-language pages).
Value: Combination values: `layoutGridMode || layoutGridType || layoutGridLine || layoutGridChar`. See individual properties for their value formats.
Example: `elementRef.style.layoutGrid = "2em fixed";`

layoutGridChar

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-
Controls: Size of the character grid (Asian languages).
Value: Length; Percentage; Constant `none | auto`
Example: `elementRef.style.layoutGridChar = "2em";`

layoutGridLine

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-
Controls: Line height of the grid (Asian languages).
Value: Value: Length; Percentage; Constant `none | auto`
Example: `elementRef.style.layoutGridLine = "110%";`

layoutGridMode

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-
Controls: One- or two-dimensional grid (Asian languages).
Value: Constant `both | none | line | char`
Example: `elementRef.style.layoutGridMode = "both";`

***elementRef*.style.layoutGridType**

layoutGridType

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-
Controls: Type of grid for text content (Asian languages).
Value: Constant `loose | strict | fixed`
Example: `elementRef.style.layoutGridType = "strict";`

markerOffset

Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-
Controls: Distance between the edges of a marker box (content whose display is of a marker type) and a block-level element's box. Note: The CSS property is not implemented in MacIE5 or NN6/7, but the property is listed as valid for a `style` object.
Value: Length; Constant `auto`
Example: `elementRef.style.markerOffset = "2em";`

marks

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari-
Controls: Rendering of crop marks and the like on the printed page. Note: The CSS property is not implemented in MacIE5 or NN6/7, but the property is listed as valid for a `style` object.
Value: Constant `crop || cross | none`
Example: `elementRef.style.marks = "crop";`

maxHeight
maxWidth
minHeight
minWidth

Compatibility: WinIE (see text), MacIE-, NN6+, Moz1+, Safari-
Controls: Maximum or minimum height or width of an element. IE6 supports `minHeight` only.
Value: Length; Percentage; Constant (for max properties only) `none`
Example: `elementRef.style.maxWidth = "300px";`

overflow

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: The rendering of a block-level element's content when its native rectangle exceeds that of its next outermost rectangular space. A `hidden` overflow clips the block-level content; a `scrolled` overflow forces the outermost rectangle to display scroll bars so that users can scroll around the block-level element's content; a `visible` overflow causes the block-level element to extend beyond the outermost container's rectangle (indeed, "overflowing" the container).
Value: Constant `auto | hidden | scroll | visible`
Example: `elementRef.style.overflow = "scroll";`

overflowX
overflowY

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-
Controls: The rendering of a block-level element's content when its native rectangle exceeds the width (`overflowX`) or height (`overflowY`) of its next outermost rectangular space. A hidden overflow clips the block-level content; a scrolled overflow forces the outermost rectangle to display scroll bars so that users can scroll around the block-level element's content; a visible overflow causes the block-level element to extend beyond the outermost container's rectangle (indeed, "overflowing" the container).
Value: Constant `auto | hidden | scroll | visible`
Example: `elementRef.style.overflowX = "scroll";`

styleFloat

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-
Controls: Horizontal alignment of an element that allows other content to wrap around the element (usually text wrapping around an image). Corresponds to the CSS `float` style attribute. See also the `cssFloat` property, above. Floating (non-positioned) elements follow a long sequence of rules for their behavior, detailed at `http://www.w3.org/TR/REC-CSS2/visuren.html#propdef-float`.
Value: Constant `left | right | none`
Example: `elementRef.style.styleFloat = "right";`

verticalAlign

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari-
Controls: How inline and table cell content aligns vertically with surrounding content. Not all constant values are supported by all browsers.
Value: Constant `baseline | bottom | middle | sub | super | text-bottom | text-top | top`; Length; Percentage.
Example: `elementRef.style.verticalAlign = "baseline";`

visibility

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Whether an element is displayed on the page. The element's space is preserved as empty space when the element is hidden. To cinch up surrounding content, see the `display` property. This property is used frequently for hiding and showing positioned element under script control.
Value: Constant `collapse | hidden | visible`
Example: `elementRef.style.visibility = "hidden";`

width

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Horizontal dimension of a block-level element. Earlier browsers exhibit unexpected behavior when nesting elements that have their `width` style properties set.
Value: Length; Percentage; Constant `auto`
Example: `elementRef.style.width = "200px";`

zoom

Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-
Controls: Magnification factor of a rendered element.
Value: Constant `normal`; Percentage (where `100%` is normal); floating-point number (scale multiplier, where 1.0 is normal)
Example: `elementRef.style.zoom = ".9";`

Positioning properties

(See Chapter 39 on the CD-ROM for coding examples of positioned elements and their `style` properties.)

bottom
right

Compatibility: WinIE5+, MacIE5, NN6+, Moz1+, Safari1+
Controls: The offset measure of a positioned element from its containing rectangle's bottom and right edges, respectively. In practice, you should adjust the size of a positioned element via the style's `height` and `width` properties.
Value: Length; Percentage; Constant `auto`
Example: `elementRef.style.bottom = "20px";`

left
top

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: The offset measure of a positioned element from its containing rectangle's left and top edges, respectively. In practice, use these properties to position an element under script control. To position an absolute-positioned element atop an inline element, calculate the position of the inline element via the `offsetTop` and `offsetLeft` properties with some browser-specific adjustments, as shown in Chapter 39 on the CD-ROM.
Value: Length; Percentage; Constant `auto`
Example: `elementRef.style.top = "250px";`

height
width

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Height and width of a block-level element's box. Used most commonly to adjust the dimensions of a positioned element (Chapter 39 on the CD-ROM).
Value: Length; Percentage; Constant `auto`
Example: `elementRef.style.height = "300px";`

pixelBottom
pixelHeight
pixelLeft
pixelRight
pixelTop
pixelWidth

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari1+
Controls: Integer pixel values for (primarily positioned) elements. Because the non-pixel versions of these properties return strings that also contain the unit measure (for example, `30px`), these properties let you work exclusively in integers for pixel units. The same can be done cross-platform by using `parseInt()` on the non-pixel versions of these properties. The `pixelBottom` and `pixelRight` properties are not in MacIE4.
Value: Integer
Example: `elementRef.style.pixelTop = elementRef.style.pixelTop + 20;`

posBottom
posHeight
posLeft
posRight
posTop
posWidth

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-
Controls: Numeric values for (primarily positioned) elements in whatever unit was specified by the corresponding `style` attribute. Because the non-pos versions of these properties return strings that also contain the unit measure (for example, `1.2em`), these properties let you work exclusively in numbers in the same units as the style was originally defined. The same can be done cross-platform by using `parseFloat()` on the non-pixel versions of these properties.
Value: Integer
Example: `elementRef.style.posTop = elementRef.style.posTop + 0.5;`

position

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: The type of positioning to be applied to the element. An element that is not explicitly positioned is said to be *static*. A relative-positioned element appears in its normal page flow location but can be explicitly positioned relative to that location. An absolute-positioned element must have its `top` and `left` style attributes set to give the element a set of coordinates for its location. MacIE5 and NN6+/Moz/Safari also allow for a fixed positioned element, which remains at its designated position in the browser window, even if the page scrolls (for example, for a watermark effect). You cannot use scripts to change between positioned and non-positioned style settings. See Chapter 39 on the CD-ROM for more information on positioned elements.
Value: Constant `absolute | fixed | relative | static`
Example: `elementRef.style.position = "absolute";`

zIndex

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Front-to-back layering of positioned elements. Multiple items with the same `zIndex` value are layered in source code order (earliest item at the bottom). The higher the value, the closer to the user's eye the element is.
Value: Integer number; Constant `auto`
Example: `elementRef.style.zIndex = "3";`

Background properties

background

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Up to five background style properties for an element.
Value: Combination values: `backgroundAttachment || backgroundColor || backgroundImage || backgroundPosition || backgroundRepeat`
Example: `elementRef.style.background = "scroll url(bricks.jpg) repeat-x";`

backgroundAttachment

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari-
Controls: Whether the background image remains fixed or scrolls with the content. Default is `scroll`.
Value: Constant `fixed | scroll`
Example: `elementRef.style.backgroundAttachment = "fixed";`

backgroundColor

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Solid, opaque color for the background, or completely transparent. If you assign a background image, the color is layered behind the image so that any transparent spots of the image show the background color.
Value: Color value; Constant `transparent`
Example: `elementRef.style.backgroundColor = "salmon";`

backgroundImage

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: The URL (if any) of an image to be used for the background for the element.
Value: URL value; Constant `none`
Example: `elementRef.style.backgroundImage = "url(bricks.jpg)";`

backgroundPosition

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: The left-top location of the background image. Any offset from the left-top corner (default value "0% 0%") allows background color to show through along left and top edges of the element.
Value: Length values; Percentages; Constant `left | center | right || top | center | bottom`. While single values are accepted, their behavior may not be as expected. Providing space-delimited pairs of values is more reliable.
Example: `elementRef.style.backgroundPosition = "left top";`

backgroundPositionX
backgroundPositionY

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-
Controls: The left (backgroundPositionX) and top (backgroundPositionY) locations of the background image. Any offset from the left-top corner (default value "0%") allows background color to show through along left and top edges of the element.
Value: Length value; Percentage; Constant left | center | right (for backgroundPositionX); Constant top | center | bottom (for backgroundPositionY).
Example:
```
elementRef.style.backgroundPositionX = "5px";
```

backgroundRepeat

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Image repetition characteristics of a background image. You can force the image to repeat along a single axis, if you want.
Value: Constant repeat | repeat-x | repeat-y | no-repeat
Example:
```
elementRef.style.backgroundRepeat = "repeat-y";
```

Border and edge properties

border

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Up to three border characteristics (color, style, and width) for all four edges of an element.
Value: Combination values borderColor || borderStyle || borderWidth
Example:
```
elementRef.style.border = "green groove 2px";
```

borderBottom
borderLeft
borderRight
borderTop

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Up to three border characteristics (color, style, and width) for a single edge of an element.
Value: Combination values
(for borderBottom) borderBottomColor || borderBottomStyle || borderBottomWidth
(for borderLeft) borderLeftColor || borderLeftStyle || borderLeftWidth
(for borderRight) borderRightColor || borderRightStyle || borderRightWidth
(for borderTop) borderTopColor || borderTopStyle || borderTopWidth
Example:
```
elementRef.style.borderLeft = "#3300ff solid 2px";
```

borderBottomColor
borderLeftColor
borderRightColor
borderTopColor

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Color for a single border edge of an element.
Value: Color values; Constant `transparent`
Example: `elementRef.style.borderTopColor = "rgb(30%, 50%, 0%)";`

borderBottomStyle
borderLeftStyle
borderRightStyle
borderTopStyle

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Rendered style for a border edge of an element.
Value: Constant `none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset`. WinIE versions prior to IE5.5 do not respond to the `dotted` or `dashed` types; MacIE does not respond to the `hidden` type.
Example: `elementRef.style.borderRightStyle = "double";`

borderBottomWidth
borderLeftWidth
borderRightWidth
borderTopWidth

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Thickness of a border edge of an element.
Value: Length value; Constant `thin | medium | thick` (precise measure is at browser's discretion).
Example: `elementRef.style.borderBottomWidth = "5px";`

borderColor

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Rendered color for one to four sides of an element.
Value: Color values for one to four rectangle sides.
Example: `elementRef.style.borderColor = "green black";`

borderStyle

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Rendered style for one to four sides of an element.
Value: One to four rectangle side constants `none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset`. WinIE versions prior to IE5.5 do not respond to the `dotted` or `dashed` types; MacIE does not respond to the `hidden` type.
Example: `elementRef.style.borderStyle = "ridge";`

borderWidth

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Thickness of border for one to four sides of an element.
Value: One to four rectangle side length value or constants `thin | medium | thick` (precise dimension is at browser's discretion).
Example: `elementRef.style.borderWidth = "5px 4px 5px 3px";`

margin

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Thickness of transparent margin space outside the element's borders for one to four edges.
Value: One to four rectangle side length values.
Example: `elementRef.style.margin = "10px 5px";`

marginBottom
marginLeft
marginRight
marginTop

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Thickness of transparent margin space outside the element's borders for a single border edge.
Value: Length value
Example: `elementRef.style.marginBottom = "50px";`

mozBorderRadius
mozBorderRadiusBottomLeft
mozBorderRadiusBottomRight
mozBorderRadiusTopLeft
mozBorderRadiusTopRight

Compatibility: WinIE-, MacIE-, NN7+, Moz1+, Safari
Controls: Radius of the border around the element. You can specify each radius corner as a series of values in the `mozBorderRadius` style (one value for all four corners; two values for top-left/bottom-right and top-right/bottom-left; three values for top-left, top-right/bottom-left, and bottom-right; four values for top-left, top-right, bottom-right, bottom-left), or set each corner radius individually with its own property.
Value: Radius length value
Example: `elementRef.style.mozBorderRadius = "20px 10px 20px 10px";`

mozOpacity

Compatibility: WinIE-, MacIE-, NN7+, Moz1+, Safari
Controls: The level of opacity (transparency) of the element as a percentage; the lower the value, the more transparent the element becomes (0% or 0.0 is completely transparent, while 100% or 1.0 is completely opaque).
Value: Percentage, or numeric value between 0.0 and 1.0.
Example: `elementRef.style.mozOpacity = "75%";`

***elementRef*.style.outline**

outline

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari1+
Controls: Up to three characteristics of an outline surrounding an element (similar to a border, but not shifting the location of internal content). This style is not fully supported in the above browsers, even though the properties are reflected in the `style` object. It does, however, work well in Safari 1.0.
Value: Combination values: `outlineColor || outlineStyle || outlineWidth`
Example: `elementRef.style.outline = "red groove 2px";`

outlineColor

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari1+
Controls: Color of all four edges of an outline. This style is not fully supported in the above browsers, even though the properties are reflected in the `style` object. It does, however, work well in Safari 1.0.
Value: Color values; Constant `invert`
Example: `elementRef.style.outlineColor = "cornflowerblue";`

outlineStyle

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari1+
Controls: Rendered style for all four sides of an element outline. This style is not fully supported in the above browsers, even though the properties are reflected in the `style` object. It does, however, work well in Safari 1.0.
Value: Constant `none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset`
Example: `elementRef.style.outlineStyle = "ridge";`

outlineWidth

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari1+
Controls: Thickness of all four sides of an element outline. This style is not fully supported in the above browsers, even though the properties are reflected in the `style` object. It does, however, work well in Safari 1.0.
Value: Length value or constant `thin | medium | thick` (precise dimension is at browser's discretion)
Example: `elementRef.style.outlineWidth = "4px";`

padding

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Thickness of space between an element's content and its borders for one to four edges.
Value: One to four rectangle side length values.
Example: `elementRef.style.padding = "5px";`

paddingBottom
paddingLeft
paddingRight
paddingTop

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Thickness of space between an element's content and its borders for a single edge.
Value: Length value
Example: `elementRef.style.paddingBottom = "20px";`

List properties

listStyle

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Up to three characteristics of a list (`ol` or `ul`) presentation. Also applies to `dd`, `dt`, and `li` elements.
Value: Combination values `listStyleImage || listStylePosition || listStyleType`
Example: `elementRef.style.listStyle = "none inside lower-alpha";`

listStyleImage

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: URL of the image to be used as a marker for a list item.
Value: URL value; Constant `none`
Example: `elementRef.style.listStyleImage = "url(custombullet.jpg)";`

listStylePosition

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Whether the marker should be formatted inside the wrapped text of its content or dangle outside the wrapped text (default).
Value: Constant `inside | outside`
Example: `elementRef.style.listStylePosition = "inside";`

listStyleType

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+
Controls: Which of the standard marker sets should be used for items in the list. A change to this property for a single `li` element causes succeeding items to be in the same style.
Value: For `ul` elements, constant `circle | disc | square`
For `ol` elements, constant `decimal | decimal-leading-zero | lower-alpha | lower-greek | lower-latin | lower-roman | upper-alpha | upper-greek | upper-latin | upper-roman`, and non-Roman formats when supported by the operating system (as in Mozilla for MacOS X): `armenian | georgian | hebrew | cjk-ideographic | hiragana | hiragana-iroha | katakana | katakana-iroha`.
Example: `elementRef.style.listStyleType = "upper-roman";`

Scroll bar properties

scrollbar3dLightColor
scrollbarArrowColor
scrollbarBaseColor
scrollbarDarkShadowColor
scrollbarFaceColor
scrollbarHighlightColor
scrollbarShadowColor
scrollbarTrackColor

Compatibility: WinIE5.5, Mac-, NN-, Moz-, Safari-
Controls: Colors of individual components of scroll bars when they are displayed for `applet`, `body`, `div`, `embed`, `object`, or `textarea` elements. To experiment with how different colors can affect the individual components, visit `http://msdn.microsoft.com/workshop/samples/author/dhtml/refs/scrollbarColor.htm`.
Value: Color values; Constant `none`
Example: `elementRef.style.scrollbarTrackColor = "hotpink";`

Table properties

borderCollapse

Compatibility: WinIE5+, MacIE5, NN6+, Moz1+, Safari-
Controls: Whether a `table` element adheres to the CSS2 separated borders model or the collapsed borders model. Style is not fully supported in MacIE5.
Value: Constant `collapse | separate`
Example: `elementRef.style.borderCollapse = "separate";`

borderSpacing

Compatibility: WinIE5+, MacIE5, NN6+, Moz1+, Safari1+
Controls: For a table following the separated borders model, the thickness of the spacing between cell rectangles (akin to the `cellspacing` attribute of `table` elements). Style is not fully supported in MacIE5.
Value: One length value (for horizontal and vertical spacing) or comma-delimited list of two length values (the first for horizontal; the second for vertical).
Example: `elementRef.style.borderSpacing = "10px";`

captionSide

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari1+
Controls: Position of the `caption` element inside a `table` element. Style is not implemented in MacIE5 and is only partially implemented in Safari.
Value: Constant `top | right | bottom | left`
Example: `elementRef.style.captionSide = "bottom";`

emptyCells

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari-
Controls: Rendering of cells and their borders when the cells have no content. Default behavior is to not render borders around empty cells. Style is not implemented in MacIE5.
Value: Constant `show | hide`
Example: `elementRef.style.emptyCells = "show";`

tableLayout

Compatibility: WinIE5+, MacIE5, NN6+, Moz1+, Safari1+
Controls: Whether table is rendered progressively based on fixed width settings of the first row of cells or is rendered after the widths of all row content can be determined. Modifying this property after a table loads has no effect on the table.
Value: Constant `auto | fixed`
Example: `elementRef.style.tableLayout = "auto";`

Page and printing properties

orphans
widows

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari-
Controls: The minimum number of lines of a paragraph to be displayed at the bottom of a page (orphans) or top of a page (widows) when a page break occurs.
Value: Integer
Example: `elementRef.style.orphans = "4";`

page

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari-
Controls: The page (defined in an `@page` rule) with which the current element should be associated for printing.
Value: Identifier assigned to an existing `@page` rule
Example: `elementRef.style.page = "landscape";`

pageBreakAfter
pageBreakBefore

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari-
Controls: Whether a printed page break should be before or after the current element and the page break type.
Value: Constant `auto | always | avoid | left | right`
Example: `elementRef.style.pageBreakBefore = "always";`

pageBreakInside

Compatibility: WinIE-, MacIE5, NN6+, Moz1+, Safari-
Controls: Whether a printed page break is allowed inside an element.
Value: Constant `auto | avoid`
Example: `elementRef.style.pageBreakInside = "avoid";`

size

Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-
Controls: The size or orientation of the page box (linked to the style rule via the page property) used to determine printed pages.
Value: One (same value for width and height) or two space-delimited (width and height) length values; constant `auto | portrait | landscape`
Example: `elementRef.style.size = "portrait";`

Miscellaneous properties

accelerator

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-
Controls: Whether an accelerator key is defined for an element.
Value: Boolean
Example: `elementRef.style.accelerator = "true";`

behavior

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-
Controls: The external behavior to be applied to the current element.
Value: Space-delimited list of URL values. URLs can be a file location, an `object` element ID, or one of the built-in (default) behaviors.
Example: `elementRef.style.behavior = "url(#default#anchorClick)";`

cssText

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari-
Controls: Actual CSS rule text (read-only). This property exists by virtue of the browser's object model and is not part of the CSS specification. There is no corresponding CSS attribute.
Value: String
Example: var cssRuleText = `elementRef.style.cssText;`

imeMode

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-
Controls: Whether text is entered into a text `input` or `textarea` element through the Input Method Editor (for languages, such as Chinese, Japanese, or Korean).
Value: Constant `auto | active | inactive | disabled`
Example: `elementRef.style.imeMode = "active";`

Aural properties

Although these properties are defined in the CSS2 specification and placeholders exist for them in Mozilla-based browsers the styles are not implemented. The script equivalent properties are listed here for the sake of completeness only.

```
azimuth
cue
cueAfter
cueBefore
elevation
pause
pauseAfter
pauseBefore
pitch
pitchRange
playDuring
richness
speak
speakHeader
speakNumeral
speakPunctuation
speechRate
stress
voiceFamily
volume
```

Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-
Controls: A variety of styles primarily for browsers that support speech synthesis output.
Value: Consult `http://www.w3.org/TR/REC-CSS2/aural.html` for details on aural stylesheets.

filter Object

Properties	*Methods*	*Event Handlers*
See text		

Syntax

Accessing `filter` object properties and methods:

```
(IE4+)    document.all.objectID.filters[i].property | method([parameters])
(IE5.5+)  document.all.objectID.filters[filterName].property |
method([parameters])
```

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

About this object

Earlier in this chapter, the `style.filter` property was shown to allow reading and writing of the string value that is assigned to an element's `style.filter` property. Filters are available in WinIE only, even though MacIE5 returns the `style.filter` property value. The purpose of this section is to teach you not how to use filters but rather, how to script them.

Multiple filters are merely part of the space-delimited list of filters. Some filter types have additional specifications. For example, the `glow()` filter has three properties that more clearly define how the element should be rendered with a glow effect. The stylesheet rule for an element whose ID is `glower` looks like the following:

```
#glower {filter:glow(color=yellow, strength=5, enabled=true)}
```

Accessing the `currentStyle.filter` property for that element yields the string value:

```
glow(color=yellow, strength=5, enabled=true)
```

Attempting to modify a single subproperty of the `glow()` filter by way of string parsing would be cumbersome and hazardous at best. For example, imagine trying to increment the glow filter's `strength` property by 5.

Reading and writing subproperties

A cleaner way to work with individual properties of a filter is to access the filter as an object belonging to the element affected by the filter. Each type of filter object has as its properties the individual sub-properties that you set in the stylesheet. Continuing with the `glow()` filter example, you could access just the `color` property of the filter as follows:

```
var currColor = document.all.glower.filters["glow"].color;
```

To modify the color, assign a new value to the `filter` object's property:

```
document.all.glower.filters["glow"].color = "green";
```

To increment a numeric value, such as increasing the `glow()` filter's `strength` property by 5, use a construction such as the following (long-winded though it may be):

```
document.all.glower.filters["glow"].strength =
   document.all.glower.filters["glow"].strength + 5;
```

Table 26-1 lists the filter object names that work all the way back to IE4 and the properties associated with each filter type.

Table 26-1: IE4-Compatible Static Filter Types

Filter Name	*Description and Properties*		
`alpha()`	Transparency level		
	Properties:	`opacity`	(0 to 100)
		`finishopacity`	(0 to 100)
		`style`	(gradient shape 0 to 3)
		`startX`	(coordinate integer)
		`startY`	(coordinate integer)
		`finishX`	(coordinate integer)
		`finishY`	(coordinate integer)

Filter Name	*Description and Properties*		
`blur()`	Simulating blurred motion		
	Properties:	`add`	(1 or 0)
		`direction`	(0, 45, 90, 135, 180, 225, 270, 315)
		`strength`	(pixel count)
`chroma()`	Color transparency		
	Properties:	`color`	(color value)
`dropShadow()`	Shadow effect		
	Properties:	`color`	(color value)
		`offx`	(horizontal offset pixels)
		`offy`	(vertical offset pixels)
		`positive`	(1 or 0)
`flipH()`	Horizontally mirrored image		
	Properties:	None	
`flipV()`	Vertically mirrored image		
	Properties:	None	
`glow()`	Outer edge radiance		
	Properties:	`color`	(color value)
		`strength`	(intensity 1 to 255)
`gray()`	Eliminate color		
	Properties:	None	
`invert()`	Opposite hue, saturation, brightness levels		
	Properties:	None	
`light()`	Add light source (controlled by methods)		
	Properties:	None	
`mask()`	Overlay transparent mask		
	Properties:	`color`	(color value)
`shadow()`	Render as silhouette		
	Properties:	`color`	(color value)
		`direction`	(0, 45, 90, 135, 180, 225, 270, 315)
`wave()`	Add sine-wave distortion		
	Properties:	`add`	(1 or 0)
		`freq`	(integer number of waves)
		`light`	(strength 0 to 100)
		`phase`	(percentage offset 0 to 100)
		`strength`	(intensity 0 to 255)
`xRay()`	Render edges only		
	Properties:	None	

In addition to the static filter types, which are applied to content and sit there unless modified by script, the IE4+ `filter` object also provides types for blends and reveals for transitions between visible and invisible elements. Scripting transitions to act when a script hides or shows an element requires a few lines of code, including calls to some of the `filter` object's methods. First, Table 26-2 shows the IE4+ syntax for transition filters.

Table 26-2: IE4+ Transition Filters

FilterName	*Description and Properties*		
`blendTrans()`	Fades out old element, fades in new element		
	Properties:	`duration`	(floating-point number of seconds)
	Methods:	`apply()`	(freezes current display)
		`play()`	(plays the transition)
		`stop()`	(stops transition mid-stream)
`revealTrans()`	Reveals element to be shown through an effect		
	Properties:	`duration`	(floating-point number of seconds)
		`transition`	(code number for effect)
			`0` Box in
			`1` Box out
			`2` Circle in
			`3` Circle out
			`4` Wipe up
			`5` Wipe down
			`6` Wipe right
			`7` Wipe left
			`8` Vertical blinds
			`9` Horizontal blinds
			`10` Checkerboard across
			`11` Checkerboard down
			`12` Random dissolve
			`13` Split vertical in
			`14` Split vertical out
			`15` Split horizontal in
			`16` Split horizontal out
			`17` Strips left down
			`18` Strips left up
			`19` Strips right down
			`20` Strips right up
			`21` Random bars horizontally
			`22` Random bars vertically
			`23` Random effect
	Methods:	`apply()`	(freezes current display)
		`play()`	(plays the transition)
		`stop()`	(stops transition mid-stream)

To make a transition work under script control, a filter must be applied to the element that you want the transition to work on. That can be done by script or by assigning a filter style to the element. As for the scripting, you begin by invoking the `apply()` method of the desired `filter` object. Next, script the change, such as assigning a new URL to the `src` property of an `img` element. While you do this, the `apply()` method freezes the image until you invoke

the `play()` method on the filter. Listing 26-1 effects a checkerboard transition between two images after you click the image.

Listing 26-1: **A Reveal Transition Between Images**

```
<html>
   <head>
      <title>IE4+ Transition</title>
      <style type="text/css">
      img {filter:revealTrans(transition=10)}
      </style>
      <script type="text/javascript">
      function doReveal() {
         document.all.myIMG.filters["revealTrans"].apply();
         if (document.all.myIMG.src.indexOf("desk1") != -1) {
            document.all.myIMG.src = "desk3.gif";
         } else {
            document.all.myIMG.src = "desk1.gif";
         }
         document.all.myIMG.filters["revealTrans"].play();
      }
      </script>
   </head>
   <body>
      <h1>IE4+ Transition</h1>
      <hr />
      <p>Click on the image to cause a reveal transition.</p>
      <img alt="image" id="myIMG" src="desk1.gif" height="90" width="120"
      onclick="doReveal()" />
   </body>
</html>
```

Building on the example in Listing 26-1, the next example in Listing 26-2 demonstrates how a script can also modify a `filter` object's property, including a transition filter. Before the transition filter has its `apply()` method invoked, the script sets the transition type based on a user choice in a `select` list.

Listing 26-2: **Choosing Reveal Transitions Between Images**

```
<html>
   <head>
      <title>IE4+ Transition and Choices</title>
      <style type="text/css">
      img {filter:revealTrans(transition=10)}
      </style>
      <script type="text/javascript">
      function doReveal() {
         document.all.myIMG.filters["revealTrans"].transition =
document.forms[0].transChoice.value;
         document.all.myIMG.filters["revealTrans"].apply();
         if (document.all.myIMG.src.indexOf("desk1") != -1) {
            document.all.myIMG.src = "desk3.gif";
         } else {
```

Continued

Listing 26-2 *(continued)*

```
            document.all.myIMG.src = "desk1.gif";
         }
         document.all.myIMG.filters["revealTrans"].play();
      }
      </script>
   </head>
   <body>
      <h1>IE4+ Transition and Choices</h1>
      <hr />
      <form>
         <p>Choose the desired transition type: <select name="transChoice">
            <option value="0">Box in</option>
            <option value="1">Box out</option>
            <option value="2">Circle in</option>
            <option value="3">Circle out</option>
            <option value="4">Wipe up</option>
            <option value="5">Wipe down</option>
            <option value="6">Wipe right</option>
            <option value="7">Wipe left</option>
            <option value="8">Vertical blinds</option>
            <option value="9">Horizontal blinds</option>
            <option value="10">Checkerboard across</option>
            <option value="11">Checkerboard down</option>
            <option value="12">Random dissolve</option>
            <option value="13">Split vertical in</option>
            <option value="14">Split vertical out</option>
            <option value="15">Split horizontal in</option>
            <option value="16">Split horizontal out</option>
            <option value="17">Strips left down</option>
            <option value="18">Strips left up</option>
            <option value="19">Strips right down</option>
            <option value="20">Strips right up</option>
            <option value="21">Random bars horizontally</option>
            <option value="22">Random bars vertically</option>
            <option value="23">Random effect</option>
         </select></p>
      </form>
      <p>Click on the image to cause a reveal transition.</p>
      <img alt="image" id="myIMG" src="desk1.gif" height="90" width="120"
      onclick="doReveal()" />
   </body>
</html>
```

WinIE5.5+ filter syntax changes

While WinIE5.5+ still supports the original IE4 way of controlling filters, the browser also implements a new filter component, which Microsoft strongly encourages authors to use (as evidenced by the difficulty in finding documentation for the IE4 syntax at its developer Web site). In the process of implementing this new filter component, the names of many filters change, as do their individual properties. Moreover, the way the filter component is invoked in the stylesheet is also quite different from the original component.

The stylesheet syntax requires a reference to the new component as well as the filter name. Here is the old way:

```
#glower {filter:glow(color=yellow, strength=5, enabled=true)}
```

And here is the new way:

```
#glower {filter:progid:DXImageTransform.Microsoft.Glow(color=yellow,
   strength=5, enabled=true)}
```

Don't overlook the extra `progid:` pointer in the reference. This program identifier becomes part of the filter name that your scripts use to reference the filter:

```
document.getElementById("glower").filters[
   "DXImageTransform.Microsoft.Glow"].color = "green";
```

While some of the filter names and properties stay the same (except for the huge prefix), several older properties are subsumed by new filters whose properties help identify the specific effect. The former `revealTrans()` filter is now divided among several new filters dedicated to transition effects. Table 26-3 shows the IE5.5+ syntax.

Using the new syntax in IE5.5+ can cause frequent crashes of the browser (at least early released versions), especially transition filters. If you implement the new syntax, be sure to torture-test your pages extensively. Ideally, you should encourage users of these pages to run IE6+.

Table 26-3: IE5.5 DXImageTransform.Microsoft Filter Names

Filter Name	*Description and Properties*		
`Alpha()`	Transparency level		
	Properties:	`opacity`	(0 to 100)
		`finishopacity`	(0 to 100)
		`style`	(gradient shape 0 to 3)
		`startX`	(coordinate integer)
		`startY`	(coordinate integer)
		`finishX`	(coordinate integer)
		`finishY`	(coordinate integer)
`Barn()`	Barn-door style transition		
	Properties:	`duration`	(floating-point number of seconds)
		`motion`	(`in` or `out`)
		`orientation`	(`horizontal` or `vertical`)
		`percent`	(0 to 100)
		`status`	0 (stopped), 1 (applied), 2 (playing)
	Methods:	`apply()`	(freezes current display)
		`play()`	(plays the transition)
		`stop()`	(stops transition mid-stream)

Continued

Table 26-3 *(continued)*

Filter Name	*Description and Properties*		
`BasicImage()`	Element rotation, flip, color effects, and opacity		
	Properties:	`grayScale`	(1 or 0)
		`invert`	(1 or 0)
		`mask`	(1 or 0)
		`maskColor`	(color value)
		`mirror`	(1 or 0)
		`opacity`	(0.0 to 1.0)
		`rotation`	0 (no rotation), 1 (90°), 2 (180°), 3 (270°)
		`xRay`	(1 or 0)
`Blinds()`	Action transition with Venetian blind effect		
	Properties:	`direction`	(`up`, `down`, `right`, `left`)
		`squaresX`	(integer column count)
		`squaresY`	(integer row count)
		`status`	0 (stopped), 1 (applied), 2 (playing)
	Methods:	`apply()`	(freezes current display)
		`play()`	(plays the transition)
		`stop()`	(stops transition mid-stream)
`Checkerboard()`	Action transition with checkerboard effect		
	Properties:	`bands`	(1 to 100)
		`direction`	(`up`, `down`, `right`, `left`)
		`duration`	(floating-point number of seconds)
		`percent`	(0 to 100)
		`slideStyle`	(`HIDE`, `PUSH`, `SWAP`)
		`status`	0 (stopped), 1 (applied), 2 (playing)
	Methods:	`apply()`	(freezes current display)
		`play()`	(plays the transition)
		`stop()`	(stops transition mid-stream)
`Chroma()`	Color transparency		
	Properties:	`color`	(color value)
`DropShadow()`	Shadow effect		
	Properties:	`color`	(color value)
		`offx`	(horizontal offset pixels)
		`offy`	(vertical offset pixels)
		`positive`	(1 or 0)
`Fade()`	Blend transition		
	Properties:	`duration`	(floating-point number of seconds)
		`overlap`	(0.0 to 1.0 seconds)
		`percent`	(0 to 100)
		`status`	0 (stopped), 1 (applied), 2 (playing)

Filter Name	*Description and Properties*		
	Methods:	`apply()`	(freezes current display)
		`play()`	(plays the transition)
		`stop()`	(stops transition mid-stream)
`Glow()`	Outer edge radiance		
	Properties:	`color`	(color value)
		`strength`	(intensity 1 to 255)
`Iris()`	Action transition with zoom effect		
	Properties:	`duration`	(floating-point number of seconds)
		`irisStyle`	(`CIRCLE`, `CROSS`, `DIAMOND`, `PLUS`, `SQUARE`, `STAR`)
		`motion`	(`in` or `out`)
		`percent`	(0 to 100)
		`status`	0 (stopped), 1 (applied), 2 (playing)
	Methods:	`apply()`	(freezes current display)
		`play()`	(plays the transition)
		`stop()`	(stops transition mid-stream)
`Light()`	Add light source (controlled by methods)		
	Properties:	None	
	Methods:	`addAmbient (red, green, blue, strength)`	
		`addCone (sourceLeft, sourceTop, sourceZAxis, targetLeft, targetTop, red, green, blue, strength, spreadAngle)`	
		`addPoint (sourceLeft, sourceTop, sourceZAxis, red, green, blue, strength)`	
		`changeColor (lightID, red, green, blue, absoluteColorFlag)`	
		`changeStrength (lightID, strength, absoluteIntensityFlag)`	
		`clear()`	
		`moveLight (lightID, sourceLeft, sourceTop, sourceZAxis, absoluteMovementFlag)`	
`MaskFilter()`	Overlay transparent mask		
	Properties:	`color`	(color value)
`MotionBlur()`	Simulating blurred motion		
	Properties:	`add`	(1 or 0)
		`direction`	(0, 45, 90, 135, 180, 225, 270, 315)
		`strength`	(pixel count)
`RandomDissolve()`	Pixelated dissolve transition		
	Properties:	`duration`	(floating-point number of seconds)
		`percent`	(0 to 100)
		`status`	0 (stopped), 1 (applied), 2 (playing)
	Methods:	`apply()`	(freezes current display)
		`play()`	(plays the transition)
		`stop()`	(stops transition mid-stream)

Continued

Table 26-3 *(continued)*

Filter Name	*Description and Properties*		
`RandomBars()`	Bar style transition		
	Properties:	`duration`	(floating-point number of seconds)
		`orientation`	(`horizontal` or `vertical`)
		`percent`	(0 to 100)
		`status`	0 (stopped), 1 (applied), 2 (playing)
	Methods:	`apply()`	(freezes current display)
		`play()`	(plays the transition)
		`stop()`	(stops transition mid-stream)
`Shadow()`	Render as silhouette		
	Properties:	`color`	(color value)
		`direction`	(0, 45, 90, 135, 180, 225, 270, 315)
`Stripes()`	Striped style transition		
	Properties:	`duration`	(floating-point number of seconds)
		`motion`	(`in` or `out`)
		`percent`	(0 to 100)
		`status`	0 (stopped), 1 (applied), 2 (playing)
	Methods:	`apply()`	(freezes current display)
		`play()`	(plays the transition)
		`stop()`	(stops transition mid-stream)
`Wave()`	Add sine-wave distortion		
	Properties:	`add`	(1 or 0)
		`freq`	(integer number of waves)
		`light`	(strength 0 to 100)
		`phase`	(percentage offset 0 to 100)
		`strength`	(intensity 0 to 255)
`xRay()`	Render edges only		
	Properties:	None	

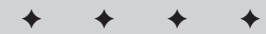
For more details on deploying filters in IE for Windows, visit `http://msdn.microsoft.com/workshop/author/filter/filters.asp`. Because most of the live examples require WinIE5.5+, be sure to use that version for the best experience at that page.

✦ ✦ ✦

PART

IV

JavaScript Core Language Reference



In This Part

Chapter 27
The String Object

Chapter 28
The Math, Number, and Boolean Objects

Chapter 29
The Date Object

Chapter 30
The Array Object

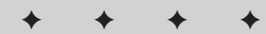
Chapter 31
Control Structures and Exception Handling

Chapter 32
JavaScript Operators

Chapter 33
Functions and Custom Objects

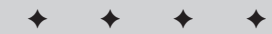
Chapter 34
Global Functions and Statements

Chapter 35
Body Text Objects



CHAPTER 27

The String Object

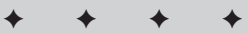


In This Chapter

How to parse and work with text

Performing search-and-replace operations

Scripted alternatives to text formatting



Chapter 6's tutorial introduced you to the concepts of values and the types of values that JavaScript works with — features, such as strings, numbers, and Boolean values. In this chapter, you look more closely at the very important String data type, as well as its relationship to the Number data type. Along the way, you encounter the many ways in which JavaScript enables scripters to manipulate strings.

Much of the syntax that you see in this chapter is identical to that of the Java programming language. Because the scope of JavaScript activity is narrower than that of Java, you don't have nearly as much to learn for JavaScript as for Java.

String and Number Data Types

Although JavaScript is what is known as a "loosely typed" language, you still need to be aware of several data types because of their impact on the way you work with the information in those forms. In this section, I focus on strings and two types of numbers.

Simple strings

A *string* consists of one or more standard text characters between matching quote marks. JavaScript is forgiving in one regard: You can use single or double quotes, as long as you match two single quotes or two double quotes around a string. Another benefit to this scheme becomes apparent when you try to include a quoted string inside a string. For example, say that you're assembling a line of HTML code in a variable that you will eventually write to a new window completely controlled by JavaScript. The line of text that you want to assign to a variable is the following:

```
<input type="checkbox" name="candy" />Chocolate
```

To assign this entire line of text to a variable, you have to surround the line in quotes. But because quotes appear inside the string, JavaScript (or any language) has problems deciphering where the string begins or ends. By carefully placing the other kind of quote pairs, however, you can make the assignment work. Here are two equally valid ways:

```
result = '<input type="checkbox" name="candy" />Chocolate';
result = "<input type='checkbox' name='candy' />Chocolate";
```

Notice that in both cases, the same unique pair of quotes surrounds the entire string. Inside the string, two quoted strings appear that are treated as such by JavaScript. I recommend that you settle on one form or the other, and then use that form consistently throughout your scripts.

Building long string variables

The act of joining strings together — concatenation — enables you to assemble long strings out of several little pieces. This feature is very important for some of your scripting — for example, when you need to build an HTML page's specifications entirely within a variable before writing the page to another frame with one `document.write()` statement.

One tactic that I use keeps the length of each statement in this building process short enough so that it's easily readable in your text editor. This method uses the add-by-value assignment operator (+=) that appends the right-hand side of the equation to the left-hand side. Here is a simple example, which begins by initializing a variable as an empty string:

```
var newDocument = "";
newDocument += "<html><head><title>Life and Times</title></head>";
newDocument += "<body><h1>My Life and Welcome to It</h1>";
newDocument += "by Sidney Finortny<hr />";
```

Starting with the second line, each statement adds more data to the string being stored in `newDocument`. You can continue appending string data until the entire page's specification is contained in the `newDocument` variable.

Excessive use of the add-by-value operator involving large quantities of text can become inefficient. If you are experiencing slow performance when accumulating large strings, try pushing your string segments into items of an array (see Chapter 30). Then use the array's `join()` method to generate the resulting large string value.

Joining string literals and variables

In some cases, you need to create a string out of literal strings (characters with quote marks around them) and string variable values. The methodology for concatenating these types of strings is no different from that of multiple string literals. The plus-sign operator does the job. Therefore, in the following example, a variable contains a name. That variable value is made a part of a larger string whose other parts are string literals:

```
yourName = prompt("Please enter your name:","");
var msg = "Good afternoon, " + yourName + ".";
alert(msg);
```

Some common problems that you may encounter while attempting this kind of concatenation include the following:

- ✦ Accidentally omitting one of the quotes around a literal string
- ✦ Failing to insert blank spaces in the string literals to accommodate word spaces
- ✦ Forgetting to concatenate punctuation after a variable value

Also, don't forget that what I show here as variable values can be any expression that evaluates to a string, including property references and the results of some methods. For example

```
var msg = "The name of this document is " + document.title + ".";
alert(msg);
```

Special inline characters

The way string literals are created in JavaScript makes adding certain characters to strings difficult. I'm talking primarily about adding quotes, carriage returns, apostrophes, and tab characters to strings. Fortunately, JavaScript provides a mechanism for entering such

characters into string literals. A backslash symbol, followed by the character that you want to appear as inline, makes that task happen. For the "invisible" characters, a special set of letters following the backslash tells JavaScript what to do.

The most common backslash pairs are as follows:

- ✦ `\"` Double quote
- ✦ `\'` Single quote (apostrophe)
- ✦ `\\` Backslash
- ✦ `\b` Backspace
- ✦ `\t` Tab
- ✦ `\n` New line
- ✦ `\r` Carriage return
- ✦ `\f` Form feed

Use these "inline characters" (also known as "escaped characters," but this terminology has a different connotation for Internet strings) inside quoted string literals to make JavaScript recognize them. When assembling a block of text that needs a new paragraph, insert the `\n` character pair. Here are some examples of syntax using these special characters:

```
msg = "You\'re doing fine.";
msg = "This is the first line.\nThis is the second line.";
msg = document.title + "\n" + document.links.length + " links present.";
```

Technically speaking, a complete carriage return, as known from typewriting days, is both a line feed (advance the line by one) and a carriage return (move the carriage all the way to the left margin). Although JavaScript strings treat a line feed (`\n` new line) as a full carriage return, you may have to construct `\r\n` breaks when assembling strings that go back to a `cgi` script on a server. The format that you use all depends on the string-parsing capabilities of the `cgi` program. (Also see the special requirements for the `textarea` object in Chapter 20.)

Confusing the strings assembled for display in `textarea` objects or alert boxes with strings to be written as HTML is easy. For HTML strings, make sure that you use the standard HTML tags for line-breaks (`<br>`) and paragraph breaks (`<p>`) rather than the inline return or line feed symbols.

String Object

Properties	*Methods*
`constructor`	`anchor()`
`length`	`big()`
`prototype`†	`blink()`
	`bold()`
	`charAt()`
	`charCodeAt()`
	`concat()`

Continued

stringObject

Properties	*Methods*
	`fixed()`
	`fontcolor()`
	`fontsize()`
	`fromCharCode()`†
	`indexOf()`
	`italics()`
	`lastIndexOf()`
	`link()`
	`localeCompare()`
	`match()`
	`replace()`
	`search()`
	`slice()`
	`small()`
	`split()`
	`strike()`
	`sub()`
	`substr()`
	`substring()`
	`sup()`
	`toLocaleLowerCase()`
	`toLocaleUpperCase()`
	`toLowerCase()`
	`toString()`
	`toUpperCase()`
	`valueOf()`

†Member of the static `String` object

Syntax

Creating a string object:

```
var myString = new String("characters");
```

Creating a string value:

```
var myString = "characters";
```

Accessing static `String` object properties and methods:

```
String.property | method([parameters])
```

Accessing string object properties and methods:

```
string.property | method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

About this object

JavaScript draws a fine line between a string value and a string object. Both let you use the same methods on their contents, so that by and large, you do not have to create a string object (with the `new String()` constructor) every time you want to assign a string value to a variable. A simple assignment operation (`var myString = "fred"`) is all you need to create a string value that behaves on the surface very much like a full-fledged string object.

Where the difference comes into play is when you want to exploit the "object-ness" of a genuine string object, which I explain further in the discussion of the *string*`.prototype` property later in this chapter. You may also encounter the need to use a full-fledged string object when passing string data to Java applets. If you find that your applet doesn't receive a string value as a Java `String` data type, then create a new string object via the JavaScript constructor function before passing the value onto the applet.

With string data often comes the need to massage that text in scripts. In addition to concatenating strings, you at times need to extract segments of strings, delete parts of strings, and replace one part of a string with some other text. Unlike many plain-language scripting languages, JavaScript is fairly low-level in its built-in facilities for string manipulation. This characteristic means that unless you can take advantage of the regular expression powers of NN4+/IE4+ or advanced array techniques, you must fashion your own string handling routines out of very elemental powers built into JavaScript. Later in this chapter, I provide several functions that you can use in your own scripts for common string handling in a manner fully compatible with older browsers.

As you work with string values, visualize every string value as an object with properties and methods like other JavaScript objects. The latest versions of JavaScript define a few properties and a slew of methods for any string value (and one extra property for the static `String` object that is always present in the context of the browser window). The syntax is the same for string methods as it is for any other object method:

```
stringObject.method()
```

What may seem odd at first is that the *stringObject* part of this reference can be any expression that evaluates to a string, including string literals, variables containing strings, methods or functions that return strings, or other object properties. Therefore, the following examples of calling the `toUpperCase()` method are all valid:

```
"george burns".toUpperCase()
yourName.toUpperCase() // yourName is a variable containing a string
window.prompt("Enter your name","").toUpperCase()
document.forms[0].entry.value.toUpperCase() // entry is a text field object
```

An important concept to remember is that invoking a string method does not change the string object that is part of the reference. Rather, the method returns a value, which can be used as a parameter to another method or function call, or assigned to a variable value.

Therefore, to change the contents of a string variable to the results of a method, you must use an assignment operator, as in

```
yourName = yourName.toUpperCase(); // variable is now all uppercase
```

Properties

constructor

Value: Function reference. Read/Write
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The `constructor` property is a reference to the function that was invoked to create the current string. For a native JavaScript string object, the constructor function is the built-in `String()` constructor.

When you use the `new String()` constructor to create a string object, the type of the value returned by the constructor is `object` (meaning the `typeof` operator returns `object`). Therefore, you can use the `constructor` property on an object value to see if it is a string object:

```
if (typeof someValue == "object" ) {
   if (someValue.constructor == String) {
      // statements to deal with string object
   }
}
```

Although the property is read/write, and you can assign a different constructor to the `String.prototype`, the native behavior of a `String` object persists through the new constructor.

Example

Use The Evaluator (Chapter 13) to test the value of the `constructor` property. Enter the following statements into the top text box:

```
a = new String("abcd")
a.constructor == String
a.constructor == Number
```

Related Items: `prototype` property.

`length`

Value: Integer. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The most frequently used property of a string is `length`. To derive the length of a string, read its property as you would read the `length` property of any object:

```
string.length
```

The length value represents an integer count of the number of characters within the string. Spaces and punctuation symbols count as characters. Any backslash special characters embedded in a string count as one character, including such characters as newline and tab. Here are some examples:

```
"Lincoln".length // result = 7
"Four score".length // result = 10
"One\ntwo".length // result = 7
"".length // result = 0
```

The `length` property is commonly summoned when dealing with detailed string manipulation in repeat loops.

`prototype`

Value: String object. Read/Write
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

`String` objects defined with the `new String("stringValue")` constructor are robust objects compared with plain, old variables that are assigned string values. You certainly don't

have to create this kind of string object for every string in your scripts, but these objects do come in handy if you find that strings in variables go awry. This happens occasionally while trying to preserve string information as script variables in other frames or windows. By using the string object constructor, you can be relatively assured that the string value will be available in the distant frame when needed.

Another byproduct of true string objects is that you can assign prototype properties and methods to all string objects in the document. A *prototype* is a property or method that becomes a part of every new object created after the prototype items are added. For strings, as an example, you may want to define a new method for converting a string into a new type of HTML font tag not already defined by the JavaScript string object. Listing 27-1 shows how to create and use such a prototype.

Listing 27-1: **A String Object Prototype**

```
<html>
  <head>
     <title>String Object Prototype</title>
     <script type="text/javascript">
     function makeItHot() {
        return "<font color='red'>" + this.toString() + "<\/font>";
     }
     String.prototype.hot = makeItHot;
     </script>
  </head>
  <body>
     <script type="text/javascript">
     document.write("<h1>This site is on " + "FIRE".hot() + "!!<\/h1>");
     </script>
  </body>
</html>
```

A function definition (`makeItHot()`) accumulates string data to be returned to the object when the function is invoked as the object's method. The `this` keyword refers to the object making the call, which you convert to a string for concatenation with the rest of the strings to be returned. In the page's Body, that prototype method is invoked in the same way one invokes existing `String` methods that turn strings into HTML tags (discussed later in this chapter).

In the next sections, I divide string object methods into two distinct categories. The first, parsing methods, focuses on string analysis and character manipulation within strings. The second group, formatting methods, is devoted entirely to assembling strings in HTML syntax for those scripts that assemble the text to be written into new documents or other frames.

Parsing methods

string.charAt(*index*)

Returns: One-character string.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Use the *string*.charAt() method to read a single character from a string when you know the position of that character. For this method, you specify an index value in the string as a

parameter to the method. The index value of the first character of the string is 0. To grab the last character of a string, mix string methods:

```
myString.charAt(myString.length - 1)
```

If your script needs to get a range of characters, use the *string*.substring() method. Using *string*.substring() to extract a character from inside a string is a common mistake, when the *string*.charAt() method is more efficient.

Example

Enter each of the following statements into the top text box of The Evaluator:

```
a = "banana daiquiri"
a.charAt(0)
a.charAt(5)
a.charAt(6)
a.charAt(20)
```

Results from each of the charAt() methods should be b, a (the third "a" in "banana"), a space character, and an empty string, respectively.

Related Items: string.lastIndexOf(), string.indexOf(), string.substring() methods.

string.charCodeAt([*index*])
String.fromCharCode(*num1* [, *num2* [, ... *numn*]])

Returns: Integer code number for a character; concatenated string value of code numbers supplied as parameters.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

Conversions from plain language characters to their numeric equivalents have a long tradition in computer programming. For a long time, the most common numbering scheme was the ASCII standard, which covers the basic English, alphanumeric characters and punctuation within 128 values (numbered 0 through 127). An extended version with a total of 256 characters, with some variations depending on the operating system, accounts for other roman characters in other languages, particularly vowels with umlauts and other pronunciation marks. To bring all languages, including pictographic languages and other non-Roman alphabets, into the computer age, a world standard called Unicode provides space for thousands of characters.

In JavaScript, the character conversions are string methods. Acceptable values depend on the browser that you are using. NN4 works only with the 256 ISO-Latin-I values; IE4+, NN6+, and W3C browsers work with the Unicode system.

The two methods that perform these conversions work in very different ways syntactically. The first, *string*.charCodeAt(), converts a single string character to its numerical equivalent. The string being converted is the one to the left of the method name — and the string may be a literal string or any other expression that evaluates to a string value. If no parameter is passed, the character being converted is by default the first character of the string. However, you can also specify a different character as an index value into the string (first character is 0), as demonstrated here:

```
"abc".charCodeAt()  // result = 97
"abc".charCodeAt(0)  // result = 97
"abc".charCodeAt(1)  // result = 98
```

If the string value is an empty string or the index value is beyond the last character, the result is NaN.

To convert numeric values to their characters, use the `String.fromCharCode()` method. Notice that the object beginning the method call is the static `String` object, not a string value. Then, as parameters, you can include one or more integers separated by commas. In the conversion process, the method combines the characters for all of the parameters into one string, an example of which is shown here:

```
String.fromCharCode(97, 98, 99)  // result "abc"
```

Although recent browsers support character values across the entire Unicode range, the browser won't render characters above 255 unless the computer is equipped with language and font support for the designated language.

Example

Listing 27-2 provides examples of both methods on one page. Moreover, because one of the demonstrations relies on the automatic capture of selected text on the page, the scripts include code to accommodate the different handling of selection events and capture of the selected text in a variety of browsers.

After you load the page, select part of the body text anywhere on the page. If you start the selection with the lowercase letter "a," the character code displays as 97. If you select no text, the result is `NaN`.

Try entering numeric values in the three fields at the bottom of the page. Values below 32 are ASCII control characters that most fonts represent as hollow squares. But try all other values to see what you get. Notice that the script passes all three values as a group to the `String.fromCharCode()` method, and the result is a combined string. Thus, Figure 27-1 shows what happens when you enter the uppercase ASCII values for a three-letter animal name.

Listing 27-2: **Character Conversions**

```
<html>
   <head>
      <title>Character Codes</title>
      <script type="text/javascript">
      function showCharCode() {
         var theText = "";
         if (window.getSelection) {
            theText = window.getSelection().toString();
         } else if (document.getSelection) {
            theText = document.getSelection();
         } else if (document.selection && document.selection.createRange) {
            theText = document.selection.createRange().text;
         }
         if (theText) {
            document.forms[0].charCodeDisplay.value = theText.charCodeAt();
         } else {
            document.forms[0].charCodeDisplay.value = " ";
         }
      }
      function showString(form) {
         form.result.value =
String.fromCharCode(form.entry1.value,form.entry2.value,form.entry3.value);
      }
```

Continued

Listing 27-2 *(continued)*

```
      document.onmouseup = showCharCode;
      </script>
   </head>
   <body>
      <b>Capturing Character Codes</b>
      <form>
         Select any of this text, and see the character code of the first
         character.
         <p>Character Code:<input type="text" name="charCodeDisplay"
            size="3" /><br /></p>
         <hr />
         <b>Converting Codes to Characters</b><br />
         Enter a value 0-255:<input type="text" name="entry1" size="6" /><br />
         Enter a value 0-255:<input type="text" name="entry2" size="6" /><br />
         Enter a value 0-255:<input type="text" name="entry3" size="6" /><br />
         <input type="button" value="Show String"
         onclick="showString(this.form)" /> Result:<input type="text"
         name="result" size="5" />
      </form>
   </body>
</html>
```

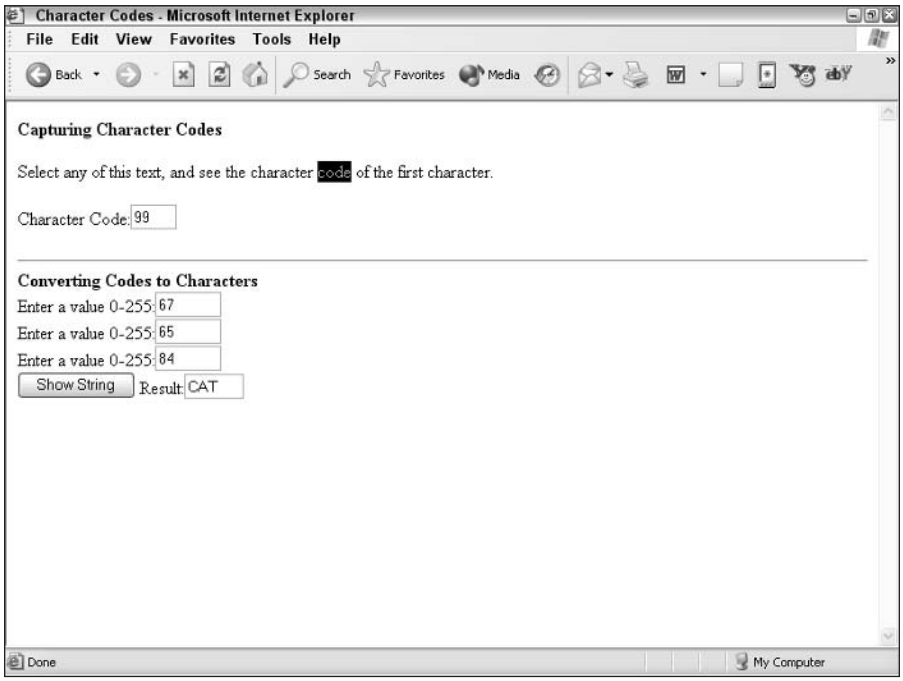


Figure 27-1: Conversions from text characters to ASCII values and vice versa.

Related Items: None.

string.concat(*string2*)

Returns: Combined string.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

JavaScript's add-by-value operator (+=) provides a convenient way to concatenate strings. Recent browsers, however, include a string object method that performs the same task. The base string to which more text is appended is the object or value to the left of the period. The string to be appended is the parameter of the method, as the following example demonstrates:

```
"abc".concat("def")  // result: "abcdef"
```

As with the add-by-value operator, the `concat()` method doesn't know about word endings. You are responsible for including the necessary space between words if the two strings require a space between them in the result.

Related Items: Add-by-value (+=) operator.

string.indexOf(*searchString* [, *startIndex*])

Returns: Index value of the character within *string* where *searchString* begins.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Like some languages' offset string function, JavaScript's `indexOf()` method enables your script to obtain the number of the character in the main string where a search string begins. Optionally, you can specify where in the main string the search should begin — but the returned value is always relative to the very first character of the main string. Such as all string object methods, index values start their count with `0`. If no match occurs within the main string, the returned value is `-1`. Thus, this method is a convenient way to determine whether one string contains another, regardless of position.

Example

Enter each of the following statements (up to but not including the "//" comment symbols) into the top text box of The Evaluator (you can simply replace the parameters of the `indexOf()` method for each statement after the first one). Compare your results with the results shown below.

```
a = "bananas"
a.indexOf("b")      // result = 0 (index of 1st letter is zero)
a.indexOf("a")      // result = 1
a.indexOf("a",1)    // result = 1 (start from 2nd letter)
a.indexOf("a",2)    // result = 3 (start from 3rd letter)
a.indexOf("a",4)    // result = 5 (start from 5th letter)
a.indexOf("nan")    // result = 2
a.indexOf("nas")    // result = 4
a.indexOf("s")      // result = 6
a.indexOf("z")      // result = -1 (no "z" in string)
```

Related Items: `string.lastIndexOf()`, `string.charAt()`, `string.substring()` methods.

string.lastIndexOf(*searchString*[, *startIndex*])

Returns: Index value of the last character within string where *searchString* begins.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The *string*.lastIndexOf() method is closely related to the method *string*.indexOf(). The only difference is that this method starts its search for a match from the end of the string (*string*.length - 1) and works its way backward through the string. All index values are still counted, starting with 0, from the front of the string. The examples that follow use the same values as in the examples for *string*.indexOf() so that you can compare the results. In cases where only one instance of the search string is found, the results are the same; but when multiple instances of the search string exist, the results can vary widely — hence the need for this method.

Example

Enter each of the following statements (up to, but not including the "//" comment symbols) into the top text box of The Evaluator (you can simply replace the parameters of the lastIndexOf() method for each statement after the first one). Compare your results with the results shown below.

```
a = "bananas"
a.lastIndexOf("b")    // result = 0 (index of 1st letter is zero)
a.lastIndexOf("a")    // result = 5
a.lastIndexOf("a",1)  // result = 1 (from 2nd letter toward the front)
a.lastIndexOf("a",2)  // result = 1 (start from 3rd letter working toward front)
a.lastIndexOf("a",4)  // result = 3 (start from 5th letter)
a.lastIndexOf("nan")  // result = 2 [except for -1 Nav 2.0 bug]
a.lastIndexOf("nas")  // result = 4
a.lastIndexOf("s")    // result = 6
a.lastIndexOf("z")    // result = -1 (no "z" in string)
```

Related Items: string.lastIndexOf(), string.charAt(), string.substring() methods.

string.localeCompare(*string2*)

Returns: Integer.
Compatibility: WinIE5.5+, MacIE-, NN6+, Moz1+, Safari1+

The localeCompare() method lets a script compare the cumulative Unicode values of two strings, taking into account the language system for the browser. The need for this method affects only some language systems (Turkish is said to be one). If the two strings, adjusted for the language system, are equal, the value returned is zero. If the string value on which the method is invoked (meaning the string to the left of the period) sorts ahead of the parameter string, the value returned is a negative integer; otherwise the returned value is a positive integer.

The ECMA standard for this method leaves the precise positive or negative values up to the browser designer. NN6+ calculates the cumulative Unicode values for both strings and subtracts the string parameter's sum from the string value's sum. IE5.5+, on the other hand, returns -1 or 1 if the strings are not colloquially equal.

Related Items: string.toLocaleLowerCase(), string.toLocaleUpperCase() methods.

string.match(*regExpression*)

Returns: Array of matching strings.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The *string*.match() method relies on the RegExp (regular expression) object introduced to JavaScript in version 4 browsers. The string value under scrutiny is to the left of the dot, while the regular expression to be used by the method is passed as a parameter. The parameter must be a regular expression object, created according to the two ways these objects can be generated.

This method returns an array value when at least one match turns up; otherwise the returned value is null. Each entry in the array is a copy of the string segment that matches the specifications of the regular expression. You can use this method to uncover how many times a substring or sequence of characters appears in a larger string. Finding the offset locations of the matches requires other string parsing.

Example

To help you understand the *string*.match() method, Listing 27-3 provides a workshop area for experimentation. Two fields occur for data entry: the first is for the long string to be examined by the method; the second is for a regular expression. Some default values are provided in case you're not yet familiar with the syntax of regular expressions (see Chapter 42 on the CD-ROM). A checkbox lets you specify whether the search through the string for matches should be case-sensitive. After you click the "Execute match()" button, the script creates a regular expression object out of your input, performs the *string*.match() method on the big string, and reports two kinds of results to the page. The primary result is a string version of the array returned by the method; the other is a count of items returned.

Listing 27-3: Regular Expression Match Workshop

```
<html>
   <head>
      <title>Regular Expression Match</title>
      <script type="text/javascript">
      function doMatch(form) {
         var str = form.entry.value;
         var delim = (form.caseSens.checked) ? "/g" : "/gi";
         var regexp = eval("/" + form.regexp.value + delim);
         var resultArray = str.match(regexp);
         if (resultArray) {
            form.result.value = resultArray.toString();
            form.count.value = resultArray.length;
         } else {
            form.result.value = "<no matches>";
            form.count.value = "";
         }
      }
      </script>
   </head>
```

Continued

Listing 27-3 *(continued)*

```
   <body>
      <b>String Match with Regular Expressions</b>
      <hr />
      <form>
         Enter a main string:<input type="text" name="entry" size="60"
         value="Many a maN and womAN have meant to visit GerMAny." /><br />
         Enter a regular expression to match:<input type="text" name="regexp"
         size="25" value="\wa\w" /> <input type="checkbox"
         name="caseSens" />Case-sensitive
         <p><input type="button" value="Execute match()"
            onclick="doMatch(this.form)" /> <input type="reset" /></p>
         <p>Result:<input type="text" name="result" size="40" /><br />
            Count:<input type="text" name="count" size="3" /><br /></p>
      </form>
   </body>
</html>
```

The default value for the main string has unusual capitalization intentionally. The capitalization lets you see more clearly where some of the matches come from. For example, the default regular expression looks for any three-character string that has the letter "a" in the middle. Six string segments match that expression. With the help of capitalization, you can see where each of the four strings containing "man" are extracted from the main string. The following table lists some other regular expressions to try with the default main string.

RegExp	*Description*
`man`	Both case-sensitive and not
`man\b`	Where "man" is at the end of a word
`\bman`	Where "man" is at the start of a word
`me*an`	Where zero or more "e" letters occur between "m" and "a"
`.a.`	Where "a" is surrounded by any one character (including space)
`\sa\s`	Where "a" is surrounded by a space on both sides
`z`	Where a "z" occurs (none in the default string)

In the scripts for Listing 27-3, if the `string.match()` method returns `null`, you are informed politely, and the count field is emptied.

Related Items: `RegExp` object (Chapter 42 on the CD-ROM).

string.replace(*regExpression, replaceString*)

Returns: Changed string.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

Regular expressions are commonly used to perform search-and-replace operations. JavaScript's *string*`.replace()` method provides a simple framework in which to perform this kind of operation on any string.

Searching and replacing requires three components. The first is the main string that is the target of the operation. Second is the regular expression to search for. And third is the string to replace each instance of the text found by the operation. For the *string*.replace() method, the main string is the string value or object referenced to the left of the period. This string can also be a literal string (that is, text surrounded by quotes). The regular expression to search for is the first parameter, while the replacement string is the second parameter.

The regular expression definition determines whether the replacement is of just the first match encountered in the main string or all matches in the string. If you add the g parameter to the end of the regular expression, then one invocation of the replace() method performs global search-and-replace through the entire main string.

As long as you know how to generate a regular expression, you don't have to be a whiz to use the *string*.replace() method to perform simple replacement operations. But using regular expressions can make the operation more powerful. Consider these soliloquy lines by Hamlet:

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

If you wanted to replace both instances of "be" with "exist," you can do it in this case by specifying

```
var regexp = /be/g;
soliloquy.replace(regexp, "exist");
```

But you can't always be assured that the letters "b" and "e" will be standing alone as a word. What happens if the main string contains the word "being" or "saber"? The above example replaces the "be" letters in them as well.

The regular expression help comes from the special characters to better define what to search for. In the example here, the search is for the word "be." Therefore, the regular expression surrounds the search text with word boundaries (the \b special character), as in

```
var regexp = /\bbe\b/g;
soliloquy.replace(regexp, "exist");
```

This syntax also takes care of the fact that the first two "be" words are followed by punctuation, rather than a space, as you may expect for a freestanding word. For more about regular expression syntax, see Chapter 42 on the CD-ROM.

Example

The page in Listing 27-4 lets you practice with the *string*.replace() and *string*.search() methods and regular expressions in a friendly environment. The source text is a five-line excerpt from *Hamlet*. You can enter the regular expression to search for, and the replacement text as well. Note that the script completes the job of creating the regular expression object, so that you can focus on the other special characters used to define the matching string. All replacement activities act globally, because the g parameter is automatically appended to any expression you enter.

Default values in the fields replace the contraction 'tis with "it is" after you click the "Execute replace()" button (see Figure 27-2). Notice that the backslash character in front of the apostrophe of 'tis (in the string assembled in mainString) makes the apostophe a non-word boundary, and thus allows the \B't regular expression to find a match there. As described in the section on the *string*.search() method, the button connected to that method returns the offset character number of the matching string (or -1 if no match occurs).

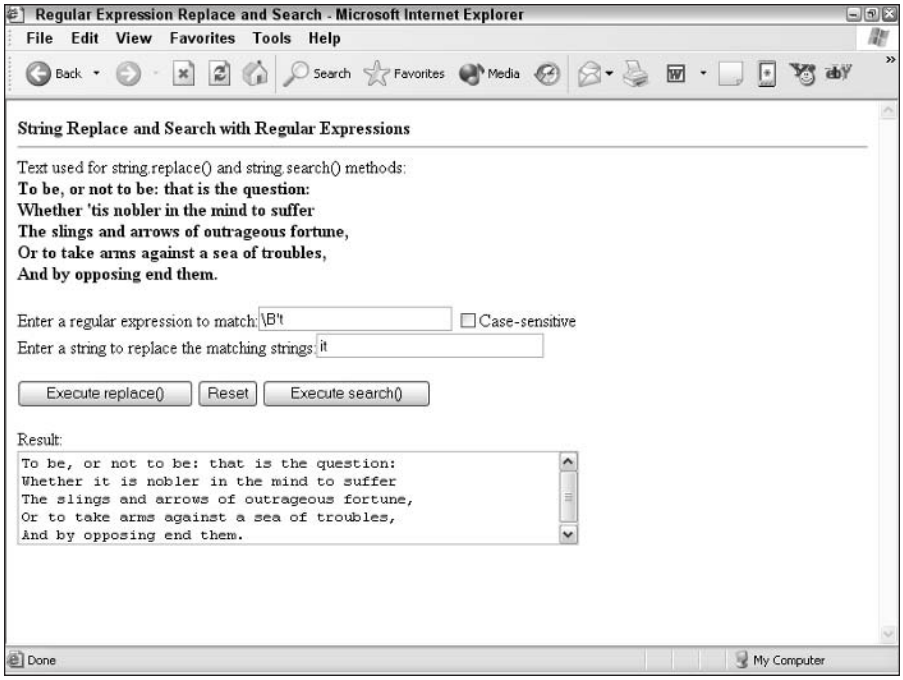


Figure 27-2: Using the default replacement regular expression.

You could modify the listing so that it actually replaces text in the HTML paragraph for IE4+ and W3C browsers. The steps include wrapping the paragraph in its own element (for example, a `span`) and invoking the `replace()` method on the `innerHTML` of that element. Assign the results to the `innerHTML` property of that element to complete the job.

Listing 27-4: **Lab for string.replace() and string.search()**

```
<html>
   <head>
      <title>Regular Expression Replace and Search</title>
      <script type="text/javascript">
      var mainString = "To be, or not to be: that is the question:\n";
      mainString += "Whether \'tis nobler in the mind to suffer\n";
      mainString += "The slings and arrows of outrageous fortune,\n";
      mainString += "Or to take arms against a sea of troubles,\n";
      mainString += "And by opposing end them.";

      function doReplace(form) {
         var replaceStr = form.replaceEntry.value;
         var delim = (form.caseSens.checked) ? "/g" : "/gi";
         var regexp = eval("/" + form.regexp.value + delim);
         form.result.value = mainString.replace(regexp, replaceStr);
      }
      function doSearch(form) {
         var replaceStr = form.replaceEntry.value;
         var delim = (form.caseSens.checked) ? "/g" : "/gi";
```

```
            var regexp = eval("/" + form.regexp.value + delim);
            form.result.value = mainString.search(regexp);
         }
         </script>
      </head>
      <body>
         <b>String Replace and Search with Regular Expressions</b>
         <hr />
         Text used for string.replace() and string.search() methods:<br />
         <b>To be, or not to be: that is the question:<br />
         Whether 'tis nobler in the mind to suffer<br />
         The slings and arrows of outrageous fortune,<br />
         Or to take arms against a sea of troubles,<br />
         And by opposing end them.</b>
         <form>
            Enter a regular expression to match:<input type="text" name="regexp"
            size="25" value="\B't" /> <input type="checkbox"
            name="caseSens" />Case-sensitive<br />
            Enter a string to replace the matching strings:<input type="text"
            name="replaceEntry" size="30" value="it " />
            <p><input type="button" value="Execute replace()"
               onclick="doReplace(this.form)" /> <input type="reset" /> <input
               type="button" value="Execute search()"
               onclick="doSearch(this.form)" /></p>
            <p>Result:<br />
               <textarea name="result" cols="60" rows="5" wrap="virtual">
               </textarea></p>
         </form>
      </body>
   </html>
```

Related Items: `string.match()` method; `RegExp` object.

string.search(*regExpression*)

Returns: Offset integer.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The results of the *string*.search() method may remind you of the *string*.indexOf() method. In both cases, the returned value is the character number where the matching string first appears in the main string, or `-1` if no match occurs. The big difference, of course, is that the matching string for *string*.search() is a regular expression.

Example

Listing 27-4, for the *string*.replace() method, also provides a laboratory to experiment with the *string*.search() method.

Related Items: `string.match()` method; `RegExp` object.

string.slice(*startIndex* [, *endIndex*])

Returns: String.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The *string*.slice() method resembles the method *string*.substring() in that both let you extract a portion of one string and create a new string as a result (without modifying the

original string). A helpful improvement in *string*.slice(), however, is that specifying an ending index value relative to the end of the main string is easier.

Using *string*.substring() to extract a substring that ends before the end of the string requires machinations, such as the following:

```
string.substring(4, (string.length-2))
```

Instead, you can assign a negative number to the second parameter of *string*.slice() to indicate an offset from the end of the string:

```
string.slice(4, -2)
```

The second parameter is optional. If you omit the second parameter, the returned value is a string from the starting offset to the end of the main string.

Example

With Listing 27-5, you can try several combinations of parameters with the *string*.slice() method (see Figure 27-3). A base string is provided (along with character measurements). Select from the different choices available for parameters and study the outcome of the slice.

Listing 27-5: **Slicing a String**

```
<html>
   <head>
      <title>String Slicing and Dicing, Part I</title>
      <script type="text/javascript">
      var mainString = "Electroencephalograph";
      function showResults() {
         var form = document.forms[0];
         var param1 = parseInt(
            form.param1.options[form.param1.selectedIndex].value);
         var param2 = parseInt(
            form.param2.options[form.param2.selectedIndex].value);
         if (!param2) {
            form.result1.value = mainString.slice(param1);
         } else {
            form.result1.value = mainString.slice(param1, param2);
         }
      }
      </script>
   </head>
   <body onload="showResults()">
      <b>String slice() Method</b>
      <hr />
      Text used for the methods:<br />
      <font size="+1"><tt><b>Electroencephalograph<br />
      ----5----5----5----5-</b></tt></font>
      <form>
         <table>
            <tr>
               <th>String Method</th>
               <th>Method Parameters</th>
               <th>Results</th>
            </tr>
            <tr>
               <td>string.slice()</td>
```

```
            <td rowspan="3" valign="middle">
            ( <select name="param1" onchange="showResults()">
               <option value="0">0</option>
               <option value="1">1</option>
               <option value="2">2</option>
               <option value="3">3</option>
               <option value="5">5</option>
            </select>, <select name="param2" onchange="showResults()">
               <option>(None)</option>
               <option value="5">5</option>
               <option value="10">10</option>
               <option value="-1">-1</option>
               <option value="-5">-5</option>
               <option value="-10">-10</option>
            </select> )
            </td>
            <td><input type="text" name="result1" size="25" /></td>
         </tr>
      </table>
   </form>
</body>
</html>
```

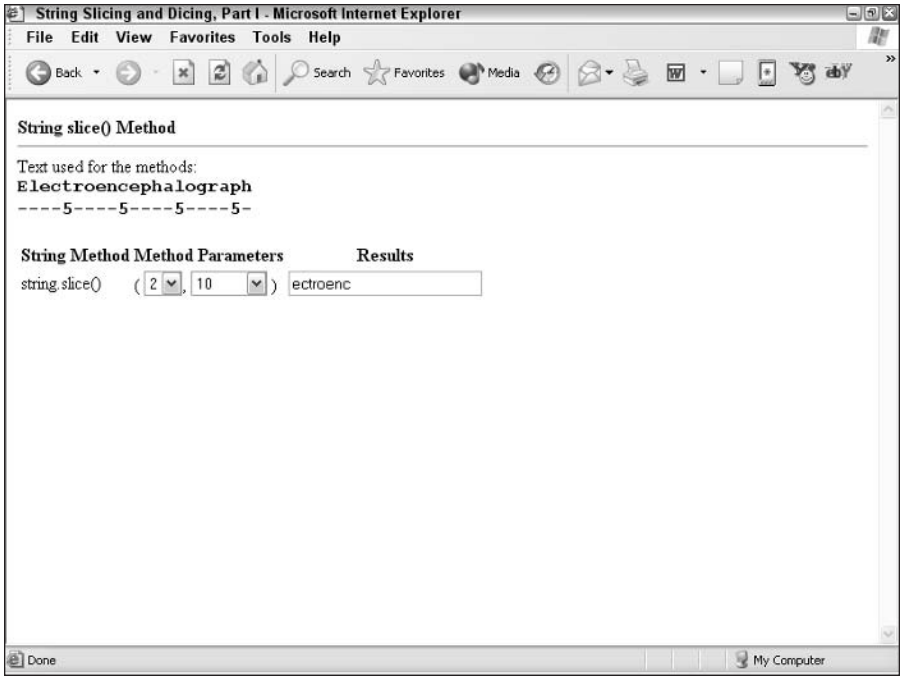


Figure 27-3: Lab for exploring the string.slice() method.

Related Items: `string.substr()`, `string.substring()` methods.

***stringObject*.split()**

string.split("*delimiterCharacter*" [, *limitInteger*])

Returns: Array of delimited items.
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

The `split()` method is the functional opposite of the *`array`*`.join()` method (see Chapter 30). From the string object point of view, JavaScript splits a long string into pieces delimited by a specific character and then creates a dense array with those pieces. You do not need to initialize the array via the `new Array()` constructor. Given the powers of array object methods, such as *`array`*`.sort()`, you may want to convert a series of string items to an array to take advantage of those powers. Also, if your goal is to divide a string into an array of single characters, you can still use the `split()` method, but specify an empty string as a parameter. For NN3 and IE4, only the first parameter is observed.

In NN4+, IE4+, and W3C browsers, you can use a regular expression object for the first parameter, enhancing the powers of finding delimiters in strings. For example, consider the following string:

```
var nameList = "1.Fred,2.Jane,3.Steve";
```

To convert that string into a three-element array of only the names takes a lot of parsing without regular expressions before you can even use *`string`*`.split()`. However, with a regular expression as a parameter,

```
var regexp = /,*\d.\b/;
var newArray = nameList.split(regexp);
    // result = an array "Fred", "Jane", "Steve"
```

the new array entries hold only the names and not the leading numbers or periods. A second addition is an optional second parameter. This integer value allows you to specify a limit to the number of array elements generated by the method.

Example

Use The Evaluator (Chapter 13) to see how the `string.split()` method works. Begin by assigning a comma-delimited string to a variable:

```
a = "Anderson,Smith,Johnson,Washington"
```

Now split the string at comma positions so that the string pieces become items in an array, saved as `b`:

```
b = a.split(",")
```

To prove that the array contains four items, inspect the array's `length` property:

```
b.length    // result: 4
```

Related Items: `array.join()` method.

string.substr(*start* [, *length*])

Returns: String.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The *`string`*`.substr()` method offers a variation of the *`string`*`.substring()` method that has been in the language since the beginning. The distinction is that the *`string`*`.substr()` method's parameters specify the starting index and a number of characters to be included from that start point. In contrast, the *`string`*`.substring()` method parameters specify index points for the start and end characters within the main string.

As with all string methods requiring an index value, the *string*.substr() first parameter is zero-based. If you do not specify a second parameter, the returned substring starts at the indexed point and extends to the end of the string. A second parameter value that exceeds the end point of the string means that the method returns a substring to the end of the string.

Even though this method is newer than its partner, it is not part of the ECMA standard as of Edition 3 of the language spec. But because the method is so widely used, the standard does acknowledge it so that other scripting contexts can implement the method consistent with browser practice.

Example

Listing 27-6 lets you experiment with a variety of values to see how the *string*.substr() method works.

Listing 27-6: **Reading a Portion of a String**

```
<html>
   <head>
      <title>String Slicing and Dicing, Part II</title>
      <script type="text/javascript">
      var mainString = "Electroencephalograph";
      function showResults() {
         var form = document.forms[0];
         var param1 = parseInt(
            form.param1.options[form.param1.selectedIndex].value);
         var param2 = parseInt(
            form.param2.options[form.param2.selectedIndex].value);
         if (!param2) {
            form.result1.value = mainString.substr(param1);
         } else {
            form.result1.value = mainString.substr(param1, param2);
         }
      }
      </script>
   </head>
   <body onload="showResults()">
      <b>String substr() Method</b>
      <hr />
      Text used for the methods:<br />
      <font size="+1"><tt><b>Electroencephalograph<br />
      ----5----5----5----5-</b></tt></font>
      <form>
         <table>
            <tr>
               <th>String Method</th>
               <th>Method Parameters</th>
               <th>Results</th>
            </tr>
            <tr>
               <td>string.substr()</td>
               <td rowspan="3" valign="middle">
               ( <select name="param1" onchange="showResults()">
                  <option value="0">0</option>
                  <option value="1">1</option>
```

Continued

Listing 27-6 *(continued)*

```
                <option value="2">2</option>
                <option value="3">3</option>
                <option value="5">5</option>
             </select>, <select name="param2" onchange="showResults()">
                <option>(None)</option>
                <option value="5">5</option>
                <option value="10">10</option>
                <option value="20">20</option>
             </select> )
             </td>
             <td><input type="text" name="result1" size="25" /></td>
          </tr>
       </table>
    </form>
  </body>
</html>
```

Related Items: `string.substring()` method.

string.substring(*indexA, indexB*)

Returns: String of characters between index values *indexA* and *indexB*.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The *string*.substring() method enables your scripts to extract a copy of a contiguous range of characters from any string. The parameters to this method are the starting and ending index values (first character of the string object is index value 0) of the main string from which the excerpt should be taken. An important item to note is that the excerpt goes up to, but does not include, the character pointed to by the higher index value.

It makes no difference which index value in the parameters is larger than the other: The method starts the excerpt from the lowest value and continues to (but does not include) the highest value. If both index values are the same, the method returns an empty string; and if you omit the second parameter, the end of the string is assumed to be the endpoint.

Example

Listing 27-7 lets you experiment with a variety of values to see how the *string*.substring() method works.

Listing 27-7: **Reading a Portion of a String**

```
<html>
   <head>
      <title>String Slicing and Dicing, Part III</title>
      <script type="text/javascript">
      var mainString = "Electroencephalograph";
      function showResults() {
         var form = document.forms[0];
         var param1 = parseInt(
            form.param1.options[form.param1.selectedIndex].value);
         var param2 = parseInt(
```

```
            form.param2.options[form.param2.selectedIndex].value);
         if (!param2) {
            form.result1.value = mainString.substring(param1);
         } else {
            form.result1.value = mainString.substring(param1, param2);
         }
      }
      </script>
   </head>
   <body onload="showResults()">
      <b>String substr() Method</b>
      <hr />
      Text used for the methods:<br />
      <font size="+1"><tt><b>Electroencephalograph<br />
      ----5----5----5----5-</b></tt></font>
      <form>
         <table>
            <tr>
               <th>String Method</th>
               <th>Method Parameters</th>
               <th>Results</th>
            </tr>
            <tr>
               <td>string.substring()</td>
               <td>( <select name="param1" onchange="showResults()">
                  <option value="0">0</option>
                  <option value="1">1</option>
                  <option value="2">2</option>
                  <option value="3">3</option>
                  <option value="5">5</option>
               </select>, <select name="param2" onchange="showResults()">
                  <option>(None)</option>
                  <option value="3">3</option>
                  <option value="5">5</option>
                  <option value="10">10</option>
               </select> )
               </td>
               <td><input type="text" name="result1" size="25" /></td>
            </tr>
         </table>
      </form>
   </body>
</html>
```

Related Items: `string.substr()`, `string.slice()` methods.

string.toLocaleLowerCase()
string.toLocaleUpperCase()

Returns: String.
Compatibility: WinIE5.5+, MacIE-, NN6+, Moz1+, Safari-

These two methods are variations on the standard methods for changing the case of a string. They take into account some language systems whose cases for a particular character don't necessarily map to the Latin alphabet character mappings.

Related Items: `string.toLowerCase()`, `string.toUpperCase()` methods.

string.toLowerCase()
string.toUpperCase()

Returns: The string in all lower- or uppercase, depending on which method you invoke.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

A great deal of what takes place on the Internet (and in JavaScript) is case-sensitive. URLs on some servers, for instance, are case-sensitive for directory names and filenames. These two methods, the simplest of the string methods, return a copy of a string converted to either all lowercase or all uppercase. Any mixed-case strings get converted to a uniform case. If you want to compare user input from a field against some coded string without worrying about matching case, you can convert both strings to the same case for the comparison.

Example

You can use the `toLowerCase()` and `toUpperCase()` methods on literal strings, as follows:

```
var newString = "HTTP://www.Netscape.COM".toLowerCase();
    // result = "http://www.netscape.com"
```

The methods are also helpful in comparing strings when case is not important, as follows:

```
if (guess.toUpperCase() == answer.toUpperCase()) {...}
    // comparing strings without case sensitivity
```

Related Items: `string.toLocaleLowerCase()`, `string.toLocaleUpperCase()` methods.

string.toString()
string.valueOf()

Returns: String value.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

Both of these methods return string values (as opposed to full-fledged string objects). If you have created a string object via the `new String()` constructor, the type of that item is object. Therefore, if you want to examine more precisely what kind of value is held by the object, you can use the `valueOf()` method to get the value and then examine it via the `typeof` operator. The `toString()` method is present for this object primarily because a string object inherits the method from the root object of JavaScript.

Example

Use The Evaluator (Chapter 13) to test the `valueOf()` method. Enter the following statements into the top text box and examine the values that appear in the Results field:

```
a = new String("hello")
typeof a
b = a.valueOf()
typeof b
```

Because all other JavaScript core objects also have the `valueOf()` method, you can build generic functions that receive a variety of object types as parameters, and the script can branch its code based on the type of value that is stored in the object.

Related Items: `typeof` operator (Chapter 32).

String Utility Functions

Figuring out how to apply the various string object methods to a string manipulation challenge is not always an easy task, especially if you need backward compatibility with older scriptable browsers. I also find it difficult to anticipate every possible way you may need to massage strings in your scripts. But to help you get started, Listing 27-8 contains a fully backward-compatible library of string functions for inserting, deleting, and replacing chunks of text in a string. If your audience uses browsers capable of including external `.js` library files, that would be an excellent way to make these functions available to your scripts.

Listing 27-8: **Utility String Handlers**

```
// extract front part of string prior to searchString
function getFront(mainStr,searchStr){
   foundOffset = mainStr.indexOf(searchStr);
   if (foundOffset == -1) {
      return null;
   }
   return mainStr.substring(0,foundOffset);
}

// extract back end of string after searchString
function getEnd(mainStr,searchStr) {
   foundOffset = mainStr.indexOf(searchStr);
   if (foundOffset == -1) {
      return null;
   }
   return mainStr.substring(foundOffset+searchStr.length,mainStr.length);
}

// insert insertString immediately before searchString
function insertString(mainStr,searchStr,insertStr) {
   var front = getFront(mainStr,searchStr);
   var end = getEnd(mainStr,searchStr);
   if (front != null && end != null) {
      return front + insertStr + searchStr + end;
   }
   return null;
}

// remove deleteString
function deleteString(mainStr,deleteStr) {
   return replaceString(mainStr,deleteStr,"");
}

// replace searchString with replaceString
function replaceString(mainStr,searchStr,replaceStr) {
   var front = getFront(mainStr,searchStr);
   var end = getEnd(mainStr,searchStr);
   if (front != null && end != null) {
      return front + replaceStr + end;
   }
   return null;
}
```

The first two functions extract the front or end components of strings as needed for some of the other functions in this suite. The final three functions are the core of these string-handling functions. If you plan to use these functions in your scripts, be sure to notice the dependence that some functions have on others. Including all five functions as a group ensures that they work as designed.

A modern alternative to Listing 27-8 utilizes a combination of string and array methods to perform a global replace operation in a one-statement function:

```
function replaceString(mainStr, searchStr, replaceStr) {
   return mainStr.split(searchStr).join(replaceStr);
}
```

Going one step further, you can create a custom method to use with all string values or objects in your scripts. Simply let the following statement execute as the page loads:

```
String.prototype.replaceString = function(mainStr, searchStr, replaceStr) {
   return mainStr.split(searchStr).join(replaceStr);
}
```

Then invoke this method of any string value in other scripts on the page, as in:

```
myString = myString.replaceString(" CD ", " MP3 ");
```

Formatting methods

Now we come to the other group of string object methods, which ease the process of creating the numerous string display characteristics when you use JavaScript to assemble HTML code. The following is a list of these methods:

```
string.anchor("anchorName")          string.link(locationOrURL)
string.blink()                       string.big()
string.bold()                        string.small()
string.fixed()                       string.strike()
string.fontcolor(colorValue)         string.sub()
string.fontsize(integer1to7)         string.sup()
string.italics()
```

First examine the methods that don't require any parameters. You probably see a pattern: All of these methods are font-style attributes that have settings of on or off. To turn on these attributes in an HTML document, you surround the text in the appropriate tag pairs, such as `<b>...</b>` for boldface text. These methods take the string object, attach those tags, and return the resulting text, which is ready to be put into any HTML that your scripts are building. Therefore, the expression

```
"Good morning!".bold()
```

evaluates to

```
<b>Good morning!</b>
```

Of course, nothing is preventing you from building your HTML by embedding real tags instead of by calling the string methods. The choice is up to you. One advantage to the string methods is that they never forget the ending tag of a tag pair. Listing 27-9 shows an example of incorporating a few simple string methods in a string variable that is eventually written to

the page as it loads. Internet Explorer does not support the <blink> tag and therefore ignores the *string*.blink() method.

Listing 27-9: **Using Simple String Methods**

```
<html>
   <head>
      <title>HTML by JavaScript</title>
   </head>
   <body>
      <script type="text/javascript">
      var page = "";
      page += "JavaScript can create HTML on the fly.<P>Numerous string object
         methods facilitate creating text that is " +
         "boldfaced".bold() + ", " + "italicized".italics() +
         ", or even the terribly annoying " + "blinking text".blink() + ".";
      document.write(page);
      </script>
   </body>
</html>
```

Of the remaining string methods, two more (*string*.fontsize() and *string*.fontcolor()) also affect the font characteristics of strings displayed in the HTML page. The parameters for these items are pretty straightforward — an integer between 1 and 7 corresponding to the seven browser font sizes and a color value (as either a hexadecimal triplet or color constant name) for the designated text. Listing 27-10 adds a line of text to the string of Listing 27-9. This line of text not only adjusts the font size of some parts of the string but also nests multiple attributes inside one another to set the color of one word in a large-font-size string. Because these string methods do not change the content of the string, you can safely nest methods here.

Listing 27-10: **Nested String Methods**

```
<html>
   <head>
      <title>HTML by JavaScript</title>
   </head>
   <body>
      <script type="text/javascript">
      var page = "";
      page += "JavaScript can create HTML on the fly.<P>Numerous string object
         methods facilitate creating text that is " + "boldfaced".bold() +
         ", " + "italicized".italics() + ", or even the terribly annoying " +
         "blinking text".blink() + ".<br />";
      page += "We can make " + "some words big".fontsize(5) + " and some words
         both " + ("big and " + "colorful".fontcolor('coral')).fontsize(5) +
         " at the same time.";
      document.write(page);
      </script>
   </body>
</html>
```

The final two string methods let you create an anchor and a link out of a string. The *string*.anchor() method uses its parameter to create a name for the anchor. Thus, the following expression

```
"Table of Contents".anchor("toc")
```

evaluates to

```
<a name="toc">Table of Contents</a>
```

In a similar fashion, the *string*.link() method expects a valid location or URL as its parameter, creating a genuine HTML link out of the string:

```
"Back to Home".link("index.html")
```

This evaluates to the following:

```
<a href="index.html">Back to Home</a>
```

Again, the choice of whether you use string methods to build HTML anchors and links over assembling the actual HTML is up to you. The methods may be a bit easier to work with if the values for the string and the parameters are variables whose content may change based on user input elsewhere in your Web site.

URL String Encoding and Decoding

When browsers and servers communicate, some non-alphanumeric characters that we take for granted (such as a space) cannot make the journey in their native form. Only a narrower set of letters, numbers, and punctuation is allowed. To accommodate the rest, the characters must be encoded with a special symbol (%) and their hexadecimal ASCII values. For example, the space character is hex 20 (ASCII decimal 32). When encoded, it looks like %20. You may have seen this symbol in browser history lists or URLs.

JavaScript includes two functions, escape() and unescape(), that offer instant conversion of whole strings. To convert a plain string to one with these escape codes, use the escape function, as in

```
escape("Howdy Pardner"); // result = "Howdy%20Pardner"
```

The unescape() function converts the escape codes into human-readable form. Both of these functions and some newer, more robust versions for recent browsers are covered in Chapter 34.

✦ ✦ ✦

CHAPTER 28

The Math, Number, and Boolean Objects

In This Chapter

Advanced math operations

Number base conversions

Working with integers and floating-point numbers

The introduction to data types and values in Chapter 6's tutorial scratched the surface of JavaScript's numeric and Boolean powers. In this chapter, you look more closely at JavaScript's way of working with numbers and Boolean data.

Math often frightens away budding programmers; but as you've seen so far in this book, you don't really have to be a math genius to program in JavaScript. The powers described in this chapter are here when you need them — if you need them. So if math is not your strong suit, don't freak out over the terminology here.

An important point to remember about the objects described in this chapter is that (like string values and string objects) numbers and Booleans are both values and objects. Fortunately for script writers, the differentiation is rarely, if ever, a factor unless you get into some very sophisticated programming. To those who actually write the JavaScript interpreters inside the browsers we use, the distinctions are vital.

For most scripters, the information about numeric data types and conversions as well as the `Math` object are important to know. I present other details in this chapter about the number and Boolean objects primarily for completeness because their direct powers are almost never used in day-to-day scripting of Web applications.

Numbers in JavaScript

More powerful programming languages have many different kinds of numbers, each related to the amount of memory it occupies in the computer. Managing all these different types may be fun for some, but it gets in the way of quick scripting. A JavaScript number has only two possibilities. It can be an integer or a floating-point value. An *integer* is any whole number within a humongous range that does not have any fractional part. Integers never contain a decimal point in their representation. *Floating-point numbers* in JavaScript spread across the same range, but they are represented with a decimal point and some fractional value. If you are an experienced programmer,

refer to the discussion about the `number` object later in this chapter to see how the JavaScript number type lines up with numeric data types you use in other programming environments.

Integers and floating-point numbers

Deep inside a computer, the microprocessor has an easier time performing math on integer values as compared to any number with a decimal value tacked on it, which requires the microprocessor to go through extra work to add even two such floating-point numbers. We, as scripters, are unfortunately saddled with this historical baggage and must be conscious of the type of number used in certain calculations.

Most internal values generated by JavaScript, such as index values and `length` properties, consist of integers. Floating-point numbers usually come into play as the result of the division of numeric values, special values such as pi, and human-entered values such as dollars and cents. Fortunately, JavaScript is forgiving if you try to perform math operations on mixed numeric data types. Notice how the following examples resolve to the appropriate data type:

```
3 + 4 = 7 // integer result
3 + 4.1 = 7.1 // floating-point result
3.9 + 4.1 = 8 // integer result
```

Of the three examples, perhaps only the last result is unexpected. When two floating-point numbers yield a whole number, the result is rendered as an integer.

When dealing with floating-point numbers, be aware that not all browser versions return the precise same value down to the last digit to the right of the decimal. For example, the following table shows the result of 8/9 as calculated by numerous scriptable browsers (all in Windows) and converted for string display:

Navigator 2	0.88888888888888884
Navigator 3 & 4	.8888888888888888
Navigator 6+/Mozilla 1+/Safari 1+	0.8888888888888888
Internet Explorer 3	0.888888888888889
Internet Explorer 4+	0.8888888888888888

Clearly, from this display, you don't want to use floating-point math in JavaScript browsers to plan space flight trajectories. For everyday math, however, you need to be cognizant of floating-point errors that accrue in PC arithmetic.

In Navigator, JavaScript relies on the operating system's floating-point math for its own math. Operating systems that offer accuracy to as many places to the right of the decimal as JavaScript displays are exceedingly rare. As you can detect from the preceding table, the modern versions of browsers from Netscape and Microsoft agree about how many digits to display and how to perform internal rounding for this display. That's good for the math, but not particularly helpful when you need to display numbers in a specific format.

Until you get to IE5.5, NN6, and W3C-compatible browsers, JavaScript does not offer built-in facilities for formatting the results of floating-point arithmetic. (For the newer browsers, see the `Number` object later in this chapter for formatting methods.) Listing 28-1 demonstrates a generic formatting routine for positive values, plus a specific call that turns a value into a dollar value. Remove the comments and the routine is fairly compact.

Listing 28-1: **A Generic Number-Formatting Routine**

```
<html>
   <head>
      <title>Number Formatting</title>
      <script type="text/javascript">
      // generic positive number decimal formatting function
      function format(expr, decplaces) {
         // raise incoming value by power of 10 times the
         // number of decimal places; round to an integer; convert to string
         var str = "" + Math.round(eval(expr) * Math.pow(10,decplaces));
         // pad small value strings with zeros to the left of rounded number
         while (str.length <= decplaces) {
            str = "0" + str;
         }
         // establish location of decimal point
         var decpoint = str.length - decplaces;
         // assemble final result from: (a) the string up to the position of
         // the decimal point; (b) the decimal point; and (c) the balance
         // of the string. Return finished product.
         return str.substring(0,decpoint) + "." +
            str.substring(decpoint,str.length);
      }
      // turn incoming expression into a dollar value
      function dollarize(expr) {
         return "$" + format(expr,2);
      }
      </script>
   </head>
   <body>
      <h1>How to Make Money</h1>
      <form>
         Enter a positive floating point value or arithmetic expression to be
         converted to a currency format:
         <p><input type="text" name="entry" value="1/3" /> <input type="button"
            value="&gt;Dollars and Cents&gt;"
            onclick="this.form.result.value=dollarize(this.form.entry.value)"
            />
            <input type="text" name="result" /></p>
      </form>
   </body>
</html>
```

This routine may seem like a great deal of work, but it's essential if your application relies on floating-point values and specific formatting for all browsers.

You can also enter floating-point numbers with exponents. An exponent is signified by the letter "e" (upper- or lowercase), followed by a sign (+ or –) and the exponent value. Here are examples of floating-point values expressed as exponents:

```
1e6 // 1,000,000 (the "+" symbol is optional on positive exponents)
1e-4 // 0.0001 (plus some error further to the right of the decimal)
-4e-3 // -0.004
```

For values between 1e-5 and 1e15, JavaScript renders numbers without exponents (although you can force a number to display in exponential notation in IE5.5+, NN6+, and W3C browsers). All other values outside these boundaries return with exponential notation in all browsers.

Hexadecimal and octal integers

JavaScript enables you to work with values in decimal (base-10), hexadecimal (base-16), and octal (base-8) formats. You have only a few rules to follow when dealing with any of these values.

Decimal values cannot begin with a leading 0. Therefore, if your page asks users to enter decimal values that begin with a 0, your script must strip those zeros from the input string or use the number parsing global functions (described in the next section) before performing any math on the values.

Hexadecimal integer values are expressed with a leading `0x` or `0X`. (That's a zero, not the letter "o.") The A through F values can appear in upper- or lowercase, as you prefer. Here are some hex values:

```
0X2B
0X1a
0xcc
```

Don't confuse the hex values used in arithmetic with the hexadecimal values used in color property specifications for Web documents. Those values are expressed in a special *hexadecimal triplet* format, which begins with a crosshatch symbol followed by the three hex values bunched together (such as `#c0c0c0`).

Octal values are represented by a leading 0 followed by any digits between 0 and 7. Octal values consist only of integers.

You are free to mix and match base values in arithmetic expressions, but JavaScript renders all results in decimal form. For conversions to other number bases, you have to employ a user-defined function in your script. Listing 28-2, for example, is a function that converts any decimal value from 0 to 255 into a JavaScript hexadecimal value.

Listing 28-2: **Decimal-to-Hexadecimal Converter Function**

```
function toHex(dec) {
   hexChars = "0123456789ABCDEF";
   if (dec > 255) {
      return null;
   }
   var i = dec % 16;
   var j = (dec - i) / 16;
   result = "0X";
   result += hexChars.charAt(j);
   result += hexChars.charAt(i);
   return result;
}
```

The `toHex()` conversion function assumes that the value passed to the function is a decimal integer. If you simply need a hexadecimal representation of a number in string format, see the `toString()` method in Chapter 34.

Converting strings to numbers

What is missing so far from this discussion is a way to convert a number represented as a string to a number with which the JavaScript arithmetic operators can work. Before you get too concerned about this, be aware that most JavaScript operators and math methods gladly accept string representations of numbers and handle them without complaint. You will run into data type incompatibilities most frequently when trying to accomplish addition with the + operator (which is also the string concatenation operator). Also know that if you perform math operations on values retrieved from form text boxes, those object `value` properties are strings. Therefore, in many cases, you need to convert those values to values of the number type for math operations.

Conversion to numbers requires one of two JavaScript functions:

```
parseInt(string [,radix])
parseFloat(string  [,radix])
```

These functions are inspired by the Java language. The term *parsing* has many implied meanings in programming. One meaning is the same as *extracting*. The `parseInt()` function returns whatever integer value it can extract from the string passed to it; the `parseFloat()` function returns the floating-point number that can be extracted from the string. Here are some examples and their resulting values:

```
parseInt("42")        // result = 42
parseInt("42.33")     // result = 42
parseFloat("42.33")   // result = 42.33
parseFloat("42")      // result = 42
parseFloat("fred")    // result = NaN
```

Because the `parseFloat()` function can also work with an integer and return an integer value, you may prefer using this function in scripts that have to deal with either kind of number, depending on the string entered into a text field by a user.

An optional second parameter to both functions enables you to specify the base of the number represented by the string. This comes in handy particularly when you need a decimal number from a string that starts with one or more zeros. Normally, the leading zero indicates an octal value. But if you force the conversion to recognize the string value as a decimal, it is converted the way you expect:

```
parseInt("010")       // result = 8
parseInt("010",10)    // result = 10
parseInt("F2")        // result = NaN
parseInt("F2", 16)    // result = 242
```

Use these functions wherever you need the integer or floating-point value. For example:

```
var result = 3 + parseInt("3");    // result = 6
var ageVal = parseInt(document.forms[0].age.value);
```

The latter technique ensures that the string value of this property is converted to a number (although you should do more data validation — see Chapter 43 on the CD-ROM — before trying any math on a user-entered value).

Both the `parseInt()` and `parseFloat()` methods start working on the first character of a string and continue until there are no more numbers or decimal characters. That's why you can use them on strings — such as the one returned by the `navigator.appVersion` property (for example, `5.0 (Windows; en-US)`) — to obtain just the leading, numeric part of the string. If the string does not begin with an acceptable character, the methods return `NaN` (not a number).

Converting numbers to strings

If you attempt to pass a numeric data type value to many of the string methods discussed in Chapter 27, JavaScript complains. Therefore, you should convert any number to a string before you, for example, find out how many digits make up a number.

Several ways exist to force conversion from any numeric value to a string. The old-fashioned way is to precede the number with an empty string and the concatenation operator. For example, assume that a variable named `dollars` contains the integer value of `2500`. To use the string object's `length` property (discussed later in this chapter) to find out how many digits the number has, use this construction:

```
("" + dollars).length    // result = 4
```

The parentheses force JavaScript to evaluate the concatenation before attempting to extract the `length` property.

A more elegant way is to use the `toString()` method. Construct such statements as you do to invoke any object's method. For example, to convert the `dollars` variable value to a string, use this statement:

```
dollars.toString()    // result = "2500"
```

This method has one added power in NN3+ and IE4+: You can specify a number base for the string representation of the number. Called the *radix,* the base number is added as a parameter to the method name. Here is an example of creating a numeric value for conversion to its hexadecimal equivalent as a string:

```
var x = 30;
var y = x.toString(16);    // result = "1e"
```

Use a parameter of `2` for binary results and `8` for octal. The default is base `10`. Be careful not to confuse these conversions with true numeric conversions. You cannot use results from the `toString()` method as numeric operands in other statements.

Finally, in IE5.5+, NN6, and W3C browsers, three additional methods of the `Number` object — `toExponential()`, `toFixed()`, and `toPrecision()` — return string versions of numbers formatted according to the rules and parameters passed to the methods. I describe these in detail later in this chapter.

When a number isn't a number

In a couple of examples in the previous section, you probably noticed that the result of some operations was a value named `NaN`. That value is not a string but rather a special value that stands for Not a Number. For example, if you try to convert the string `"joe"` to an integer with `parseFloat()`, the function cannot possibly complete the operation. It reports back that the source string, when converted, is not a number.

When you design an application that requests user input or retrieves data from a server-side database, you cannot be guaranteed that a value you need to be numeric is, or can be converted to, a number. If that's the case, you need to see if the value is a number before performing some math operation on it. JavaScript provides a special global function, `isNaN()`, that enables you to test the "numberness" of a value. The function returns `true` if the value is not a number and `false` if it is a number. For example, you can examine a form field that should be a number:

```
var ageEntry = parseInt(document.forms[0].age.value);
if (isNaN(ageEntry)) {
   alert("Try entering your age again.");
}
```

Math Object

Whenever you need to perform math that is more demanding than simple arithmetic, look through the list of `Math` object methods for the solution.

Syntax

Accessing `Math` object properties and methods:

```
Math.property
Math.method(value [, value])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

About this object

In addition to the typical arithmetic operations (covered in detail in Chapter 32), JavaScript includes more advanced mathematical powers that you can access in a way that may seem odd to you if you have not programmed in true object-oriented environments before. Although most arithmetic takes place on the fly (such as `var result = 2 + 2`), the rest requires use of the JavaScript internal `Math` object (with a capital "M"). The `Math` object brings with it several properties (which behave like some other languages' constants) and many methods (which behave like some other languages' math functions).

The way you use the `Math` object in statements is the same way you use any JavaScript object: You create a reference beginning with the `Math` object's name, a period, and the name of the property or method you need:

```
Math.property | method([parameter]. . . [,parameter])
```

Property references return the built-in values (things such as pi). Method references require one or more values to be sent as parameters of the method. Every method returns a result.

Properties

JavaScript `Math` object properties represent a number of valuable constant values in math. Table 28-1 shows you those methods and their values as displayed to 16 decimal places.

Table 28-1: JavaScript Math Properties

Property	*Value*	*Description*
`Math.E`	`2.718281828459045091`	Euler's constant
`Math.LN2`	`0.6931471805599452862`	Natural log of 2
`Math.LN10`	`2.302585092994045901`	Natural log of 10
`Math.LOG2E`	`1.442695040888963387`	Log base-2 of E
`Math.LOG10E`	`0.4342944819032518167`	Log base-10 of E
`Math.PI`	`3.141592653589793116`	π
`Math.SQRT1_2`	`0.7071067811865475727`	Square root of 0.5
`Math.SQRT2`	`1.414213562373095145`	Square root of 2

Because these property expressions return their constant values, you use them in your regular arithmetic expressions. For example, to obtain the circumference of a circle whose diameter is in variable `d`, employ this statement:

```
circumference = d * Math.PI;
```

Perhaps the most common mistakes scripters make with these properties are failing to capitalize the `Math` object name and observing the case-sensitivity of property names.

Methods

Methods make up the balance of JavaScript `Math` object powers. With the exception of the `Math.random()` method, all `Math` object methods take one or more values as parameters. Typical trigonometric methods operate on the single values passed as parameters; others determine which of the numbers passed along are the highest or lowest of the group. The `Math.random()` method takes no parameters but returns a randomized, floating-point value between `0` and `1` (note that the method does not work on Windows or Macintosh versions of Navigator 2). Table 28-2 lists all the `Math` object methods with their syntax and descriptions of the values they return.

Table 28-2: Math Object Methods

Method Syntax	*Returns*
`Math.abs(val)`	Absolute value of ***val***
`Math.acos(val)`	Arc cosine (in radians) of ***val***
`Math.asin(val)`	Arc sine (in radians) of ***val***
`Math.atan(val)`	Arc tangent (in radians) of ***val***
`Math.atan2(val1, val2)`	Angle of polar coordinates ***x*** and ***y***
`Math.ceil(val)`	Next integer greater than or equal to ***val***
`Math.cos(val)`	Cosine of ***val***
`Math.exp(val)`	Euler's constant to the power of ***val***
`Math.floor(val)`	Next integer less than or equal to ***val***
`Math.log(val)`	Natural logarithm (base e) of ***val***

Method Syntax	*Returns*
`Math.max(val1, val2)`	The greater of ***val1*** or ***val2***
`Math.min(val1, val2)`	The lesser of ***val1*** or ***val2***
`Math.pow(val1, val2)`	***Val1*** to the ***val2*** power
`Math.random()`	Random number between 0 and 1
`Math.round(val)`	N+1 when ***val*** >= n.5; otherwise N
`Math.sin(val)`	Sine (in radians) of ***val***
`Math.sqrt(val)`	Square root of ***val***
`Math.tan(val)`	Tangent (in radians) of *val*

HTML is not exactly a graphic artist's dream environment, so using trig functions to obtain a series of values for HTML-generated charting is not a hot JavaScript prospect. Only with the advent of positionable elements have scripters been able to apply their knowledge of using these functions to define fancy trajectories for flying elements. For scripters who are not trained in programming, math is often a major stumbling block. But as you've seen so far, you can accomplish a great deal with JavaScript by using simple arithmetic and a little bit of logic — leaving the heavy-duty math for those who love it.

Creating random numbers

The `Math.random()` method returns a floating-point value between 0 and 1. If you design a script to act like a card game, you need random integers between 1 and 52; for dice, the range is 1 to 6 per die. To generate a random integer between zero and any top value, use the following formula:

```
Math.floor(Math.random() * n)
```

Here, *n* is the top number. To generate random numbers between a different range, use this formula:

```
Math.floor(Math.random() * (n - m + 1)) + m
```

Here, *m* is the lowest possible integer value of the range and *n* equals the top number of the range. For the dice game, the formula for each die is

```
newDieValue = Math.floor(Math.random() * 6) + 1;
```

Math object shortcut

In Chapter 31, you see details about a JavaScript construction that enables you to simplify the way you address multiple `Math` object properties and methods in statements. The trick is to use the `with` statement.

In a nutshell, the `with` statement tells JavaScript that the next group of statements (inside the braces) refers to a particular object. In the case of the `Math` object, the basic construction looks like this:

```
with (Math) {
   //statements
}
```

For all intervening statements, you can omit the specific references to the `Math` object. Compare the long reference way of calculating the area of a circle (with a radius of six units)

```
result = Math.pow(6,2) * Math.PI;
```

to the shortcut reference way:

```
with (Math) {
   result = pow(6,2) * PI;
}
```

Though the latter occupies more lines of code, the object references are shorter and more natural when reading the code. For a longer series of calculations involving `Math` object properties and methods, the `with` construction saves keystrokes and reduces the likelihood of a case-sensitive mistake with the object name in a reference. You can also include other full-object references within the `with` construction; JavaScript attempts to attach the object name only to those references lacking an object name. On the downside, the `with` construction is not particularly efficient in JavaScript because it must perform a lot of internal tracking in order to work.

Number Object

Properties	*Methods*
Constructor	toExponential()
MAX_VALUE	toFixed()
MIN_VALUE	toLocaleString()
NaN	toString()
NEGATIVE_INFINITY	toPrecision()
POSITIVE_INFINITY	valueOf()
Prototype	

Syntax

Creating a number object:

```
var val = new Number(number);
```

Accessing number and `Number` object properties and methods:

```
number.property | method([parameters])
Number.property | method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

About this object

The `Number` object is rarely used because (for the most part) JavaScript satisfies day-to-day numeric needs with a plain number value. But the `Number` object contains some information and power of value to serious programmers.

First on the docket are properties that define the ranges for numbers in the language. The largest number is 1.79E+308; the smallest number is 2.22E-308. Any number larger than the maximum is `POSITIVE_INFINITY`; any number smaller than the minimum is `NEGATIVE_INFINITY`. Rarely will you accidentally encounter these values.

More to the point of a JavaScript object, however, is the `prototype` property. Chapter 27 shows how to add a method to a `string` object's prototype such that every newly created object contains that method. The same goes for the `Number.prototype` property. If you have a need to add common functionality to every number object, this is where to do it. This prototype facility is unique to full-fledged number objects and does not apply to plain number values. For experienced programmers who care about such matters, JavaScript number objects and values are defined internally as `ieee` double-precision 64-bit values.

Properties

constructor

(See `string.constructor` in Chapter 27)

MAX_VALUE
MIN_VALUE
NEGATIVE_INFINITY
POSITIVE_INFINITY

Value: Number. Read-Only
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

The `Number.MAX_VALUE` and `Number.MIN_VALUE` properties belong to the static `Number` object. They represent constants for the largest and smallest possible positive numbers that JavaScript (and ECMAScript) can work with. Their actual values are $1.7976931348623157 \times 10^{308}$, and 5×10^{-324}, respectively.

A number that falls outside the range of allowable numbers is equal to the constant `Number.POSITIVE_INFINITY` or `Number.NEGATIVE_INFINITY`.

Example

Enter each of the four `Number` object expressions into the top text field of The Evaluator (Chapter 13) to see how the browser reports each value.

```
Number.MAX_VALUE
Number.MIN_VALUE
Number.NEGATIVE_INFINITY
Number.POSITIVE_INFINITY
```

Related Items: `NaN` property; `isNaN()` global function.

NaN

Value: NaN. Read-Only
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

The `NaN` property is a constant that JavaScript uses to report when a number-related function or method attempts to work on a value other than a number or the result is something other

than a number. You encounter the NaN value most commonly as the result of the parseInt() and parseFloat() functions whenever a string undergoing conversion to a number lacks a numeral as the first character. Use the isNaN() global function to see if a value is an NaN value.

Example

See the discussion of the isNaN() function in Chapter 34.

Related Item: isNaN() global function.

prototype

(See String.prototype in Chapter 27)

Methods

number.toExponential(*fractionDigits*)
number.toFixed(*fractionDigits*)
number.toPrecision(*precisionDigits*)

Returns: String.
Compatibility: WinIE5.5+, MacIE-, NN6+, Moz1+, Safari1+

A recent addition to the ECMA language — and thus to the JavaScript-enabled browsers — are three Number object methods that let scripts control the formatting of numbers for display as string text. Each method has a unique purpose, but they all return strings. You should perform all math operations as unformatted number objects because the values have the most precision. Only after you are ready to display the results should you use one of these methods to convert the number to a string for display as body text or assignment to a text field.

The toExponential() method forces a number to display in exponential notation, even if the number is in the range in which JavaScript normally uses standard notation. The parameter is an integer specifying how many digits to the right of the decimal should be returned. All digits to the right of the decimal are returned, even if they are zero. For example, if a variable contains the numeric value 345, applying toExponential(3) to that value yields 3.450e+2, which is JavaScript's exponential notation for 3.45×10^{2}.

Use the toFixed() method when you want to format a number with a specific number of digits to the right of the decimal. This is the method you use, for instance, to display the results of a financial calculation in units and hundredths of units (for example, dollars and cents). The parameter to the method is an integer indicating the number of digits to be displayed to the right of the decimal. If the number being formatted has more numbers to the right of the decimal than the number of digits specified by the parameter, the method rounds the rightmost visible digit — but only with respect to the unrounded value of the next digit. For example, the value 123.455 fixed to two digits to the right of the decimal is rounded up to 123.46. But if the starting value is 123.4549, the method ignores the 9 and sees that the 4 to the right of the 5 should be rounded down; therefore, the result is 123.45. Do not consider the toFixed() method to be an accurate rounder of numbers; however, it does a satisfactory job in most cases.

The final method is toPrecision(), which enables you to define how many total digits (including digits to the left and right of the decimal) to display of a number. In other words, you define the precision of a number. The following list demonstrates the results of several parameter values signifying a variety of precisions:

```
var num = 123.45
num.toPrecision(1)    // result = 1e+2
num.toPrecision(2)    // result = 1.2e+2
num.toPrecision(3)    // result = 123
num.toPrecision(4)    // result = 123.5
num.toPrecision(5)    // result = 123.45
num.toPrecision(6)    // result = 123.450
```

Notice that the same kind of rounding can occur with `toPrecision()` as it does for `toFixed()`.

Example

You can use The Evaluator (Chapter 13) to experiment with all three of these methods with a variety of parameter values. Before invoking any method, be sure to assign a numeric value to one of the built-in global variables in The Evaluator (`a` through `z`).

```
a = 10/3
a.toFixed(4)
"$" + a.toFixed(2)
```

None of these methods works with number literals (for example, `123.toExponential(2)` does not work).

Related Item: `Math` object.

number.toLocaleString()

Returns: String.
Compatibility: WinIE5.5+, MacIE5+, NN6+, Moz1+, Safari1+

The `number.toLocaleString()` method returns a string value version of the current number in a format that may vary according to a browser's locale settings. According to the ECMA Edition 3 standard, browsers have some leeway in determining exactly how the `toLocaleString()` method should return a string value that conforms with the language standard of the client system or browser.

Related Items: `number.toFixed()`, `number.toString()` methods.

number.toString([*radix*])

Returns: String.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The `number.toString()` method returns a string value version of the current number. The default radix parameter (`10`) converts the value to base-10 notation if the original number isn't already of that type. Or you can specify other number bases (for example, `2` for binary, `16` for hexadecimal) to convert the original number to the other base — as a string, not a number, for further calculation.

Example

Use The Evaluator (Chapter 13) to experiment with the `toString()` method. Assign the number `12` to the variable `a` and see how the number is converted to strings in a variety of number bases:

```
a = 12
a.toString()    // base 10
a.toString(2)
a.toString(16)
```

Related Item: `toLocaleString()` method.

number.valueOf()

(See `string.valueOf()` in Chapter 27)

Boolean Object

Properties	*Methods*
`constructor`	`toString()`
`prototype`	`valueOf()`

Syntax

Creating a `Boolean` object:

```
var val = new Boolean(BooleanValue);
```

Accessing `Boolean` object properties:

```
BooleanObject.property | method
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

About this object

You work with Boolean values a lot in JavaScript — especially as the result of conditional tests. Just as string values benefit from association with string objects and their properties and methods, so, too, do Boolean values receive aid from the `Boolean` object. For example, when you display a Boolean value in a text box, the `"true"` or `"false"` string is provided by the `Boolean` object's `toString()` method so you don't have to invoke it directly.

The only time you need to even think about a `Boolean` object is if you wish to attach some property or method to `Boolean` objects that you create with the new `Boolean()` constructor. Parameter values for the constructor include the string versions of the values, numbers (0 for `false`; any other integer for `true`), and expressions that evaluate to a Boolean value. Any such new `Boolean` object is imbued with the new properties or methods you add to the `prototype` property of the core `Boolean` object.

For details about the properties and methods of the `Boolean` object, see the corresponding listings for the `String` object in Chapter 27.

✦ ✦ ✦

CHAPTER 29

The Date Object

In This Chapter

Working with date and time values in JavaScript

Performing date calculations

Validating date entry form fields

Perhaps the most untapped power of JavaScript is its date and time handling. Scripters passed over the `Date` object with good cause in the early days of JavaScript, because in earlier versions of scriptable browsers, significant bugs and platform-specific anomalies made date and time programming hazardous without significant testing. Even with the improved bug situation, working with dates requires a working knowledge of the world's time zones and their relationships with the standard reference point, known as Greenwich Mean Time (GMT) or Coordinated Universal Time (abbreviated UTC).

Now that date- and time-handling has improved in the latest browsers, I hope more scripters look into incorporating these kinds of calculations into their pages. In Chapter 54 on the CD-ROM, for example, I show you an application that lets your Web site highlight the areas that have been updated since each visitor's last surf ride through your pages — an application that relies heavily on date arithmetic and time zone conversion.

Before getting to the JavaScript part of date discussions, however, the chapter summarizes key facts about time zones and their impact on scripting date and time on a browser. If you're not sure what GMT and UTC mean, the following section is for you.

Time Zones and GMT

By international agreement, the world is divided into distinct time zones that allow the inhabitants of each zone to say with confidence that when the Sun appears directly overhead, it is roughly noon, squarely in the middle of the day. The current time in the zone is what we set our clocks to — the local time.

That's fine when your entire existence and scope of life go no further than the width of your own time zone. But with instant communication among all parts of the world, your scope reaches well beyond local time. Periodically you must be aware of the local time in other zones. After all, if you live in New York, you don't want to wake up someone in Los Angeles before dawn with a phone call from your office.

For the rest of this section, I speak of the Sun "moving" as if Earth were the center of the solar system. I do so for the convenience of our daily perception of the Sun arcing across what appears to us as a stationary sky. In point of fact, I believe Copernicus's theories, so delete that e-mail you were about to send me.

From the point of view of the time zone over which the Sun is positioned at any given instant, all time zones to the east have already had their noon, so it is later in the day for them — one hour later per time zone (except for those few time zones offset by fractions of an hour). That's why when U.S. television networks broadcast simultaneously to the eastern and central time zones, the announced schedule for a program is "10 eastern, 9 central."

Many international businesses must coordinate time schedules of far-flung events. Doing so and taking into account the numerous time zone differences (not to mention seasonal national variations, such as daylight saving time) would be a nightmare. To help everyone out, a standard reference point was devised: the time zone running through the celestial observatory at Greenwich (pronounced GREN-itch), England. This time zone is called Greenwich Mean Time, or GMT for short. The "mean" part comes from the fact that on the exact opposite side of the globe (through the Pacific Ocean) is the international date line, another world standard that decrees where the first instance of the next calendar day appears on the planet. Thus, GMT is located at the middle, or mean, of the full circuit of the day. Not that many years ago, GMT was given another abbreviation that is not based on any one language of the planet. The abbreviation is UTC (pronounced as its letters: yu-tee-see), and the English version is Coordinated Universal Time. Whenever you see UTC, it is for all practical purposes the same as GMT.

If your personal computer's system clock is set correctly, the machine ticks away in GMT time. But because you set your local time zone in the appropriate control panel, all file time stamps and clock displays are in your local time. The machine knows what the offset time is between your local time and GMT. For daylight saving time, you may have to check a preference setting so that the offset is adjusted accordingly; in Windows 95 and later, the operating system knows when the changeover occurs and prompts you if changing the offset is okay. In any case, if you travel across time zones with a laptop, you should change the computer's time zone setting, not its clock.

JavaScript's inner handling of date and time works a lot like the PC clock (on which your programs rely). Date values that you generate in a script are stored internally in GMT time; however, almost all the displays and extracted values are in the local time of the visitor (not the Web site server). And remember that the date values are created on the visitor's machine by virtue of your script's generating that value — you don't send "living" date objects to the client from the server. This concept is perhaps the most difficult to grasp as you work with JavaScript date and time.

Whenever you program time and date in JavaScript for a public Web page, you must take the worldview. This view requires knowing that the visitor's computer settings determine the accuracy of the conversion between GMT and local time. You'll also have to do some testing by changing your PC's clock to times in other parts of the world and making believe you are temporarily in those remote locations, which isn't always easy to do. It reminds me of the time I was visiting Sydney, Australia. I was turning in for the night and switched on the television in the hotel. This hotel received a live satellite relay of a long-running U.S. television program, *Today*. The program broadcast from New York was for the morning of the same day I was just finishing in Sydney. Yes, this time zone stuff can make your head hurt.

The Date Object

Like a handful of other objects in JavaScript and the document object models, there is a distinction between the single, static `Date` object that exists in every window (or frame) and a date object that contains a specific date and time. The static `Date` object (uppercase "D") is used in only a few cases: Primarily to create a new instance of a date and to invoke a couple of methods that the `Date` object offers for the sake of some generic conversions.

Most of your date and time work, however, is with instances of the `Date` object. These instances are referred to generically as date objects (lowercase "d"). Each date object is a snapshot of an exact millisecond in time, whether it be for the instant at which you generate the object or for a specific time in the past or future you need for calculations. If you need to have a live clock ticking away, your scripts will repeatedly create new date objects to grab up-to-the-millisecond snapshots of your computer's clock. To show the time on the page, extract the hours, minutes, and seconds from the snapshot date object, and then display the values as you like (for example, a digital readout, a graphical bar chart, and so on). By and large, it is the methods of a date object instance that your scripts invoke to read or modify individual components of a date object (for example, the month or hour).

Despite its name, every date object contains information about date and time. Therefore, even if you're concerned only about the date part of an object's data, time data is standing by as well. As you learn in a bit, the time element can catch you off-guard for some operations.

Creating a date object

The statement that asks JavaScript to make an object for your script uses the special object construction keyword `new`. The basic syntax for generating a new date object is as follows:

```
var dateObjectName = new Date([parameters]);
```

The date object evaluates to an object data type rather than to some string or numeric value.

With the date object's reference safely tucked away in the variable name, you access all date-oriented methods in the dot-syntax fashion with which you're already familiar:

```
var result = dateObjectName.method();
```

With variables, such as result, your scripts perform calculations or displays of the date object's data (some methods extract pieces of the date and time data from the object). If you then want to put some new value into the date object (such as adding a year to the date object), you assign the new value to the object by way of the method that lets you set the value:

```
dateObjectName.method(newValue);
```

This example doesn't look like the typical JavaScript assignment statement, which has an equals sign operator. But this statement is the way in which methods that set date object data work.

You cannot get very far into scripting dates without digging into time zone arithmetic. Although JavaScript may render the string equivalent of a date object in your local time zone, the internal storage is strictly GMT.

Even though you haven't yet seen details of a date object's methods, here is how you use two of them to add one year to today's date:

```
var oneDate = new Date();            // creates object with current GMT date
var theYear = oneDate.getYear();     // theYear is now storing the value 2004
theYear = theYear + 1;               // theYear now is 2005
oneDate.setYear(theYear);            // new year value now in the object
```

At the end of this sequence, the `oneDate` object automatically adjusts all the other date components for the next year's date. The day of the week, for example, will be different, and JavaScript takes care of that for you, should you need to extract that data. With next year's data in the `oneDate` object, you may now want to extract that new date as a string value for display in a field on the page or submit it quietly to a CGI program on the server.

The issue of parameters for creating a new date object is a bit complex, mostly because of the flexibility that JavaScript offers the scripter. Recall that the job of the `new Date()` statement is to create a place in memory for all data that a date needs to store. What is missing from that task is the data — what date and time to enter into that memory spot. That's where the parameters come in.

If you leave the parameters empty, JavaScript takes that to mean you want today's date and the current time to be assigned to that new date object. JavaScript isn't any smarter, of course, than the setting of the internal clock of your page visitor's personal computer. If the clock isn't correct, JavaScript won't do any better of a job identifying the date and time.

Remember that when you create a new date object, it contains the current time as well. The fact that the current date may include a time of 16:03:19 (in 24-hour time) may throw off things, such as days-between-dates calculations. Be careful.

To create a date object for a specific date or time, you have five ways to send values as a parameter to the `new Date()` constructor function:

```
new Date("Month dd, yyyy hh:mm:ss")
new Date("Month dd, yyyy")
new Date(yy,mm,dd,hh,mm,ss)
new Date(yy,mm,dd)
new Date(milliseconds)
```

The first four variations break down into two styles — a long string versus a comma-delimited list of data — each with optional time settings. If you omit time settings, they are set to 0 (midnight) in the date object for whatever date you entered. You cannot omit date values from the parameters — every date object must have a real date attached to it, whether you need it or not.

In the long string versions, the month is spelled out in full in English. No abbreviations are allowed. The rest of the data is filled with numbers representing the date, year, hours, minutes, and seconds, even if the order is different from your local way of indicating dates. For single-digit values, you can use either a one- or two-digit version (such as 4:05:00). Colons separate hours, minutes, and seconds.

The short versions contain a non-quoted list of integer values in the order indicated. JavaScript cannot know that a 30 means the date if you accidentally place it in the month slot.

You use the last version only when you have the millisecond value of a date and time available. This generally occurs after some math arithmetic (described later in this chapter), leaving you with a date and time in millisecond format. To convert that numeric value to a date object, use the `new Date()` constructor. From the new date object created, you can retrieve more convenient values about the date and time.

Native object properties and methods

Like the `String` and `Array` objects, the `Date` object features a small handful of properties and methods that all native JavaScript objects have in common. On the property side, the `Date` object has a `prototype` property, which enables you to apply new properties and methods to every date object created in the current page. You can see examples of how this works in discussions of the prototype property for `String` and `Array` objects (Chapters 27 and 30, respectively). At the same time, every instance of a date object in IE4+/W3C has a `constructor` property that references the constructor function that generated the object.

A date object has numerous methods that convert date object types to strings, most of which are more specific than the generic `toString()` one. The `valueOf()` method returns the millisecond integer that is stored for a particular date.

Date methods

The bulk of a date object's methods are for reading parts of the date and time information and for changing the date and time stored in the object. These two categories of methods are easily identifiable because they all begin with the word "get" or "set." Table 29-1 lists all of the methods of both the static `Date` object and, by inheritance, date object instances. The list is impressive — some would say frightening — but there are patterns you should readily observe. Most methods deal with a single component of a date and time value: year, month, date, and so forth. Each block of "get" and "set" methods also has two sets of methods: one for the local date and time conversion of the date stored in the object; one for the actual UTC date stored in the object. After you see the patterns, the list should be more manageable. Unless otherwise noted, a method has been part of the `Date` object since the first generation of scriptable browsers, and is therefore also supported in new browsers.

Table 29-1: Date Object Methods

Method	*Value Range*	*Description*
dateObj.getFullYear()	1970-...	Specified year (NN4+, IE3+)
dateObj.getYear()	70-...	(See Text)
dateObj.getMonth()	0-11	Month within the year (January = 0)
dateObj.getDate()	1-31	Date within the month
dateObj.getDay()	0-6	Day of week (Sunday = 0)
dateObj.getHours()	0-23	Hour of the day in 24-hour time
dateObj.getMinutes()	0-59	Minute of the specified hour
dateObj.getSeconds()	0-59	Second within the specified minute
dateObj.getTime()	0-...	Milliseconds since 1/1/70 00:00:00 GMT
dateObj.getMilliseconds()	0-999	Milliseconds since the previous full second (NN4+, IE3+)
dateObj.getUTCFullYear()	1970-...	Specified UTC year (NN4+, IE3+)
dateObj.getUTCMonth()	0-11	UTC month within the year (January = 0) (NN4+, IE3+)
dateObj.getUTCDate()	1-31	UTC date within the month (NN4+, IE3+)
dateObj.getUTCDay()	0-6	UTC day of week (Sunday = 0) (NN4+, IE3+)
dateObj.getUTCHours()	0-23	UTC hour of the day in 24-hour time (NN4+, IE3+)
dateObj.getUTCMinutes()	0-59	UTC minute of the specified hour (NN4+, IE3+)
dateObj.getUTCSeconds()	0-59	UTC second within the specified minute (NN4+, IE3+)
dateObj.getUTCMilliseconds()	0-999	UTC milliseconds since the previous full second (NN4+, IE3+)
dateObj.setYear(*val*)	1970-...	Be safe: always specify a four-digit year
dateObj.setFullYear(*val*)	1970-...	Specified year (NN4+, IE3+)
dateObj.setMonth(*val*)	0-11	Month within the year (January = 0)

Continued

Table 29-1 *(continued)*

Method	Value Range	Description
dateObj.setDate(*val*)	1-31	Date within the month
dateObj.setDay(*val*)	0-6	Day of week (Sunday = 0)
dateObj.setHours(*val*)	0-23	Hour of the day in 24-hour time
dateObj.setMinutes(*val*)	0-59	Minute of the specified hour
dateObj.setSeconds(*val*)	0-59	Second within the specified minute
dateObj.setMilliseconds(*val*)	0-999	Milliseconds since the previous full second (NN4+, IE3+)
dateObj.setTime(*val*)	0-...	Milliseconds since 1/1/70 00:00:00 GMT
dateObj.setUTCFullYear(*val*)	1970-...	Specified UTC year (NN4+, IE3+)
dateObj.setUTCMonth(*val*)	0-11	UTC month within the year (January = 0) (NN4+, IE3+)
dateObj.setUTCDate(*val*)	1-31	UTC date within the month (NN4+, IE3+)
dateObj.setUTCDay(*val*)	0-6	UTC day of week (Sunday = 0) (NN4+, IE3+)
dateObj.setUTCHours(*val*)	0-23	UTC hour of the day in 24-hour time (NN4+, IE3+)
dateObj.setUTCMinutes(*val*)	0-59	UTC minute of the specified hour (NN4+, IE3+)
dateObj.setUTCSeconds(*val*)	0-59	UTC second within the specified minute (NN4+, IE3+)
dateObj.setUTCMilliseconds(*val*)	0-999	UTC milliseconds since the previous full second (NN4+, IE3+)
dateObj.getTimezoneOffset()	0-...	Minutes offset from GMT/UTC
dateObj.toDateString()		Date-only string in a format determined by browser (WinIE5.5+)
dateObj.toGMTString()		Date/time string in universal format
dateObj.toLocaleDateString()		Date-only string in your system's localized format (NN6+, WinIE5.5+)
dateObj.toLocaleString()		Date/time string in your system's localized format
dateObj.toLocaleTimeString()		Time-only string in your system's localized format (NN6+, WinIE5.5+)
dateObj.toString()		Date/time string in a format determined by browser
dateObj.toTimeString()		Time-only string in a format determined by browser (WinIE5.5+)
dateObj.toUTCString()		Date/time string in universal format (NN4+, IE3+)
Date.parse("*dateString*")		Converts string date to milliseconds integer
Date.UTC(date values)		Converts GMT string date to milliseconds integer

Deciding between using the UTC or local versions of the methods depends on several factors. If the browsers you must support go back to the beginning, you will be stuck with the local versions in any case. But even for newer browsers, activities, such as calculating the number of days between dates or creating a countdown timer for a quiz, won't care which set you use, but you must use the same set for all calculations. If you start mixing local and UTC versions of date methods, you'll be destined to get wrong answers. The UTC versions come in most handy when your date calculations must take into account the time zone of the client machine compared to some absolute in another time zone — calculating the time remaining to the chiming of Big Ben signifying the start of the New Year in London.

JavaScript maintains its date information in the form of a count of milliseconds (thousandths of a second) starting from January 1, 1970, in the GMT (UTC) time zone. Dates before that starting point are stored as negative values (but see the section on bugs and gremlins later in this chapter). Regardless of the country you are in or the date and time formats specified for your computer, the millisecond is the JavaScript universal measure of time. Any calculations that involve adding or subtracting times and dates should be performed in the millisecond values to ensure accuracy. Therefore, though you may never display the milliseconds value in a field or dialog box, your scripts will probably work with them from time to time in variables. To derive the millisecond equivalent for any date and time stored in a date object, use the *dateObj*`.getTime()` method, as in

```
var startDate = new Date();
var started = startDate.getTime();
```

Although the method has the word "time" in its name, the fact that the value is the total number of milliseconds from January 1, 1970, means the value also conveys a date.

Other date object get methods read a specific component of the date or time. You have to exercise some care here, because some values begin counting with 0 when you may not expect it. For example, January is month `0` in JavaScript's scheme; December is month `11`. Hours, minutes, and seconds all begin with `0`, which, in the end, is logical. Calendar dates, however, use the actual number that would show up on the wall calendar: The first day of the month is date value `1`. For the twentieth-century years, the year value is whatever the actual year number is, minus 1900. For 1996, that means the year value is `96`. But for years before 1900 and after 1999, JavaScript uses a different formula, showing the full year value. This means you have to check whether a year value is less than `100` and add `1900` to it before displaying that year.

```
var today = new Date();
var thisYear = today.getYear();
if (thisYear < 100) {
   thisYear += 1900;
}
```

This assumes, of course, you won't be working with years before A.D. 100. If your audience is strictly IE3+/NN4+, which is quite likely, use only the `getFullYear()` method. This method returns the complete set of year digits from all ranges.

To adjust any one of the elements of a date value, use the corresponding set method in an assignment statement. If the new value forces the adjustment of other elements, JavaScript takes care of that. For example, consider the following sequence and how some values are changed for us:

```
myBirthday = new Date("September 11, 2001");
result = myBirthday.getDay(); // result = 2, a Tuesday
myBirthday.setYear(2002);     // bump up to next year
result = myBirthday.getDay(); // result = 3, a Wednesday
```

Because the same date in the following year is on a different day, JavaScript tracks that for you.

Accommodating time zones

Understanding the *dateObj*.getTimezoneOffset() method involves both your operating system's time control panel setting and an internationally recognized (in computerdom, anyway) format for representing dates and times. If you have ignored the control panel stuff about setting your local time zone, the values you get for this property may be off for most dates and times. In the eastern part of North America, for instance, the eastern standard time zone is five hours earlier than Greenwich Mean Time. With the getTimezoneOffset() method producing a value of minutes' difference between GMT and the PC's time zone, the five hours difference of eastern standard time is rendered as a value of 300 minutes. On the Windows platform, the value automatically changes to reflect changes in daylight saving time in the user's area (if applicable). Offsets to the east of GMT (to the date line) are expressed as negative values.

Dates as strings

When you generate a date object, JavaScript automatically applies the toString() method to the object if you attempt to display that date either in a page or alert box. The format of this string varies with browser and operating system platform. For example, in IE6 for Windows XP, the string is in the following format:

```
Mon Jan 5 14:11:49 CDT 2004
```

But in NN7 for Windows XP, the string is

```
Mon Jan 05 2004 14:11:49 GMT-0500 (Central Standard Time)
```

Other browsers return their own variations on the string. The point is not to rely on a specific format and character location of this string for the components of dates. Use the date object methods to read date object components.

JavaScript does, however, provide two methods that return the date object in more constant string formats. One, *dateObj*.toGMTString(), converts the date and time to the GMT equivalent on the way to the variable that you use to store the extracted data. Here is what such data looks like:

```
Mon, 05 Jan 2004 19:11:49 GMT
```

If you're not familiar with the workings of GMT and how such conversions can present unexpected dates, exercise great care in testing your application. Eight o'clock on a Friday evening in California in the winter is four o'clock on Saturday morning GMT.

If time zone conversions make your head hurt, you can use the second string method, *dateObj*.toLocaleString(). In NN7 for North American Windows users, the returned value looks like this:

```
Monday, January 05, 2004 14:11:49
```

Starting with IE5.5 and NN6, you can also have JavaScript convert a date object to just the date or time portions in a nicely formatted version. The best pair of methods for this are toLocaleDateString() and toLocaleTimeString(), because these methods return values that make the most sense to the user, based on the localization settings of the user's operating system and browser.

Friendly date formats for older browsers

If you don't have the luxury of writing only for IE5.5+/NN6+/W3C, you can create your own formatting function to do the job for a wide range of browsers. Listing 29-1 demonstrates one way of creating this kind of string from a date object (in a form compatible with IE4+ and NN4+).

Listing 29-1: Creating a Friendly Date String

```
<html>
   <head>
      <title>Date String Maker</title>
      <script type="text/javascript">
      monthNames = ["January", "February", "March", "April", "May", "June",
"July",
         "August", "September", "October", "November", "December"];
      dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
         "Friday", "Saturday"];

      function customDateString(oneDate) {
         var theDay = dayNames[oneDate.getDay()];
         var theMonth = monthNames[oneDate.getMonth()];
         var theYear = oneDate.getFullYear();
         return theDay + ", " + theMonth + " " + oneDate.getDate() +
            ", " + theYear;
      }
      </script>
   </head>
   <body>
      <h1>
         Welcome!
      </h1>
      <script type="text/javascript">
      document.write(customDateString(new Date()))
      </script>
      <hr />
   </body>
</html>
```

Assuming the user has the PC's clock set correctly (a big assumption), the date appearing just below the opening headline is the current date — making it appear as though the document had been updated today. The downside to this approach (as opposed to the newer `toLocaleDateString()` method) is that international users are forced to view dates in the format you design, which may be different from their local custom.

More conversions

The last two methods shown in Listing 29-1 are methods of the static `Date` object. These utility methods convert dates from string or numeric forms into millisecond values of those dates. The primary beneficiary of these actions is the *dateObj*`.setTime()` method, which requires a millisecond measure of a date as a parameter. You use this method to throw an entirely different date into an existing `date` object.

`Date.parse()` accepts as a parameter date strings similar to the ones you've seen in this section, including the internationally approved version. `Date.UTC()`, on the other hand, requires the comma-delimited list of values (in proper order: *yy*,*mm*,*dd*,*hh*,*mm*,*ss*) in the GMT zone. The `Date.UTC()` method gives you a backward-compatible way to hard-code a GMT time (you can do the same in NN4+ and IE4+ via the UTC methods). The following is an example that creates a new date object for 6 p.m. on March 4, 2004, GMT in WinIE6:

```
var newObj = new Date(Date.UTC(2004,2,4,18,0,0));
result = newObj.toString();    // result = "Thu, Mar 04 10:00:00 PST 2004"
```

The second statement returns a value in a local time zone, because all non-UTC methods automatically convert the GMT time stored in the object to the client's local time.

Date and time arithmetic

You may need to perform some math with dates for any number of reasons. Perhaps you need to calculate a date at some fixed number of days or weeks in the future or figure out the number of days between two dates. When calculations of these types are required, remember the *lingua franca* of JavaScript date values: milliseconds.

What you may need to do in your date-intensive scripts is establish some variable values representing the number of milliseconds for minutes, hours, days, or weeks, and then use those variables in your calculations. Here is an example that establishes some practical variable values, building on each other:

```
var oneMinute = 60 * 1000;
var oneHour = oneMinute * 60;
var oneDay = oneHour * 24;
var oneWeek = oneDay * 7;
```

With these values established in a script, I can use one to calculate the date one week from today:

```
var targetDate = new Date();
var dateInMs = targetDate.getTime();
dateInMs += oneWeek;
targetDate.setTime(dateInMs);
```

Another example uses components of a date object to assist in deciding what kind of greeting message to place in a document, based on the local time of the user's PC clock. Listing 29-2 adds to the scripting from Listing 29-1, bringing some quasi-intelligence to the proceedings.

Listing 29-2: **A Dynamic Welcome Message**

```
<html>
   <head>
      <title>Date String Maker</title>
      <script type="text/javascript">
      monthNames = ["January", "February", "March", "April", "May", "June",
"July",
         "August", "September", "October", "November", "December"];
      dayNames = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
         "Friday", "Saturday"];

      function customDateString(oneDate) {
         var theDay = dayNames[oneDate.getDay()];
         var theMonth = monthNames[oneDate.getMonth()];
         var theYear = oneDate.getFullYear();
         return theDay + ", " + theMonth + " " + oneDate.getDate() +
            ", " + theYear;
      }
      function dayPart(oneDate) {
         var theHour = oneDate.getHours();
         if (theHour < 6 )
            return "wee hours";
         if (theHour < 12)
            return "morning";
```

```
         if (theHour < 18)
            return "afternoon";
         return "evening";
      }
      </script>
   </head>
   <body>
      <h1>Welcome!</h1>
      <script type="text/javascript">
      today = new Date();
      var header = (customDateString(today)).italics();
      header += "<BR>We hope you are enjoying the ";
      header += dayPart(today) + ".";
      document.write(header);
      </script>
      <hr />
   </body>
</html>
```

The script divides the day into four parts and presents a different greeting for each part of the day. The greeting that plays is based, simply enough, on the hour element of a date object representing the time the page is loaded into the browser. Because this greeting is embedded in the page, the greeting does not change no matter how long the user stays logged on to the page.

Counting the days...

You may find one or two more date arithmetic applications useful. One displays the number of shopping days left until Christmas (in the user's time zone); the other is a countdown timer to the start of the year 2100.

Listing 29-3 demonstrates how to calculate the number of days between the current day and some fixed date in the future. The assumption in this application is that all calculations take place in the user's time zone. The example shows the display of the number of shopping days before the next Christmas day (December 25). The basic operation entails converting the current date and the next December 25 to milliseconds, calculating the number of days represented by the difference in milliseconds. If you let the millisecond values represent the dates, JavaScript automatically takes care of leap years.

The only somewhat tricky part is setting the year of the next Christmas day correctly. You can't just slap the fixed date with the current year, because if the program is run on December 26, the year of the next Christmas must be incremented by one. That's why the constructor for the Christmas date object doesn't supply a fixed date as its parameters, but rather, sets individual components of the object.

Listing 29-3: **How Many Days Until Christmas**

```
<html>
  <head>
     <title>Christmas Countdown</title>
     <script type="text/javascript">
     function getDaysUntilXmas() {
        var oneMinute = 60 * 1000;
```

Continued

Listing 29-3 *(continued)*

```
        var oneHour = oneMinute * 60;
        var oneDay = oneHour * 24;
        var today = new Date();
        var nextXmas = new Date();
        nextXmas.setMonth(11);
        nextXmas.setDate(25);
        if (today.getMonth() == 11 && today.getDate() > 25) {
           nextXmas.setFullYear(nextXmas.getFullYear() + 1);
        }
        var diff = nextXmas.getTime() - today.getTime();
        diff = Math.floor(diff/oneDay);
        return diff;
     }
     </script>
  </head>
  <body>
     <h1>
        <script type="text/javascript">
        var header = "You have <i>" + getDaysUntilXmas()  + "<\/i> ";
        header += "shopping days until Christmas.";
        document.write(header);
        </script>
     </h1>
     <hr />
  </body>
</html>
```

The second variation on calculating the amount of time before a certain event takes time zones into account. For this demonstration, the page is supposed to display a countdown timer to the precise moment when the flame for the 2008 Summer Games in Beijing is to be lit. That event takes place in a time zone that may be different from that of the page's viewer, so the countdown timer must calculate the time difference accordingly.

Listing 29-4 shows a simplified version that simply displays the ticking timer in a text field. The output, of course, could be customized in any number of ways, depending on the amount of dynamic HTML you want to employ on a page. The time of the lighting for this demo is set at 11:00 GMT on August 8, 2008 (the date is certainly accurate, but the officials may set a different time closer to the actual event).

Because this application is implemented as a live ticking clock, the code starts by setting some global variables that should be calculated only once so that the function that gets invoked repeatedly has a minimum of calculating to do (to be more efficient). The `Date.UTC()` method provides the target time and date in standard time. The `getTimeUntil()` function accepts a millisecond value (as provided by the `targetDate` variable) and calculates the difference between the target date and the actual internal millisecond value of the client's PC clock.

The core of the `getCountDown()` function peels off the number of whole days, hours, minutes, and seconds from the total number of milliseconds difference between now and the target date. Notice that each chunk is subtracted from the total so that the next smaller chunk can be calculated from the leftover milliseconds.

One extra touch on this page is a display of the local date and time of the actual event.

Listing 29-4: **Summer Games Countdown**

```
<html>
   <head>
      <title>Summer Games Countdown</title>
      <script type="text/javascript">
      // globals -- calculate only once
      // set target date to 1100GMT on August 8, 2008
      var targetDate = Date.UTC(2008, 7, 8, 11, 0, 0, 0);
      var oneMinute = 60 * 1000;
      var oneHour = oneMinute * 60;
      var oneDay = oneHour * 24;

      function getTimeUntil(targetMS) {
         var today = new Date();
         var diff = targetMS - today.valueOf();
         return Math.floor(diff);
      }
      function getCountDown() {
         var ms = getTimeUntil(targetDate);
         var output = "";
         var days, hrs, mins, secs;
         if (ms >= 0) {
            days = Math.floor(ms/oneDay);
            ms -= oneDay * days;
            hrs = Math.floor(ms/oneHour);
            ms -= oneHour * hrs;
            mins = Math.floor(ms/oneMinute);
            ms -= oneMinute * mins;
            secs = Math.floor(ms/1000);
            output += days + " Days, " + hrs + " Hours, " +
               mins + " Minutes, " + secs + " Seconds";
         } else {
            output += "The time has passed.";
         }
         return output;
      }
      function updateCountDown() {
         document.forms[0].timer.value = getCountDown();
         setTimeout("updateCountDown()", 1000);
      }
      </script>
   </head>
   <body onload="updateCountDown()">
      <h1>Beijing Games Torch Lighting Countdown</h1>
      <p>
         <script type="text/javascript">
         document.write("(" + (new Date(targetDate)).toLocaleString());
         document.write(" in your time zone.)");
         </script>
      </p>
      <form>
         <input type="text" name="timer" size="60" />
      </form>
      <hr />
   </body>
</html>
```

Early browser date bugs and gremlins

Each new browser generation improves the stability and reliability of scripted date objects. Unfortunately, Navigator 2 has enough bugs and crash problems across many platforms to make scripting complex world-time applications for this browser impossible. The Macintosh version also has bugs that throw off dates by as much as a full day. I recommend avoiding NN2 on all platforms for serious date and time scripting.

The situation is much improved for NN3. Still, some bugs persist. One bug in particular affects Macintosh versions of NN3. Whenever you create a new date object with daylight saving time engaged in the Date and Time control panel, the browser automatically adds one hour to the object. See the time-based application in Chapter 54 on the CD-ROM for an example of how to counteract the effects of typical time bugs. Also afflicting the Macintosh in NN3 is a faulty calculation of the time zone offset for all time zones east of GMT. Instead of generating these values as negative numbers (getting lower and lower as you head east), the offset values increase continuously as you head west from Greenwich. While the Western Hemisphere is fine, the values continue to increase past the international date line, rather than switch over to the negative values.

Internet Explorer 3 isn't free of problems. It cannot handle dates before January 1, 1970 (GMT). Attempts to generate a date before that one result in that base date as the value. IE3 also completely miscalculates the time zone offset, following the erroneous pattern of NN2. Even Navigators 3 and 4 have problems with historic dates. You are asking for trouble if the date extends earlier than January 1, A.D. 1. Internet Explorer 4, on the other hand, appears to sail very well into ancient history.

You should be aware of one more discrepancy between Mac and Windows versions of Navigator through Version 4. In Windows, if you generate a date object for a date in another part of the year, the browser sets the time zone offset for that object according to the time zone setting for that time of year. On the Mac, the current setting of the control panel governs whether the normal or daylight saving time offset is applied to the date, regardless of the actual date within the year. This discrepancy affects Navigator 3 and 4 and can throw off calculations from other parts of the year by one hour.

It may sound as though the road to `Date` object scripting is filled with land mines. While date and time scripting is far from hassle free, you can put it to good use with careful planning and a lot of testing.

Validating Date Entries in Forms

Given the bug horror stories in the previous section, you may wonder how you can ever perform data entry validation for dates in forms. The problem is not so much in the calculations as it is in the wide variety of acceptable date formats around the world. No matter how well you instruct users to enter dates in a particular format, many will follow their own habits and conventions. Moreover, how can you know whether an entry of 03/04/2002 is the North American March 4, 2002, or the European April 3, 2002? The answer: You can't.

My recommendation is to divide a date field into three components: month, day, and year. Let the user enter values into each field and validate each field individually for its valid range. Listing 29-5 shows an example of how this is done. The page includes a form that is to be validated before it is submitted. Each component field does its own range checking on the fly as the user enters values. But because this kind of validation can be defeated, the page includes one further check triggered by the form's `onsubmit` event handler. If any field is out of whack, the form submission is cancelled.

Listing 29-5: **Date Validation in a Form**

```
<html>
   <head>
      <title>Date Entry Validation</title>
      <script type="text/javascript">
      // **BEGIN GENERIC VALIDATION FUNCTIONS**
      // general purpose function to see if an input value has been entered at
all
      function isEmpty(inputStr) {
         if (inputStr == "" || inputStr == null) {
            return true;
         }
         return false;
      }

      // function to determine if value is in acceptable range for this
application
      function inRange(inputStr, lo, hi) {
         var num = parseInt(inputStr, 10);
         if (num < lo || num > hi) {
            return false;
         }
         return true;
      }
      // **END GENERIC VALIDATION FUNCTIONS**

      function validateMonth(field, bypassUpdate) {
         var input = field.value;
         if (isEmpty(input)) {
            alert("Be sure to enter a month value.");
            select(field);
            return false;
         } else {
            input = parseInt(field.value, 10);
            if (isNaN(input)) {
               alert("Entries must be numbers only.");
               select(field);
               return false;
            } else {
               if (!inRange(input,1,12)) {
                  alert("Enter a number between 1 (January) and 12
(December).");
                  select(field);
                  return false;
               }
            }
         }
         if (!bypassUpdate) {
            calcDate();
         }
         return true;
      }

      function validateDate(field) {
         var input = field.value;
```

Continued

Listing 29-5 *(continued)*

```
      if (isEmpty(input)) {
         alert("Be sure to enter a date value.");
         select(field);
         return false;
      } else {
         input = parseInt(field.value, 10);
         if (isNaN(input)) {
            alert("Entries must be numbers only.");
            select(field);
            return false;
         } else {
            var monthField = document.birthdate.month;
            if (!validateMonth(monthField, true))
               return false;
            var monthVal = parseInt(monthField.value, 10);
            var monthMax = new Array(31,31,29,31,30,31,30,31,31,30,31,30,31);
            var top = monthMax[monthVal];
            if (!inRange(input,1,top)) {
               alert("Enter a number between 1 and " + top + ".");
               select(field);
               return false;
            }
         }
      }
      calcDate();
      return true;
   }

   function validateYear(field) {
      var input = field.value;
      if (isEmpty(input)) {
         alert("Be sure to enter a year value.");
         select(field);
         return false;
      } else {
         input = parseInt(field.value, 10);
         if (isNaN(input)) {
            alert("Entries must be numbers only.");
            select(field);
            return false;
         } else {
            if (!inRange(input,1900,2005)) {
               alert("Enter a number between 1900 and 2005.");
               select(field);
               return false;
            }
         }
      }
      calcDate();
      return true;
   }

   function select(field) {
      field.focus();
      field.select();
   }
```

```
      function calcDate() {
         var mm = parseInt(document.birthdate.month.value, 10);
         var dd = parseInt(document.birthdate.date.value, 10);
         var yy = parseInt(document.birthdate.year.value, 10);
         document.birthdate.fullDate.value = mm + "/" + dd + "/" + yy;
      }

      function checkForm(form) {
         if (validateMonth(form.month)) {
            if (validateDate(form.date)) {
               if (validateYear(form.year)) {
                  return true;
               }
            }
         }
         return false;
      }
      </script>
   </head>
   <body>
      <form name="birthdate" action="mailto:fun@dannyg.com" method="POST"
      onsubmit="return checkForm(this)">
         Please enter your birthdate...<br />
         Month:<input type="text" name="month" value="1" size="2"
         onchange="validateMonth(this)" /> Date:<input type="text" name="date"
         value="1" size="2" onchange="validateDate(this)" /> Year:<input
         type="text" name="year" value="1900" size="4"
         onchange="validateYear(this)" />
         <p>Thank you for entering:<input type="text" name="fullDate"
            size="10" /></p>
         <p><input type="submit" /> <input type="Reset" /></p>
      </form>
   </body>
</html>
```

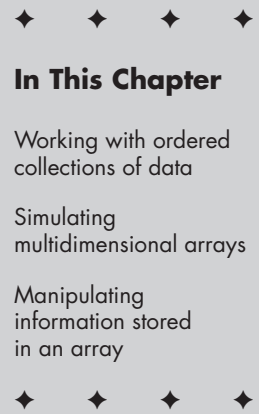
The page shows the three entry fields as well as a field that is normally hidden on a form to be submitted to a CGI program. On the server end, the CGI program responds only to the hidden field with the complete date, which is in a format for entry into, for example, an Informix database.

Not every date entry validation must be divided in this way. For example, an intranet application can be more demanding in the way users are to enter data. Therefore, you can have a single field for date entry, but the parsing required for such a validation is quite different from that shown in Listing 29-5. See Chapter 43 on the CD-ROM for an example of such a one-field date validation routine.

✦ ✦ ✦

CHAPTER 30

The Array Object



An array is the sole JavaScript data structure provided for storing and manipulating ordered collections of data. But unlike some other programming languages, JavaScript's arrays are very forgiving as to the kind of data you store in each cell or entry of the array. This allows, for example, an array of arrays, providing the equivalent of multidimensional arrays customized to the kind of data your application needs.

If you have not done a lot of programming in the past, the notion of arrays may seem like an advanced topic. But if you ignore their capabilities, you set yourself up for a harder job when implementing many kinds of tasks. Whenever I approach a script, one of my first thoughts is about the data being controlled by the application and whether handling it as an array will offer some shortcuts for creating the document and handling interactivity with the user.

I hope that by the end of this chapter, you will not only be familiar with the properties and methods of JavaScript arrays, but you will begin to look for ways to make arrays work for you.

Structured Data

In programming, an *array* is defined as an ordered collection of data. You can best visualize an array as a table, not much different from a spreadsheet. In JavaScript, arrays are limited to a table holding one column of data, with as many rows as needed to hold your data. As you have seen in many chapters in Part III, a JavaScript-enabled browser creates a number of internal arrays for the objects in your HTML documents and browser properties. For example, if your document contains five links, the browser maintains a table of those links. You access them by number (with 0 being the first link) in the array syntax: the array name is followed by the index number in square brackets, as in `document.links[0]`, which represents the first link in the document.

For many JavaScript applications, you will want to use an array as an organized warehouse for data that users of your page access, depending on their interaction with form elements. In the application shown in Chapter 50 on the CD-ROM, for example, I demonstrate an extended version of this usage in a page that lets users search a small table of data for a match between the first three digits of their U.S. Social Security numbers and the state in which they registered with the agency. Arrays are the way JavaScript-enhanced pages can re-create the behavior of more sophisticated CGI programs on servers. When the collection of data you embed in the script is no larger than a typical `.gif` image file, the user won't experience significant delays in loading your page; yet he or she has the full power of your small database collection for instant searching without any calls back to the server. Such database-oriented arrays are important applications of JavaScript for what I call *serverless CGIs*.

As you design an application, look for clues as to potential application of arrays. If you have a number of objects or data points that interact with scripts the same way, you have a good candidate for array structures. For example, you can assign like names to every text field in a column of an order form. In that sequence, like-named objects are treated as elements of an array. To perform repetitive row calculations down an order form, your scripts can use array syntax to perform all the extensions within a handful of JavaScript statements, rather than perhaps dozens of statements hard-coded to each field name. Chapter 51 (on the CD-ROM) shows an example of this application.

You can also create arrays that behave like the Java hash table: a lookup table that gets you to the desired data point instantaneously if you know the name associated with the entry. If you can conceive your data in a table format, an array is in your future.

Creating an Empty Array

Full-fledged array objects were first implemented in NN3 and IE4. It was possible to simulate some array characteristics in earlier browsers, but since those first-generation browsers have largely disappeared from users' computers, this chapter focuses on the modern array and its hefty powers.

To create a new array object, use the static `Array` object's constructor method. For example:

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

An array object automatically has a `length` property (`0` for an empty array).

Should you want to presize the array (for example, preload entries with `null` values), you can specify an initial size as a parameter to the constructor. For example, here is how to create a new array to hold information about a 500-item compact disc collection:

```
var myCDCollection = new Array(500);
```

Unlike many other programming languages presizing a JavaScript array does not give you any particular advantage, because you can assign a value to any slot in an array at any time: The `length` property adjusts itself accordingly. For instance, if you assign a value to `myCDCollection[700]`, the array object adjusts its length upward to meet that slot (with the count starting at 0):

```
myCDCollection [700] = "Gloria Estefan/Destiny";
collectionSize = myCDCollection.length;    // result = 701
```

A true array object also features a number of methods and the capability to add prototype properties, described later in this chapter.

Populating an Array

Entering data into an array is as simple as creating a series of assignment statements, one for each element of the array. Listing 30-1 generates an array containing a list of the nine planets of the solar system.

Listing 30-1: Generating and Populating a New Array

```
solarSys = new Array(9);
solarSys[0] = "Mercury";
solarSys[1] = "Venus";
solarSys[2] = "Earth";
solarSys[3] = "Mars";
solarSys[4] = "Jupiter";
solarSys[5] = "Saturn";
solarSys[6] = "Uranus";
solarSys[7] = "Neptune";
solarSys[8] = "Pluto";
```

This way of populating a single array is a bit tedious when you're writing the code, but after the array is set, it makes accessing information collections as easy as any array reference:

```
onePlanet = solarSys[4];     // result = "Jupiter"
```

A more compact way to create an array is available if you know that the data will be in the desired order (as the preceding `solarSys` array). Instead of writing a series of assignment statements (as in Listing 30-1), you can create what is called a *dense array* by supplying the data as comma-delimited parameters to the `Array()` constructor:

```
solarSys = new Array("Mercury","Venus","Earth","Mars","Jupiter","Saturn",
   "Uranus","Neptune","Pluto");
```

The term "dense array" means that data is packed into the array, without gaps, starting at index position 0.

The example in Listing 30-1 shows what you may call a vertical collection of data. Each data point contains the same type of data as the other data points — the name of a planet — and the data points appear in the relative order of the planets from the Sun.

JavaScript Array Creation Enhancements

The JavaScript version in NN4+/IE4+ provides one more way to create a dense array and also clears up a bug in the old way. A new, simpler way to create a dense array does not require the `Array` object constructor. Instead, JavaScript 1.2 (and later) accepts what is called *literal notation* to generate an array. To demonstrate the difference, the following statement is the regular dense array constructor that works with NN3:

```
solarSys = new Array("Mercury","Venus","Earth","Mars","Jupiter","Saturn",
   "Uranus","Neptune","Pluto");
```

While JavaScript 1.2+ fully accepts the preceding syntax, it also accepts the new literal notation:

```
solarSys = ["Mercury","Venus","Earth","Mars","Jupiter","Saturn",
   "Uranus","Neptune","Pluto"];
```

The square brackets stand in for the call to the `Array` constructor. Unless your audience is stuck using ancient browsers, you should use this streamlined approach to array creation.

The bug fix has to do with how to treat the earlier dense array constructor if the scripter enters only the numeric value `1` as the parameter — `new Array(1)`. In NN3 and IE4, JavaScript erroneously creates an array of length 1, but that element is `undefined`. For NN4 and all later browsers (IE5+, NN6+, W3C), the same statement creates that one-element array and places the value in that element.

Deleting Array Entries

You can always set the value of an array entry to `null` or an empty string to wipe out any data that used to occupy that space. But until the `delete` operator in version 4 browsers, you could not completely remove the element.

Deleting an array element eliminates the index from the list of accessible index values but does not reduce the array's length, as in the following sequence of statements:

```
myArray.length    // result: 5
delete myArray[2]
myArray.length    // result: 5
myArray[2]        // result: undefined
```

The process of deleting an array entry does not necessarily release memory occupied by that data. The JavaScript interpreter's internal garbage collection mechanism (beyond the reach of scripters) is supposed to take care of such activity. See the `delete` operator in Chapter 32 for further details.

Newer browsers feature the `splice()` method of an array, which lets you remove an item (or sequence of items) from an array — causing the array's length to adjust to the new item count. See the `splice()` method later in this chapter.

Parallel Arrays

Using an array to hold data is frequently desirable so that a script can do a lookup to see if a particular value is in the array (perhaps verifying that a value typed into a text box by the user is permissible); however, even more valuable is if, upon finding a match, a script can look up some related information in another array. One way to accomplish this is with two or more parallel arrays: the same indexed slot of each array contains related information.

Consider the following three arrays:

```
var regionalOffices = new Array("New York", "Chicago", "Houston", "Portland");
var regionalManagers = new Array("Shirley Smith", "Todd Gaston",
   "Leslie Jones", "Harold Zoot");
var regOfficeQuotas = new Array(300000, 250000, 350000, 225000);
```

The assumption for these statements is that Shirley Smith is the regional manager out of the New York office, and her office's quota is 300,000. This represents the data that is included with the document, perhaps retrieved by a CGI program on the server that gets the latest data from a SQL database and embeds the data in the form of array constructors. Listing 30-2 shows how this data appears in a simple page that looks up the manager name and quota values for whichever office is chosen in the `select` element. The order of the items in the list of `select` is not accidental: The order is identical to the order of the array for the convenience of the lookup script.

Lookup action in Listing 30-2 is performed by the `getData()` function. Because the index values of the options inside the `select` element match those of the parallel arrays index values, the `selectedIndex` property of the `select` element makes a convenient way to get directly at the corresponding data in other arrays.

Listing 30-2: **A Simple Parallel Array Lookup**

```
<html>
   <head>
      <title>Parallel Array Lookup</title>
      <script type="text/javascript">
      // the data
      var regionalOffices = new Array("New York", "Chicago", "Houston",
         "Portland");
      var regionalManagers = new Array("Shirley Smith", "Todd Gaston",
         "Leslie Jones", "Harold Zoot");
      var regOfficeQuotas = new Array(300000, 250000, 350000, 225000);
      // do the lookup into parallel arrays
      function getData(form) {
         var i = form.offices.selectedIndex;
         form.manager.value = regionalManagers[i];
         form.quota.value = regOfficeQuotas[i];
      }
      </script>
   </head>
   <body onload="getData(document.officeData)">
      <h1>Parallel Array Lookup</h1>
      <hr />
      <form name="officeData">
         <p>Select a regional office: <select name="offices"
         onchange="getData(this.form)">
            <option>New York</option>
            <option>Chicago</option>
            <option>Houston</option>
            <option>Portland</option>
         </select></p>
         <p>The manager is: <input type="text" name="manager" size="35" />
         <br />The office quota is: <input type="text" name="quota"
         size="8" /></p>
      </form>
   </body>
</html>
```

On the other hand, if the content to be looked up is typed into a text box by the user, you have to loop through one of the arrays to get the matching index. Listing 30-3 is a variation of Listing 30-2, but instead of the `select` element, a text field asks users to type in the name of the region. Assuming that users will always spell the input correctly (an outrageous assumption), the version of `getData()` in Listing 30-3 performs actions that more closely resemble what you may think a "lookup" should be doing: looking for a match in one array, and displaying corresponding results from the parallel arrays. The `for` loop iterates through items in the `regionalOffices` array. An `if` condition compares all uppercase versions of both the input and each array entry. If there is a match, the `for` loop breaks, with the value of `i` still pointing to the matching index value. Outside the `for` loop, another `if` condition makes sure that the index value has not reached the length of the array, which means that no match is found.

Only when the value of `i` points to one of the array entries does the script retrieve corresponding entries from the other two arrays.

Listing 30-3: **A Looping Array Lookup**

```
<html>
   <head>
      <title>Parallel Array Lookup II</title>
      <script type="text/javascript">
      // the data
      var regionalOffices = new Array("New York", "Chicago", "Houston",
         "Portland");
      var regionalManagers = new Array("Shirley Smith", "Todd Gaston",
         "Leslie Jones", "Harold Zoot");
      var regOfficeQuotas = new Array(300000, 250000, 350000, 225000);
      // do the lookup into parallel arrays
      function getData(form) {
         // make a copy of the text box contents
         var inputText = form.officeInp.value;
         // loop through all entries of regionalOffices array
         for (var i = 0; i < regionalOffices.length; i++) {
            // compare uppercase versions of entered text against one entry
            // of regionalOffices
            if (inputText.toUpperCase() == regionalOffices[i].toUpperCase()) {
               // if they're the same, then break out of the for loop
               break;
            }
         }
         // make sure the i counter hasn't exceeded the max index value
         if (i < regionalOffices.length) {
            // display corresponding entries from parallel arrays
            form.manager.value = regionalManagers[i];
            form.quota.value = regOfficeQuotas[i];
         } else {  // loop went all the way with no matches
            // empty any previous values
            form.manager.value = "";
            form.quota.value = "";
            // advise user
            alert("No match found for " + inputText + ".");
         }
      }
      </script>
   </head>
   <body>
      <h1>Parallel Array Lookup II</h1>
      <hr />
      <form name="officeData">
         <p>Enter a regional office: <input type="text" name="officeInp"
            size="35" /> <input type="button" value="Search"
            onclick="getData(this.form)" /></p>
         <p>The manager is: <input type="text" name="manager"
            size="35" /><br />
            The office quota is: <input type="text" name="quota"
            size="8" /></p>
      </form>
   </body>
</html>
```

Multidimensional Arrays

An alternate to parallel arrays is the simulation of a multidimensional array. While it's true that JavaScript arrays are one-dimensional, you can create a one-dimensional array of other arrays or objects. A logical approach is to make an array of custom objects, because the objects easily allow for naming of object properties, making references to multidimensional array data more readable (custom objects are discussed at length in Chapter 33).

Using the same data from the examples of parallel arrays, the following statements define an object constructor for each "data record." A new object is then assigned to each of four entries in the main array.

```
// custom object constructor
function officeRecord(city, manager, quota) {
   this.city = city;
   this.manager = manager;
   this.quota = quota;
}

// create new main array
var regionalOffices = new Array();
// stuff main array entries with objects
regionalOffices[0] = new officeRecord("New York", "Shirley Smith", 300000);
regionalOffices[1] = new officeRecord("Chicago", "Todd Gaston", 250000);
regionalOffices[2] = new officeRecord("Houston", "Leslie Jones", 350000);
regionalOffices[3] = new officeRecord("Portland", "Harold Zoot", 225000);
```

The object constructor function (`officeRecord()`) assigns incoming parameter values to properties of the object. Therefore, to access one of the data points in the array, you use both array notations to get to the desired entry in the array and the name of the property for that entry's object:

```
var eastOfficeManager = regionalOffices[0].manager;
```

You can also assign string index values for this kind of array, as in

```
regionalOffices["east"] = new officeRecord("New York", "Shirley Smith",
   300000);
```

and access the data via the same index:

```
var eastOfficeManager = regionalOffices["east"].manager;
```

But if you're more comfortable with the traditional multidimensional array (from your experience in other programming languages), you can also implement the above as an array of arrays with less code:

```
// create new main array
var regionalOffices = new Array();
// stuff main array entries with arrays
regionalOffices[0] = new Array("New York", "Shirley Smith", 300000);
regionalOffices[1] = new Array("Chicago", "Todd Gaston", 250000);
regionalOffices[2] = new Array("Houston", "Leslie Jones", 350000);
regionalOffices[3] = new Array("Portland", "Harold Zoot", 225000);
```

or, for the extreme of unreadable brevity with literal notation:

```
// create new main array
var regionalOffices = [ ["New York", "Shirley Smith", 300000],
                        ["Chicago", "Todd Gaston", 250000],
```

```
                        ["Houston", "Leslie Jones", 350000],
                        ["Portland", "Harold Zoot", 225000] ];
```

Accessing a single data point of an array of arrays requires a double array reference. For example, retrieving the manager's name for the Houston office requires the following syntax:

```
var HoustonMgr = regionalOffices[2][1];
```

The first index in brackets is for the outermost array (`regionalOffices`); the second index in brackets points to the item of the array returned by `regionalOffices[2]`.

Simulating a Hash Table

All arrays shown so far in this chapter have used integers as their index values. A JavaScript array is a special type of object (the `object` type is covered in Chapter 33). As a result, you can also assign values to customized properties of an array without interfering with the data stored in the array or the length of the array. In other words, you can "piggy-back" data in the array object. You may reference the values of these properties either using "dot" syntax (*`array.propertyName`*) or through array-looking syntax consisting of square brackets and the property name as a string inside the brackets (*`array.["propertyName"]`*).

Addressing object properties by way of string indexes is sometimes very useful. For example, the multidimensional array described in the previous section consists of four objects. If your page contains a form whose job is to look through the array to find a match for a city chosen from a `select` list, the typical array lookup would loop through the length of the array, compare the chosen value against the `city` property of each object, and then retrieve the other properties when there was a match. For a 4-item list, this isn't a big deal. But for a 100-item list, the process could get time consuming. A faster approach would be to jump directly to the array entry whose `city` property is the chosen value. That's what a simulated hash table can do for you (some programming languages have formal hash table constructions especially designed to act like a lookup table).

Create a simulated hash table after the array is populated by looping through the array and assigning properties to the array object as string values. Use string values that you expect to use for lookup purposes. For example, after the `regionalOffices` array has its component objects assigned, run through the following routine to make the hash table:

```
for (var i = 0; i < regionalOffices.length; i++) {
   regionalOffices[regionalOffices[i].city] = regionalOffices[i];
}
```

You can retrieve the manager property of the Houston office object as follows:

```
var HoustonMgr = regionalOffices["Houston"].manager;
```

With the aid of the hash table component of the array, your scripts have the convenience of both numeric lookup (if the script needs to cycle through all items) and an immediate jump to an item.

Array Object Properties

constructor

(See `string.constructor` in Chapter 27)

length

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

A true array object's `length` property reflects the number of entries in the array. An entry can be any kind of JavaScript value, including `null`. If an entry is in the 10th cell and the rest are `null`, the length of that array is `10`. Note that because array index values are zero-based, the index of the last cell of an array is one less than the length. This characteristic makes it convenient to use the property as an automatic counter to append a new item to an array:

```
myArray[myArray.length] = valueOfAppendedItem;
```

Thus, a generic function does not have to know which specific index value to apply to an additional item in the array.

prototype

Value: Variable or function. Read/Write
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

Inside JavaScript, an array object has its dictionary definition of methods and `length` property — items that all array objects have in common. The `prototype` property enables your scripts to ascribe additional properties or methods that apply to all the arrays you create in the currently loaded documents. You can override this prototype, however, for any individual object.

Example

To demonstrate how the `prototype` property works, Listing 30-4 creates a `prototype` property for all array objects generated from the static `Array` object. As the script generates new arrays (instances of the `Array` object, just as a date object is an instance of the `Date` object), the property automatically becomes a part of those arrays. In one array, `c`, you override the value of the prototype `sponsor` property. By changing the value for that one object, you don't alter the value of the prototype for the `Array` object. Therefore, another array created afterward, `d`, still gets the original `sponsor` property value.

Listing 30-4: Adding a prototype Property

```
<html>
  <head>
     <title>Array prototypes</title>
     <script type="text/javascript">
     // add prototype to all Array objects
     Array.prototype.sponsor = "DG";
     a = new Array(5);
     b = new Array(5);
     c = new Array(5);
     // override prototype property for one 'instance'
     c.sponsor = "JS";
     // this one picks up the original prototype
     d = new Array(5);
     </script>
  </head>
  <body>
```

Continued

Listing 30-4 *(continued)*

```
      <h2>
         <script type="text/javascript">
         document.write("Array a is brought to you by: " + a.sponsor +
            "<br />");
         document.write("Array b is brought to you by: " + b.sponsor +
            "<br />");
         document.write("Array c is brought to you by: " + c.sponsor +
            "<br />");
         document.write("Array d is brought to you by: " + d.sponsor +
            "<br />");
         </script>
      </h2>
   </body>
</html>
```

You can assign properties and functions to a prototype. To assign a function, define the function as you normally would in JavaScript. Then assign the function to the `prototype` by name:

```
function newFunc(param1) {
   // statements
}
Array.prototype.newMethod = newFunc;  // omit parentheses in this reference
```

When you need to call upon that function (which has essentially become a new temporary method for the `Array` object), invoke it as you would any object method. Therefore, if an array named `CDCollection` has been created and a prototype method `showCoverImage()` has been attached to the array, the call to invoke the method for a tenth listing in the array is

```
CDCollection.showCoverImage(9);
```

where the parameter of the function uses the index value to perhaps retrieve an image whose URL is a property of an object assigned to the 10th item of the array.

Array Object Methods

After you have information stored in an array, JavaScript provides several methods to help you manage that data. These methods, all of which belong to array objects you create, have evolved over time, so observe carefully which browser versions a desired method works with.

array.concat(*array2*)

Returns: Array object.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The *array*.concat() method allows you to join two array objects into a new, third array object. The action of concatenating the arrays does not alter the contents or behavior of the two original arrays. To join the arrays, you refer to the first array object to the left of the period before the method; a reference to the second array is the parameter to the method. For example:

```
var array1 = new Array(1,2,3);
var array2 = new Array("a","b","c");
var array3 = array1.concat(array2);
    // result: array with values 1,2,3,"a","b","c"
```

If an array element is a string or number value (not a string or number object), the values are copied from the original arrays into the new one. All connection with the original arrays ceases for those items. But if an original array element is a reference to an object of any kind, JavaScript copies a reference from the original array's entry into the new array. So if you make a change to either array's entry, the change occurs to the object, and both array entries reflect the change to the object.

Example

Listing 30-5 is a bit complex, but it demonstrates both how arrays can be joined with the *array*.concat() method and how values and objects in the source arrays do or do not propagate based on their data type. The page is shown in Figure 30-1.

After you load the page, you see readouts of three arrays. The first array consists of all string values; the second array has two string values and a reference to a form object on the page (a text box named "original" in the HTML). In the initialization routine of this page, not only are the two source arrays created, but they are joined with the *array*.concat() method, and the result is shown in the third box. To show the contents of these arrays in columns, I use the *array*.join() method, which brings the elements of an array together as a string delimited in this case by a return character — giving us an instant column of data.

Two series of fields and buttons let you experiment with the way values and object references are linked across concatenated arrays. In the first group, if you enter a new value to be assigned to arrayThree[0], the new value replaces the string value in the combined array. Because regular values do not maintain a link back to the original array, only the entry in the combined array is changed. A call to showArrays() proves that only the third array is affected by the change.

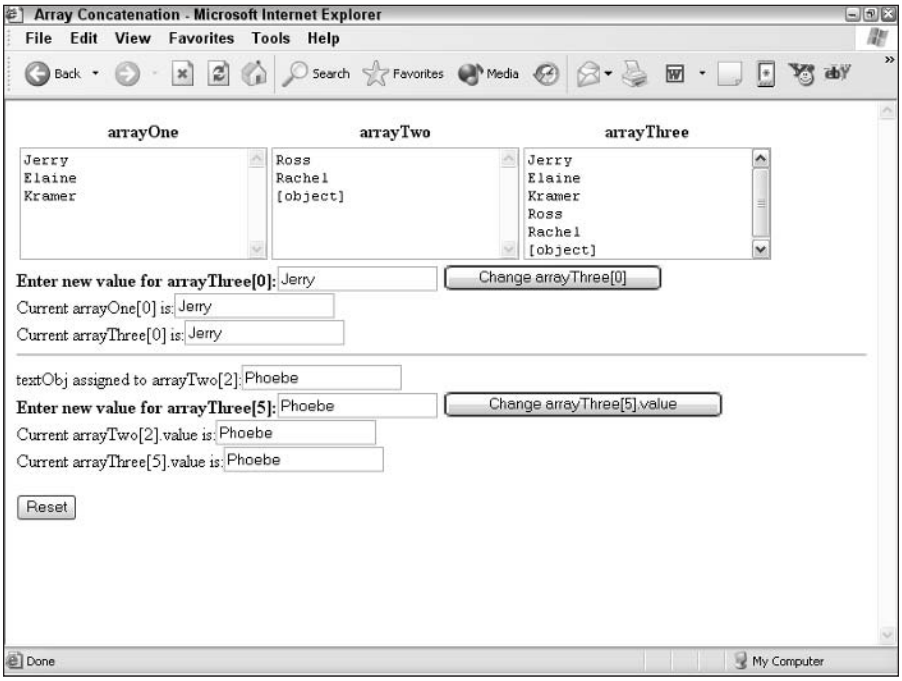


Figure 30-1: Object references remain "alive" in a concatenated array.

More complex is the object relationship for this demonstration. A reference to the first text box of the second grouping has been assigned to the third entry of `arrayTwo`. After concatenation, the same reference is now in the last entry of the combined array. If you enter a new value for a property of the object in the last slot of `arrayThree`, the change goes all the way back to the original object — the first text box in the lower grouping. Thus, the text of the original field changes in response to the change of `arrayThree[5]`. And because all references to that object yield the same result, the reference in `arrayTwo[2]` points to the same text object, yielding the same new answer. The display of the array contents doesn't change, because both arrays still contain a reference to the same object (and the `value` attribute showing in the `<input>` tag of the column listings refers to the default value of the tag, not to its current algorithmically retrievable value shown in the last two fields of the page).

Listing 30-5: **Array Concatenation**

```
<html>
   <head>
      <title>Array Concatenation</title>
      <script type="text/javascript">
      // global variables
      var arrayOne, arrayTwo, arrayThree, textObj;
      // initialize after load to access text object in form
      function initialize() {
         var form = document.forms[0];
         textObj = form.original;
         arrayOne = new Array("Jerry", "Elaine","Kramer");
         arrayTwo = new Array("Ross", "Rachel",textObj);
         arrayThree = arrayOne.concat(arrayTwo);
         update1(form);
         update2(form);
         showArrays();
      }
      // display current values of all three arrays
      function showArrays() {
         var form = document.forms[0];
         form.array1.value = arrayOne.join("\n");
         form.array2.value = arrayTwo.join("\n");
         form.array3.value = arrayThree.join("\n");
      }
      // change the value of first item in Array Three
      function update1(form) {
         arrayThree[0] = form.source1.value;
         form.result1.value = arrayOne[0];
         form.result2.value = arrayThree[0];
         showArrays();
      }
      // change value of object property pointed to in Array Three
      function update2(form) {
         arrayThree[5].value = form.source2.value;
         form.result3.value = arrayTwo[2].value;
         form.result4.value = arrayThree[5].value;
         showArrays();
      }
      </script>
   </head>
   <body onload="initialize()">
      <form>
```

```
            <table>
               <tr>
                  <th>arrayOne</th>
                  <th>arrayTwo</th>
                  <th>arrayThree</th>
               </tr>
               <tr>
                  <td><textarea name="array1" cols="25" rows="6">
                     </textarea></td>
                  <td><textarea name="array2" cols="25" rows="6">
                     </textarea></td>
                  <td><textarea name="array3" cols="25" rows="6">
                     </textarea></td>
               </tr>
            </table>
            <b>Enter new value for arrayThree[0]:</b><input type="text"
            name="source1" value="Jerry" /> <input type="button"
            value="Change arrayThree[0]" onclick="update1(this.form)" /><br />
            Current arrayOne[0] is:<input type="text" name="result1" /><br />
            Current arrayThree[0] is:<input type="text" name="result2" /><br />
            <hr />
            textObj assigned to arrayTwo[2]:<input type="text" name="original"
            onfocus="this.blur()" /><br />
             <b>Enter new value for arrayThree[5]:</b><input type="text"
            name="source2" value="Phoebe" /> <input type="button"
            value="Change arrayThree[5].value"
            onclick="update2(this.form)" /><br />
            Current arrayTwo[2].value is:<input type="text"
            name="result3" /><br />
            Current arrayThree[5].value is:<input type="text" name="result4" />
            <p><input type="button" value="Reset"
            onclick="location.reload()" /></p>
         </form>
      </body>
   </html>
```

Related Items: `array.join()` method.

array.join(*separatorString*)

Returns: String of entries from the array delimited by the *separatorString* value.
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

You cannot view data in an array when it's in that form. Nor can you put an array into a form element for transmittal to a server CGI program. To make the transition from discrete array elements to string, the *array*`.join()` method handles what would otherwise be a nasty string manipulation exercise.

The sole parameter for this method is a string of one or more characters that you want to act as a delimiter between entries. For example, if you want commas between array items in their text version, the statement is

```
var arrayText = myArray.join(",");
```

Invoking this method does not change the original array in any way. Therefore, you need to assign the results of this method to another variable or a value property of a form element.

Example

The script in Listing 30-6 converts an array of planet names into a text string. The page provides you with a field to enter the delimiter string of your choice and shows the results in a text area.

Listing 30-6: Using the Array.join() Method

```
<html>
   <head>
      <title>Array.join()</title>
      <script type="text/javascript">
      solarSys = new Array(9);
      solarSys[0] = "Mercury";
      solarSys[1] = "Venus";
      solarSys[2] = "Earth";
      solarSys[3] = "Mars";
      solarSys[4] = "Jupiter";
      solarSys[5] = "Saturn";
      solarSys[6] = "Uranus";
      solarSys[7] = "Neptune";
      solarSys[8] = "Pluto";

      // join array elements into a string
      function convert(form) {
         var delimiter = form.delim.value;
         form.output.value = unescape(solarSys.join(delimiter));
      }
      </script>
   </head>
   <body>
      <h2>Converting arrays to strings</h2>
      This document contains an array of planets in our solar system.
      <hr />
      <form>
         Enter a string to act as a delimiter between entries: <input
         type="text" name="delim" value="," size="5" />
         <p><input type="button" value="Display as String"
            onclick="convert(this.form)" /> <input type="reset" /> <textarea
            name="output" rows="4" cols="40" wrap="virtual">
            </textarea></p>
      </form>
   </body>
</html>
```

Notice that this method takes the parameter very literally. If you want to include non-alphanumeric characters, such as a newline or tab, do so with URL-encoded characters (`%0D` for a carriage return; `%09` for a tab) instead of inline string literals. In Listing 30-7, the results of the *array*`.join()` method are subjected to the `unescape()` function in order to display them in the `textarea`.

Related Items: `string.split()` method.

array.pop()
array.push(*valueOrObject*)
array.shift()
array.unshift(*valueOrObject*)

Returns: One array entry value.
Compatibility: WinIE5.5+, MacIE-, NN4+, Moz1+, Safari1+

The notion of a *stack* is well known to experienced programmers, especially those who know about the inner workings of assembly language at the CPU level. Even if you've never programmed a stack before, you have encountered the concept in real life many times. The classic analogy is the spring-loaded pile of cafeteria trays. If the pile were created one tray at a time, each tray would be pushed into the stack of trays. When a customer comes along, the topmost tray (the last one to be pushed onto the stack) gets popped off. The last one to be put on the stack is the first one to be taken off.

JavaScript in NN4+, IE5.5+, and W3C browsers lets you turn an array into one of these spring-loaded stacks. But instead of placing trays on the pile, you can place any kind of data at either end of the stack, depending on which method you use to do the stacking. Similarly, you can extract an item from either end.

Perhaps the most familiar terminology for this is *push* and *pop*. When you `push()` a value onto an array, the value is appended as the last entry in the array. When you issue the *array*`.pop()` method, the last item in the array is removed from the stack and is returned, and the array shrinks in length by one. In the following sequence of statements, watch what happens to the value of the array used as a stack:

```
var source = new Array("Homer","Marge","Bart","Lisa","Maggie");
var stack = new Array();
    // stack = <empty>
stack.push(source[0]);
    // stack = "Homer"
stack.push(source[2]);
    // stack = "Homer","Bart"
var Simpson1 = stack.pop();
    // stack = "Homer" ; Simpson1 = "Bart"
var Simpson2 = stack.pop();
    // stack = <empty> ; Simpson2 = "Homer"
```

While `push()` and `pop()` work at the end of an array, another pair of methods works at the front. Their names are not as picturesque as `push()` and `pop()`. To insert a value at the front of an array, use the *array*`.unshift()` method; to grab the first element and remove it from the array, use *array*`.shift()`. Of course, you are not required to use these methods in matching pairs. If you `push()` a series of values onto the back end of an array, you can `shift()` them off from the front end without complaint. It all depends on how you need to process the data.

Related Items: `array.concat()`, `array.slice()` method.

array.reverse()

Returns: Array of entries in the opposite order of the original.
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

Occasionally, you may find it more convenient to work with an array of data in reverse order. Although you can concoct repeat loops to count backward through index values, a CGI program on the server may prefer the data in a sequence opposite to the way it was most convenient for you to script it.

You can have JavaScript switch the contents of an array for you: Whatever element was last in the array becomes the 0 index item in the array. Bear in mind that if you do this, you're restructuring the original array, not copying it, even though the method also returns a copy of the reversed version. A reload of the document restores the order as written in the HTML document.

Example

Listing 30-7 is an enhanced version of Listing 30-6, which includes another button and function that reverse the array and display it as a string in a text area.

Listing 30-7: **Array.reverse() Method**

```
<html>
   <head>
      <title>Array.reverse()</title>
      <script type="text/javascript">
      solarSys = new Array(9);
      solarSys[0] = "Mercury";
      solarSys[1] = "Venus";
      solarSys[2] = "Earth";
      solarSys[3] = "Mars";
      solarSys[4] = "Jupiter";
      solarSys[5] = "Saturn";
      solarSys[6] = "Uranus";
      solarSys[7] = "Neptune";
      solarSys[8] = "Pluto";

      // show array as currently in memory
      function showAsIs(form) {
         var delimiter = form.delim.value;
         form.output.value = unescape(solarSys.join(delimiter));
      }
      // reverse array order, then display as string
      function reverseIt(form) {
         var delimiter = form.delim.value;
         solarSys.reverse();   // reverses original array
         form.output.value = unescape(solarSys.join(delimiter));
      }
      </script>
   </head>
   <body>
      <h2>Reversing array element order</h2>
      This document contains an array of planets in our solar system.
      <hr />
      <form>
         Enter a string to act as a delimiter between entries: <input
         type="text" name="delim" value="," size="5" />
         <p><input type="button" value="Array as-is"
            onclick="showAsIs(this.form)" /> <input type="button"
            value="Reverse the array" onclick="reverseIt(this.form)" /> <input
            type="reset" /> <input type="button" value="Reload"
            onclick="self.location.reload()" /> <textarea name="output"
            rows="4" cols="60">
            </textarea></p>
      </form>
   </body>
</html>
```

Notice that the `solarSys.reverse()` method stands by itself (meaning, nothing captures the returned value) because the method modifies the `solarSys` array. You then run the now inverted `solarSys` array through the *`array`*`.join()` method for your text display.

Related Items: `array.sort()` method.

array.slice(*startIndex* [, *endIndex*])

Returns: Array.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

Behaving as its like-named string method, *`array`*`.slice()` lets you extract a contiguous series of items from an array. The extracted segment becomes an entirely new array object. Values and objects from the original array have the same kind of behavior as arrays created with the *`array`*`.concat()` method.

One parameter is required — the starting index point for the extraction. If you don't specify a second parameter, the extraction goes all the way to the end of the array; otherwise the extraction goes to, *but does not include,* the index value supplied as the second parameter. For example, extracting Earth's neighbors from an array of planet names looks like the following:

```
var solarSys = new Array("Mercury","Venus","Earth","Mars",
   "Jupiter","Saturn","Uranus","Neptune","Pluto");
var nearby = solarSys.slice(1,4);
    // result: new array of "Venus", "Earth", "Mars"
```

Related Items: `array.splice()`, `string.slice()` methods.

array.sort([*compareFunction*])

Returns: Array of entries in the order as determined by the *compareFunction* algorithm.
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

JavaScript array sorting is both powerful and a bit complex to script if you haven't had experience with this kind of sorting methodology. The purpose, obviously, is to let your scripts sort entries of an array by almost any kind of criterion that you can associate with an entry. For entries consisting of strings, the criterion may be their alphabetical order or their length; for numeric entries, the criterion may be their numerical order.

Look first at the kind of sorting you can do with the *`array`*`.sort()` method by itself (for example, without calling a comparison function). When no parameter is specified, JavaScript takes a snapshot of the contents of the array and converts items to strings. From there, it performs a string sort of the values. ASCII values of characters govern the sort, which means that numbers are sorted by their string values, not their numeric values. This fact has strong implications if your array consists of numeric data: The value 201 sorts before 88, because the sorting mechanism compares the first characters of the strings ("2" versus "8") to determine the sort order. For simple alphabetical sorting of string values in arrays, the plain *`array`*`.sort()` method does the trick.

Fortunately, additional intelligence is available that you can add to array sorting. The key tactic is to define a function that helps the `sort()` method compare items in the array. A comparison function is passed two values from the array (what you don't see is that the *`array`*`.sort()` method rapidly sends numerous pairs of values from the array to help it sort through all entries). The comparison function lets the `sort()` method know which of the two items comes before the other, based on the value the function returns. Assuming that the function compares two values, `a` and `b`, the returned value reveals information to the `sort()` method, as shown in Table 30-1.

Table 30-1: Comparison Function Return Values

Return Value Range	*Meaning*
< 0	Value b should sort later than a
0	The order of a and b should not change
> 0	Value a should sort later than b

Consider the following example:

```
myArray = new Array(12, 5, 200, 80);
function compare(a,b) {
   return a - b;
}
myArray.sort(compare);
```

The array has four numeric values in it. To sort the items in numerical order, you define a comparison function (arbitrarily named `compare()`), which is called from the `sort()` method. Note that unlike invoking other functions, the parameter of the `sort()` method uses a reference to the function, which lacks parentheses.

When the `compare()` function is called, JavaScript automatically sends two parameters to the function in rapid succession until each element has been compared with the others. Every time `compare()` is called, JavaScript assigns two of the array's values to the parameter variables (`a` and `b`). In the preceding example, the returned value is the difference between `a` and `b`. If `a` is larger than `b`, then a positive value goes back to the `sort()` method, telling it to sort `a` later than `b` (that is, position `a` at a higher value index position than `b`). Therefore, `b` may end up at `myArray[0]`, whereas `a` ends up at a higher index-valued location. On the other hand, if `a` is smaller than `b`, the returned negative value tells `sort()` to put `a` in a lower index value spot than `b`.

Evaluations within the comparison function can go to great lengths, as long as some data connected with array values can be compared. For example, instead of numerical comparisons, as just shown, you can perform string comparisons. The following function sorts alphabetically by the last character of each array string entry:

```
function compare(a,b) {
   // last character of array strings
   var aComp = a.charAt(a.length - 1);
   var bComp = b.charAt(b.length - 1);
   if (aComp < bComp)
      return -1;
   if (aComp > bComp)
      return 1;
   return 0;
}
```

First, this function extracts the final character from each of the two values passed to it. Then, because strings cannot be added or subtracted like numbers, you compare the ASCII values of the two characters, returning the corresponding values to the `sort()` method to let it know how to treat the two values being checked at that instant.

When an array's entries happen to be objects, you can even sort by properties of those objects. If you bear in mind that the `a` and `b` parameters of the sort function are references to two array entries, then by extension you can refer to properties of those objects. For example, if an array

contains objects whose properties define information about employees, one of the properties of those objects can be the employee's age as a string. You can then sort the array based on the numeric equivalent of the `age` property of the objects by way of the following comparison function:

```
function compare(a,b) {
   return parseInt(a.age) - parseInt(b.age);
}
```

Array sorting, unlike sorting routines you may find in other scripting languages, is not a stable sort. Not being stable means that succeeding sort routines on the same array are not cumulative. Also, remember that sorting changes the sort order of the original array. If you don't want the original array harmed, make a copy of it before sorting or reload the document to restore an array to its original order. Should an array element be `null`, the method sorts such elements at the end of the sorted array.

JavaScript array sorting is extremely powerful stuff. Array sorting is one reason why it's not uncommon to take the time during the loading of a page containing an IE XML data island to make a JavaScript copy of the data as an array of objects (see Chapter 57 on the CD-ROM). Converting the XML to JavaScript arrays makes the job of sorting the data much easier and faster than cobbling together your own sorting routines on the XML elements.

Example

You can look to Listing 30-8 for a few examples of sorting an array of string values (see Figure 30-2). Four buttons summon different sorting routines, three of which invoke comparison functions. This listing sorts the planet array alphabetically (forward and backward) by the last character of the planet name and also by the length of the planet name. Each comparison function demonstrates different ways of comparing data sent during a sort.

Listing 30-8: **Array.sort() Possibilities**

```
<html>
   <head>
      <title>Array.sort()</title>
      <script type="text/javascript">
      solarSys = new Array(9);
      solarSys[0] = "Mercury";
      solarSys[1] = "Venus";
      solarSys[2] = "Earth";
      solarSys[3] = "Mars";
      solarSys[4] = "Jupiter";
      solarSys[5] = "Saturn";
      solarSys[6] = "Uranus";
      solarSys[7] = "Neptune";
      solarSys[8] = "Pluto";
      // comparison functions
      function compare1(a,b) {
         // reverse alphabetical order
         if (a > b)
            return -1;
         if (b > a)
            return 1;
         return 0;
```

Continued

Listing 30-8 *(continued)*

```
      }
      function compare2(a,b) {
         // last character of planet names
         var aComp = a.charAt(a.length - 1);
         var bComp = b.charAt(b.length - 1);
         if (aComp < bComp)
            return -1;
         if (aComp > bComp)
            return 1;
         return 0;
      }
      function compare3(a,b) {
         // length of planet names
         return a.length - b.length;
      }
      // sort and display array
      function sortIt(form, compFunc) {
         var delimiter = ";";
         if (compFunc == null) {
            solarSys.sort();
         } else {
            solarSys.sort(compFunc);
         }
         // display results in field
         form.output.value = unescape(solarSys.join(delimiter));
      }
      </script>
   </head>
   <body onload="document.forms[0].output.value =
      unescape(solarSys.join(';'))">
      <h2>Sorting array elements</h2>
      This document contains an array of planets in our solar system.
      <hr />
      <form>
         Click on a button to sort the array:
         <p><input type="button" value="Alphabetical A-Z"
            onclick="sortIt(this.form)" /> <input type="button"
            value="Alphabetical Z-A" onclick="sortIt(this.form,compare1)" />
            <input type="button" value="Last Character"
            onclick="sortIt(this.form,compare2)" /> <input type="button"
            value="Name Length" onclick="sortIt(this.form,compare3)" /> <input
            type="button" value="Reload Original"
            onclick="self.location.reload()" /> <input type="text"
            name="output" size="62" /></p>
      </form>
   </body>
</html>
```

Related Items: `array.reverse()` method.

As I show you in Chapter 42 on the CD-ROM, many regular expression object methods generate arrays as their result (for example, an array of matching values in a string). These special arrays have a custom set of named properties that assist your script in analyzing the findings of the method. Beyond that, these regular expression result arrays behave like all others.

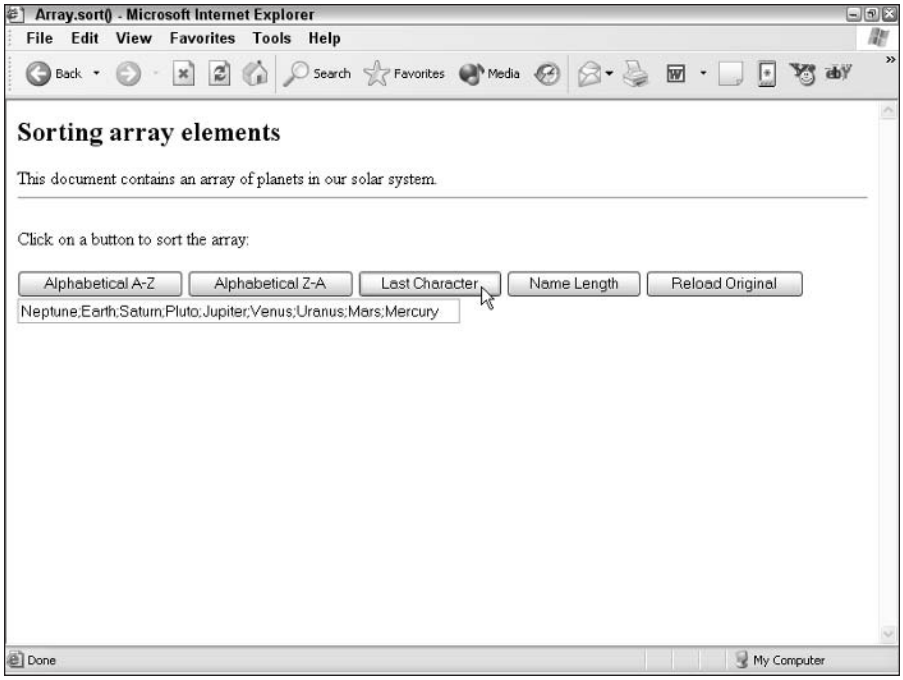


Figure 30-2: Sorting an array of planet names alphabetically by last character.

array.splice(*startIndex* , *deleteCount*[, *item1*[, *item2*[,...*itemN*]]])

Returns: Array.
Compatibility: WinIE5.5+, MacIE-, NN4+, Moz1+, Safari1+

If you need to remove items from the middle of an array, the *array*.splice() method simplifies a task that would otherwise require assembling a new array from selected items of the original array. The first of two required parameters is a zero-based index integer that points to the first item to be removed from the current array. The second parameter is another integer that indicates how many sequential items are to be removed from the array. Removing array items affects the length of the array, and those items that are removed are returned by the splice() method as their own array.

You can also use the splice() method to replace array items. Optional parameters beginning with the third let you provide data elements that are to be inserted into the array in place of the items being removed. Each added item can be any JavaScript data type, and the number of new items does not have to be equal to the number of items removed. In fact, by specifying a second parameter of zero, you can use splice() to insert one or more items into any position of the array.

Example

Use The Evaluator (Chapter 13) to experiment with the splice() method. Begin by creating an array with a sequence of numbers:

```
a = new Array(1,2,3,4,5)
```

Next, remove the center three items, and replace them with one string item:

```
a.splice(1, 3, "two/three/four")
```

The Results box shows a string version of the three-item array returned by the method. To view the current contents of the array, enter `a` into the top text box.

To put the original numbers back into the array, swap the string item with three numeric items:

```
a.splice(1, 1, 2, 3, 4)
```

The method returns the single string, and the `a` array now has five items in it again.

Related Items: `array.slice()` method.

array.toLocaleString()

Returns: String.
Compatibility: WinIE5.5+, MacIE-, NN6+, Moz1+, Safari1+

array.toString()

Returns: String.
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

The *array*`.toLocaleString()` and the older, more compatible *array*`.toString()` are methods to retrieve the contents of an array in string form. Browsers use the `toString()` method on their own whenever you attempt to display an array in text boxes, in which case the array items are comma-delimited.

The precise string conversion of the `toLocaleString()` is left up to the specific browser implementation. That IE and NN differ in some details is not surprising, even in the U.S. English versions of operating systems and browsers. For example, if the array contains integer values, the `toLocaleString()` method in IE5.5+ returns the numbers comma-and-space-delimited, formatted with two digits to the right of the decimal (as if dollars and cents). NN6+, on the other hand, returns just the integers, but these are also comma-and-space-delimited.

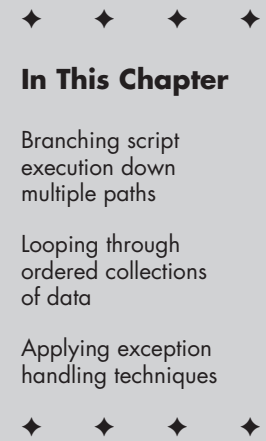
If you need to convert an array to a string for purposes of passing array data to other venues (for example, as data in a hidden text box submitted to a server or as search string data conveyed to another page), use the *array*`.join()` method instead. *Array*`.join()` gives you more reliable and flexible control over the item delimiters, and you are assured of the same results regardless of locale.

Related Items: `array.join()` method.

✦ ✦ ✦

CHAPTER 31

Control Structures and Exception Handling



In This Chapter

Branching script execution down multiple paths

Looping through ordered collections of data

Applying exception handling techniques

You get up in the morning, go about your day's business, and then turn out the lights at night. That's not much different from what a program does from the time it starts to the time it ends. But along the way, both you and a program take lots of tiny steps, not all of which advance the "processing" in a straight line. At times, you have to control what's going on by making a decision or repeating tasks until the whole job is finished. Control structures are the facilities that make these tasks possible in JavaScript.

JavaScript control structures follow along the same lines of many programming languages, particularly with additions made in Navigator 4 and Internet Explorer 4. Basic decision-making and looping constructions satisfy the needs of just about all programming tasks.

Another vital program control mechanism — error (or exception) handling — is formally addressed in Edition 3 of the ECMA-262 language standard. The concept of exception handling was added to the JavaScript version introduced in IE5.5 and NN6, but it is well known to programmers in many other environments. Adopting exception-handling techniques in your code can greatly enhance recovery from processing errors caused by errant user input or network glitches.

If and If. . .Else Decisions

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

JavaScript programs frequently have to make decisions based on the current values of variables or object properties. Such decisions can have only two possible outcomes at a time. The factor that determines the path that the program takes at these decision points is the truth of some statement. For example, when you enter a room of your home at night, the statement under test is something such as "It is too dark to see without a light." If that statement is true, you switch on the light; if that statement is false, you carry on with your primary task.

Simple decisions

JavaScript syntax for this kind of simple decision always begins with the keyword `if`, followed by the condition to test, and then the statements that execute if the condition yields a true result. JavaScript uses no "then" keyword (as some other languages do); the keyword is implied by the way parentheses and braces surround the various components of this construction. The formal syntax is

```
if (condition) {
   statementsIfTrue
}
```

This construction means that if the condition is true, program execution takes a detour to execute statements inside the braces. No matter what happens, the program continues executing statements beyond the closing brace (`}`). If household navigation were part of the scripting language, the code would look like this:

```
if (tooDark == true) {
   feel for light switch
   turn on light switch
}
```

If you're not used to C/C++, the double equals sign may have caught your eye. You learn more about this type of operator in the next chapter, but for now, know that this operator compares the equality of items on either side of it. In other words, the `condition` statement of an `if` construction must always yield a Boolean (`true` or `false`) value. Some object properties, you may recall, are Booleans, so you can stick a reference to that property into the `condition` statement by itself. Otherwise, the `condition` statement consists of two values separated by a comparison operator, such as `==` (equals) or `!=` (does not equal).

Next, look at some real JavaScript. The following function receives a form object containing a text object called `entry`:

```
function notTooHigh(form) {
   if (parseInt(form.entry.value) > 100) {
      alert("Sorry, the value you entered is too high. Try again.");
      return false;
   }
   return true;
}
```

The condition (in parentheses) tests the contents of the field against a hard-wired value of `100`. If the entered value is larger than that, the function alerts you and returns a `false` value to the calling statement elsewhere in the script. But if the value is less than `100`, all intervening code is skipped and the function returns `true`.

About (*condition*) expressions

A lot of condition testing for control structures compares a value against some very specific condition, such as a string's being empty or a value's being `null`. You can use a couple of shortcuts to take care of many circumstances. Table 31-1 details the values that evaluate to a true or false (or equivalent) to satisfy a control structure's *condition* expression.

Table 31-1: Condition Value Equivalents

True	*False*
Nonempty string	Empty string
Nonzero number	0
Nonnull value	Null
Object exists	Object doesn't exist
Property is defined	Undefined property

Instead of having to spell out an equivalency expression for a condition involving these kinds of values, you can simply supply the value to be tested. For example, if a variable named `myVal` may reach an `if` construction with a value of `null`, an empty string, or a string value for further processing, you can use the following shortcut:

```
if (myVal) {
   // do processing on myVal
}
```

All null or empty string conditions evaluate to `false`, so that only the cases of `myVal`'s being a processable value get inside the `if` construction. This mechanism is the same that you have seen elsewhere in this book to employ object detection for browser branching. For example, the code nested inside the following code segment executes only if the `document` object has an `images` array property:

```
if (document.images) {
   // do processing on image objects
}
```

Complex decisions

The simple type of `if` construction described earlier is fine when the decision is to take a small detour before returning to the main path. But not all decisions — in programming or in life — are like that. To present two alternate paths in a JavaScript decision, you can add a component to the construction. The syntax is

```
if (condition) {
   statementsIfTrue
} else {
   statementsIfFalse
}
```

By appending the `else` keyword, you give the `if` construction a path to follow in case the condition evaluates to false. The *`statementsIfTrue`* and *`statementsIfFalse`* do not have to be balanced in any way: One statement can be one line of code, the other 100 lines. But when either one of those branches completes, execution continues after the last closing brace. To demonstrate how this construction can come in handy, the following example is a script fragment that assigns the number of days in February based on whether or not the year is a leap year (using modulo arithmetic, explained in Chapter 32, to determine if the year is evenly divisible by four, and setting aside all other leap year calculation details for the moment):

```
var howMany = 0;
var theYear = 2002;
if (theYear % 4 == 0) {
   howMany = 29;
```

```
} else {
   howMany = 28;
}
```

Here is a case where execution has to follow only one of two possible paths to assign the number of days to the `howMany` variable. Had I not used the `else` portion, as in

```
var howMany = 0;
var theYear = 2002;
if (theYear % 4 == 0) {
    howMany = 29;
}
howMany = 28;
```

then the variable would always be set to 28, occasionally after momentarily being set to 29. The `else` construction is essential in this case.

Nesting if. . .else statements

Designing a complex decision process requires painstaking attention to the logic of the decisions your script must process and the statements that must execute for any given set of conditions. The need for many complex constructions disappeared with the advent of `switch` construction in NN4+/IE4+ (described later in this chapter), but there may still be times when you must fashion complex decision behavior out of a series of nested `if. . .else` constructions. Without a JavaScript-aware text editor to help keep everything properly indented and properly terminated (with closing braces), you have to monitor the authoring process very carefully. Moreover, the error messages that JavaScript provides when a mistake occurs (see Chapter 45 on the CD-ROM) may not point directly to the problem line but only to the region of difficulty.

Another important point to remember about nesting `if. . .else` statements in JavaScript before Version 1.2 is that the language does not provide a mechanism for script execution to break out of a nested part of the construction. For that reason, you have to construct complex assemblies with extreme care to make sure only the desired statement executes for each set of conditions. Extensive testing, of course, is also required (see Chapter 45 on the CD-ROM).

To demonstrate a deeply nested set of `if. . .else` constructions, Listing 31-1 presents a simple user interface to a complex problem. A single text object asks the user to enter one of three letters — A, B, or C. The script behind that field processes a different message for each of the following conditions:

- ✦ The user enters no value.
- ✦ The user enters A.
- ✦ The user enters B.
- ✦ The user enters C.
- ✦ The user enters something entirely different.

Listing 31-1: **Deeply Nested if. . .else Constructions**

```
<html>
  <head>
     <title></title>
```

```
      <script type="text/javascript">
      function testLetter(form){
         inpVal = form.entry.value;  // assign to shorter variable name
         if (inpVal != "") {  // if entry is not empty then dive in...
            if (inpVal == "A") {  // Is it an "A"?
               alert("Thanks for the A.");
            } else if (inpVal == "B") {  // No.  Is it a "B"?
               alert("Thanks for the B.");
            } else if (inpVal == "C") {  // No.  Is it a "C"?
               alert("Thanks for the C.");
            } else {          // Nope.  None of the above
               alert("Sorry, wrong letter or case.");
            }
         } else {   // value was empty, so skipped all other stuff above
            alert("You did not enter anything.");
         }
      }
      </script>
   </head>
   <body>
      <form onsubmit="return false">
         Please enter A, B, or C: <input type="text" name="entry"
         onchange="testLetter(this.form)" />
      </form>
   </body>
</html>
```

Each condition executes only the statements that apply to that particular condition, even if it takes several queries to find out what the entry is. You do not need to break out of the nested construction because when a true response is found, the relevant statement executes, and no other statements occur in the execution path to run.

Even if you understand how to construct a hair-raising nested construction, such as the one in Listing 31-1, the trickiest part is making sure that each left brace has a corresponding right brace. My technique for ensuring this pairing is to enter the right brace immediately after I type the left brace. I typically type the left brace, press Enter twice (once to open a free line for the next statement, once for the line that is to receive the right brace); tab, if necessary, to the same indentation as the line containing the left brace; and then type the right brace. Later, if I have to insert something indented, I just push down the right braces that I entered earlier. If I keep up this methodology throughout the process, the right braces appear at the desired indentation after I'm finished, even if the braces end up being dozens of lines below their original spot.

What's with the Formatting?

Indentation of the `if` construction and the further indentation of the statements executed on a true condition are not required by JavaScript. What you see here, however, is a convention that most JavaScript scripters follow. As you write the code in your text editor, you can use the Tab key to make each indentation level; some developers prefer using a setting in their editor that converts tabs to spaces, which guarantees that indentations are consistent across different editors. The browser ignores these tab characters (and/or spaces) when loading the HTML documents containing your scripts.

for

Conditional Expressions

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

While I'm showing you decision-making constructions in JavaScript, now is a good time to introduce a special type of expression that you can use in place of an `if. . .else` control structure for a common type of decision — the instance where you want to assign one of two values to a variable, depending on the outcome of some condition. The formal definition for the conditional expression is as follows:

```
variable = (condition) ? val1 : val2;
```

This expression means that if the Boolean result of the `condition` statement is true, JavaScript assigns `val1` to the variable; otherwise, it assigns `val2` to the variable. Like other instances of condition expressions, this one must also be written inside parentheses. The question mark is key here, as is the colon separating the two possible values.

A conditional expression, though not particularly intuitive or easy to read inside code, is very compact. Compare an `if. . .else` version of an assignment decision that follows

```
var collectorStatus;
if (CDCount > 500) {
   collectorStatus = "fanatic";
} else {
   collectorStatus = "normal";
}
```

with the conditional expression version:

```
var collectorStatus = (CDCount > 500) ? "fanatic" : "normal";
```

The latter saves a lot of code lines (although the internal processing is the same as that of an `if. . .else` construction). Of course, if your decision path contains more statements than just one setting the value of a variable, the `if. . .else` or `switch` construction is preferable. This shortcut, however, is a handy one to remember if you need to perform very binary actions, such as setting a true-or-false flag in a script.

Repeat (for) Loops

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

As you have seen in numerous examples throughout other chapters, the capability to cycle through every entry in an array or through every item of a form element is vital to many JavaScript scripts. Perhaps the most typical operation is inspecting a property of many similar items in search of a specific value, such as to determine which radio button in a group is selected. One JavaScript structure that allows for these repetitious excursions is the `for` loop, so-named after the keyword that begins the structure. Two other structures, called the `while` loop and `do-while` loop, are covered in the following sections.

The JavaScript `for` loop lets a script repeat a series of statements any number of times and includes an optional loop counter that can be used in the execution of the statements. The following is the formal syntax definition:

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

The three statements inside the parentheses (parameters to the `for` statement) play a key role in the way a `for` loop executes.

An initial expression in a `for` loop is executed one time, the first time the `for` loop begins to run. The most common application of the initial expression is to assign a name and starting value to a loop counter variable. Thus, seeing a `var` statement that both declares a variable name and assigns an initial value (generally `0` or `1`) to it is not uncommon. An example is

```
var i = 0;
```

You can use any variable name, but conventional usage calls for the letter `i`, which is short for *index*. If you prefer the word `counter` or another word that reminds you of what the variable represents, that's fine, too. In any case, the important point to remember about this statement is that it executes once at the outset of the `for` loop.

The second statement is a *condition*, precisely like the `condition` statement you saw in `if` constructions earlier in this chapter. When a loop-counting variable is established in the initial expression, the `condition` statement usually defines how high the loop counter should go before the looping stops. Therefore, the most common statement here is one that compares the loop counter variable against some fixed value — is the loop counter less than the maximum allowed value? If the condition is false at the start, the body of the loop is not executed. But if the loop does execute, then every time execution comes back around to the top of the loop, JavaScript reevaluates the condition to determine the current result of the expression. If the loop counter increases with each loop, eventually the counter value goes beyond the value in the `condition` statement, causing the `condition` statement to yield a Boolean value of `false`. The instant that happens, execution drops out of the `for` loop entirely.

The final statement, the *update expression*, is executed at the end of each loop execution — after all statements nested inside the `for` construction have run. Again, the loop counter variable can be a factor here. If you want the counter value to increase by one the next time through the loop (called incrementing the value), you can use the JavaScript operator that makes that happen: the `++` operator appended to the variable name. That task is the reason for the appearance of all those `i++` symbols in the `for` loops that you've seen already in this book. You're not limited to incrementing by one. You can increment by any multiplier you want or even drive a loop counter backward by decrementing the value (`i--`).

Now, take this knowledge and beef up the formal syntax definition with one that takes into account a typical loop-counting variable, `i`, and the common ways to use it:

```
//incrementing loop counter
for (var i = minValue; i <= maxValue; i++) {
   statements
}
//decrementing loop counter
for (var i = maxValue; i >= minValue; i--) {
   statements
}
```

In the top format, the variable, `i`, is initialized at the outset to a value equal to that of `minValue`. Variable `i` is immediately compared against `maxValue`. If `i` is less than or equal to `maxValue`, processing continues into the body of the loop. At the end of the loop, the update expression executes. In the top example, the value of `i` is incremented by `1`. Therefore, if `i` is initialized as `0`, then the first time through the loop, the `i` variable maintains that `0` value during the first execution of statements in the loop. The next time around, the variable has the value of `1`.

As you may have noticed in the formal syntax definition, each of the parameters to the `for` statement is optional. For example, the statements that execute inside the loop may control the value of the loop counter based on data that gets manipulated in the process. Therefore,

the update statement would probably interfere with the intended running of the loop. But I suggest that you use all three parameters until such time as you feel absolutely comfortable with their roles in the for loop. If you omit the condition statement, for instance, and you don't program a way for the loop to exit on its own, your script may end up in an infinite loop — which does your users no good.

Putting the loop counter to work

Despite its diminutive appearance, the i loop counter (or whatever name you want to give it) can be a powerful tool for working with data inside a repeat loop. For example, examine a version of the classic JavaScript function that creates an array while initializing entries to a value of 0:

```
// initialize array with n entries
function MakeArray(n) {
   this.length = n;
   for (var i = 1; i <= n; i++) {
      this[i] = 0;
   }
   return this;
}
```

The loop counter, i, is initialized to a value of 1, because you want to create an array of empty entries (with value 0) starting with the one whose index value is 1 (the zeroth entry is assigned to the length property) in the previous line. In the condition statement, the loop continues to execute as long as the value of the counter is less than or equal to the number of entries being created (n). After each loop, the counter increments by 1. In the nested statement that executes within the loop, you use the value of the i variable to substitute for the index value of the assignment statement:

```
this[i] = 0;
```

The first time the loop executes, the value expression evaluates to

```
this[1] = 0;
```

The next time, the expression evaluates to

```
this[2] = 0;
```

and so on, until all entries are created and stuffed with 0.

Recall the HTML page in Listing 30-2, where a user chooses a regional office from a select list (triggering a script to look up the manager's name and sales quota for that region). Because the regional office names are stored in an array, the page could be altered so that a script populates the select element's options from the array. That way, if there is ever a change to the alignment of regional offices, there need be only one change to the array of offices, and the HTML doesn't have to be modified. As a reminder, here is the definition of the regional offices array, created while the page loads:

```
var regionalOffices = new Array("New York", "Chicago", "Houston", "Portland");
```

A script inside the HTML form can be used to dynamically generate the select list as follows:

```
<script type="text/javaScript">
var elem = ""; // start assembling next part of page and form
elem += "<p>Select a regional office: ";
elem += "<select name='offices' onchange='getData(this.form)'>";
// build options list from array office names
```

```
for (var i = 0; i < regionalOffices.length; i++) {
   elem += "<option";   // option tags
   if (i == 0) {        // pre-select first item in list
      elem += " selected='selected'";
   }
   elem += ">" + regionalOffices[i];
}
elem += "</select></p>"; // close select item tag
document.write(elem);    // write element to the page
</script>
```

Notice one important point about the `condition` statement of the `for` loop: JavaScript extracts the `length` property from the array to be used as the loop counter boundary. From a code maintenance and stylistic point of view, this method is preferable to hard-wiring a value there. If the company added a new regional office, you would make the addition to the array "database," whereas everything else in the code would adjust automatically to those changes, including creating a longer pop-up menu in this case.

Notice, too, that the operator for the `condition` statement is less-than (<): The zero-based index values of arrays mean that the maximum index value we can use is one less than the actual count of items in the array. This is vital information, because the index counter variable (`i`) is used as the index to the `regionalOffices` array each time through the loop to read the string for each item's entry. You also use the counter to determine which is the first option, so that you can take a short detour (via the `if` construction) to add the `selected` attribute to the first option's definition.

The utility of the loop counter in `for` loops often influences the way you design data structures, such as two-dimensional arrays (see Chapter 30) for use as databases. Always keep the loop-counter mechanism in the back of your mind when you begin writing JavaScript script that relies on collections of data that you embed in your documents.

Breaking out of a loop

Some loop constructions perform their job as soon as a certain condition is met, at which point they have no further need to continue looping through the rest of the values in the loop counter's range. A common scenario for this is the cycling of a loop through an entire array in search of a single entry that matches some criterion. That criterion test is set up as an `if` construction inside the loop. If that criterion is met, you break out of the loop and let the script continue with the more meaningful processing of succeeding statements in the main flow. To accomplish that exit from the loop, use the `break` statement. The following schematic shows how the `break` statement may appear in a `for` loop:

```
for (var i = 0; i < array.length; i++) {
   if (array[i].property == magicValue) {
      statements that act on entry array[i]
      break;
   }
}
```

The `break` statement tells JavaScript to bail out of the nearest `for` loop (in case you have nested `for` loops). Script execution then picks up immediately after the closing brace of the `for` statement. The variable value of `i` remains whatever it was at the time of the break, so that you can use that variable later in the same script to access, say, that same array entry.

I use a construction similar to this in Chapter 22. There, the discussion of radio buttons demonstrates this construction, where, in Listing 22-8, you see a set of radio buttons whose `value` attributes contain the full names of four members of the Three Stooges. A function

uses a `for` loop to find out which button was selected and then uses that item's index value — after the for loop breaks out of the loop — to alert the user. Listing 31-2 shows the relevant function.

Listing 31-2: Breaking Out of a for Loop

```
function fullName(form) {
   for (var i = 0; i < form.stooges.length; i++) {
      if (form.stooges[i].checked) {
         break;
      }
   }
   alert("You chose " + form.stooges[i].value + ".");
}
```

In this case, breaking out of the `for` loop was for more than mere efficiency; the value of the loop counter (frozen at the break point) is used to summon a different property outside of the `for` loop. As of version 4 browsers, the `break` statement gained additional powers in cooperation with the new `label` feature of control structures. This subject is covered later in this chapter.

Directing loop traffic with continue

One other possibility in a `for` loop is that you may want to skip execution of the nested statements for just one condition. In other words, as the loop goes merrily on its way round and round, executing statements for each value of the loop counter, one value of that loop counter may exist for which you don't want those statements to execute. To accomplish this task, the nested statements need to include an `if` construction to test for the presence of the value to skip. When that value is reached, the `continue` command tells JavaScript to immediately skip the rest of the body, execute the `update` statement, and loop back around to the top of the loop (also skipping the `condition` statement part of the `for` loop's parameters).

To illustrate this construction, you create an artificial example that skips over execution when the counter variable is the superstitious person's unlucky `13`:

```
for (var i = 0; i <= 20; i++) {
   if (i == 13) {
      continue;
   }
   statements
}
```

In this example, the *`statements`* part of the loop executes for all values of `i` except `13`. The `continue` statement forces execution to jump to the `i++` part of the loop structure, incrementing the value of `i` for the next time through the loop. In the case of nested `for` loops, a `continue` statement affects the `for` loop in whose immediate scope the `if` construction falls. The `continue` statement was enhanced in version 4 browsers with the `label` feature of control structures, which is covered later in this chapter.

The while Loop

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The `for` loop is not the only kind of repeat loop you can construct in JavaScript. Another statement, called a `while` statement, sets up a loop in a slightly different format. Rather than providing a mechanism for modifying a loop counter, a `while` repeat loop assumes that your script statements will reach a condition that forcibly exits the repeat loop.

The basic syntax for a `while` loop is

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

The *`condition`* expression is the same kind that you saw in `if` constructions and in the middle parameter of the `for` loop. You introduce this kind of loop if some condition exists in your code (evaluates to `true`) before reaching this loop. The loop then performs some action, which affects that condition repeatedly until that condition becomes `false`. At that point, the loop exits, and script execution continues with statements after the closing brace. If the statements inside the `while` loop do not affect the values being tested in *`condition`*, your script never exits, and it becomes stuck in an infinite loop.

Many loops can be rendered with either the `for` or `while` loops. In fact, Listing 31-3 shows a `while` loop version of the `for` loop from Listing 31-2.

Listing 31-3: **A while Loop Version of Listing 31-2**

```
function fullName(form) {
   var i = 0;
   while (!form.stooges[i].checked) {
      i++;
   }
   alert("You chose " + form.stooges[i].value + ".");
}
```

One point you may notice is that if the condition of a `while` loop depends on the value of a loop counter, the scripter is responsible for initializing the counter prior to the `while` loop construction and managing its value within the `while` loop.

Should you need their powers, the `break` and `continue` control statements work inside `while` loops as they do in `for` loops. But because the two loop styles treat their loop counters and conditions differently, be extra careful (do lots of testing) when applying `break` and `continue` statements to both kinds of loops.

No hard-and-fast rules exist for which type of loop construction to use in a script. I generally use `while` loops only when the data or object I want to loop through is already a part of my script before the loop. In other words, by virtue of previous statements in the script, the values for any condition or loop counting (if needed) are already initialized. But if I need to cycle through an object's properties or an array's entries to extract some piece of data for use later in the script, I favor the `for` loop.

Another point of style, particularly with the `for` loop, is where a scripter should declare the `i` variable. Some programmers prefer to declare (or initialize if initial values are known) all variables in the opening statements of a script or function. That is why you tend to see a lot of `var` statements in those positions in scripts. If you have only one `for` loop in a function, for example, nothing is wrong with declaring and initializing the `i` loop counter in the initial expression part of the `for` loop (as demonstrated frequently in the previous sections). But if your function utilizes multiple `for` loops that reuse the `i` counter variable (that is, the loops

run completely independently of one another), then you can declare the `i` variable once at the start of the function and simply assign a new initial value to `i` in each `for` construction.

The do-while Loop

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

JavaScript brings you one more looping construction, called the `do-while` loop. The formal syntax for this construction is as follows:

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

An important difference distinguishes the `do-while` loop from the `while` loop. In the `do-while` loop, the statements in the construction always execute at least one time before the condition can be tested; in a `while` loop, the statements may never execute if the condition tested at the outset evaluates to `false`.

Use a `do-while` loop when you know for certain that the looped statements are free to run at least one time. If the condition may not be met the first time, use the `while` loop. For many instances, the two constructions are interchangeable, although only the `while` loop is compatible with all scriptable browsers.

Looping through Properties (for-in)

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

JavaScript includes a variation of the `for` loop, called a `for-in` loop, which has special powers of extracting the names and values of any object property currently in the browser's memory. The syntax looks like this:

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

The *object* parameter is not the string name of an object but a reference to the object itself. JavaScript delivers an object reference if you provide the name of the object as an unquoted string, such as `window` or `document`. Using the `var` variable, you can create a script that extracts and displays the range of properties for any given object.

Listing 31-4 shows a page containing a utility function that you can insert into your HTML documents during the authoring and debugging stages of designing a JavaScript-enhanced page. In the example, the current `window` object is examined and its properties are presented in the page (note that Safari 1.0 doesn't expose `window` object properties).

Listing 31-4: **Property Inspector Function**

```
<html>
  <head>
     <title></title>
     <script type="text/javascript">
     function showProps(obj,objName) {
```

```
            var result = "";
            for (var i in obj) {
               result += objName + "." + i + " = " + obj[i] + "<br />";
            }
            return result;
         }
         </script>
      </head>
      <body>
         <b>Here are the properties of the current window:</b>
         <p>
            <script type="text/javascript">
            document.write(showProps(window, "window"));
            </script>
         </p>
      </body>
</html>
```

For debugging purposes, you can revise the function slightly to display the results in an alert dialog box. Replace the `<br>` HTML tag with the `\n` carriage return character for a nicely formatted display in the alert dialog box. You can call this function from anywhere in your script, passing both the object reference and a string to it to help you identify the object after the results appear in an alert dialog box. If the `showProps()` function looks familiar to you, it is because it closely resembles the property inspector routines of The Evaluator (see Chapter 13). In Chapter 45 on the CD-ROM, you can see how to embed functionality of The Evaluator into a page under construction so that you can view property values while debugging your scripts.

The with Statement

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

A `with` statement enables you to preface any number of statements by advising JavaScript on precisely which object your scripts will be talking about, so that you don't have to use full, formal addresses to access properties or invoke methods of the same object. The formal syntax definition of the `with` statement is as follows:

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

The object reference is a reference to any valid object currently in the browser's memory. An example of this appears in Chapter 28's discussion of the `Math` object. By embracing several `Math`-encrusted statements inside a `with` construction, your scripts can call the properties and methods without having to make the object part of every reference to those properties and methods.

An advantage of the `with` structure is that it can make heavily object-dependent statements easier to read and understand. Consider this long version of a function that requires multiple calls to the same object (but different properties):

```
function seeColor(form) {
   newColor = (form.colorsList.options[form.colorsList.selectedIndex].text);
   return newColor;
}
```

with

Using the `with` structure, you can shorten the long statement:

```
function seeColor(form) {
   with (form.colorsList) {
      newColor = (options[selectedIndex].text);
   }
   return newColor;
}
```

When JavaScript encounters an otherwise unknown identifier inside a `with` statement, it tries to build a reference out of the object specified as its parameter and that unknown identifier. You cannot, however, nest `with` statements that build on one another. For instance, in the preceding example, you cannot have a `with` (`colorsList`) nested inside a `with` (`form`) statement and expect JavaScript to create a reference to options out of the two object names.

As clever as the `with` statement may seem, be aware that it introduces some inherent performance penalties in your script (because of the way the JavaScript interpreter must artificially generate references). You probably won't notice degradation with occasional use of this construction, but if it's used inside a loop that must iterate many times, processing speed will almost certainly be affected negatively.

Labeled Statements

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

Crafting multiple nested loops can sometimes be difficult when the final condition your script is looking for is met deep inside the nests. The problem is that the `break` or `continue` statement by itself has scope only to the nearest loop level. Therefore, even if you break out of the inner loop, the outer loop(s) continue to execute. If all you want to do is exit the function after the condition is met, a simple `return` statement performs the same job as some other languages' exit command. But if you also need some further processing within that function after the condition is met, you need the NN4+/IE4+/W3C facility that lets you assign labels to blocks of JavaScript statements. Your `break` and `continue` statements can then alter their scope to apply to a labeled block other than the one containing the statement.

A *label* is any identifier (that is, name starting with a letter and containing no spaces or odd punctuation other than an underscore) followed by a colon preceding a logical block of executing statements, such as an `if. . .then` or loop construction. The formal syntax looks like the following:

```
labelID:
   statements
```

For a `break` or `continue` statement to apply itself to a labeled group, the label is added as a kind of parameter to each statement, as in

```
break labelID;
continue labelID;
```

To demonstrate how valuable this can be in the right situation, Listing 31-5 contains two versions of the same nested loop construction. The goal of each version is to loop through two different index variables until both values equal the target values set outside the loop. When those targets are met, the entire nested loop construction should break off and continue processing afterward. To help you visualize the processing that goes on during the execution of the loops, the scripts output intermediate and final results to a textarea.

In the version without labels, when the targets are met, only the simple `break` statement is issued. This breaks the inner loop at that point, but the outer loop picks up on the next iteration.

By the time the entire construction has ended, a lot of wasted processing has gone on. Moreover, the values of the counting variables max themselves out, because the loops execute in their entirety several times after the targets are met.

But in the labeled version, the inner loop breaks out of the labeled outer loop as soon as the targets are met. Far fewer lines of code are executed, and the loop counting variables are equal to the targets, as desired. Experiment with Listing 31-5 by changing the `break` statements to `continue` statements. Then closely analyze the two results in the Results textarea to see how the two versions behave.

Listing 31-5: **Labeled Statements**

```
<html>
   <head>
      <title>Breaking Out of Nested Labeled Loops</title>
      <script type="text/javascript">
      var targetA = 2;
      var targetB = 2;
      var range = 5;
      function run1() {
         var out = document.forms[0].output;
         out.value = "Running WITHOUT labeled break\n";
         for (var i = 0; i <= range; i++) {
            out.value += "Outer loop #" + i + "\n";
            for (var j = 0; j <= range; j++) {
               out.value += "  Inner loop #" + j + "\n";
               if (i == targetA && j == targetB) {
                  out.value += "**BREAKING OUT OF INNER LOOP**\n";
                  break;
               }
            }
         }
         out.value += "After looping, i = " + i + ", j = " + j + "\n";
      }
      function run2() {
         var out = document.forms[0].output;
         out.value = "Running WITH labeled break\n";
         outerLoop:
         for (var i = 0; i <= range; i++) {
            out.value += "Outer loop #" + i + "\n";
            innerLoop:
            for (var j = 0; j <= range; j++) {
               out.value += "  Inner loop #" + j + "\n";
               if (i == targetA && j == targetB) {
                  out.value += "**BREAKING OUT OF OUTER LOOP**\n";
                  break outerLoop;
               }
            }
         }
         out.value += "After looping, i = " + i + ", j = " + j + "\n";
      }
      </script>
   </head>
   <body>
      <h1>Breaking Out of Nested Labeled Loops</h1>
```

Continued

Listing 31-5 *(continued)*

```
      <hr />
      <p>Look in the Results field for traces of these button scripts:</p>
      <form>
         <p><input type="button" value="Execute WITHOUT Label"
            onclick="run1()" /></p>
         <p><input type="button" value="Execute WITH Label"
            onclick="run2()" /></p>
         <p>Results:</p>
         <textarea name="output" rows="43" cols="60"></textarea>
      </form>
   </body>
</html>
```

The switch Statement

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

In some circumstances, a binary — true or false — decision path is not enough to handle the processing in your script. An `object` property or `variable` value may contain any one of several values, and a separate execution path is required for each one. In the past, the way to accommodate this was with a series of `if. . .else` constructions. The more conditions you must test, the less efficient the processing is, because each condition must be tested. Moreover, the sequence of clauses and braces can get very confusing.

In version 4 browsers, a control structure in use by many languages was introduced to JavaScript. The implementation is similar to that of Java and C, using the `switch` and `case` keywords. The basic premise is that you can create any number of execution paths based on the value of some expression. At the beginning of the structure, you identify what that expression is and then, for each execution path, assign a label matching a particular value.

The formal syntax for the `switch` statement is

```
switch (expression) {
case label1:
   statements
   [break]
case label2:
   statements
   [break]
...
[default:
   statements]
}
```

The *`expression`* parameter of the `switch` statement can evaluate to any string or number value. Labels are surrounded by quotes when the labels represent string values of the expression. Notice that the `break` statements are optional. A `break` statement forces the switch expression to bypass all other checks of succeeding labels against the expression value. Another option is the `default` statement, which provides a catchall execution path when the expression value does not match any of the `case` statement labels. If you'd rather not have any execution take place with a non-matching expression value, omit the default part of the construction.

To demonstrate the syntax of a working `switch` statement, Listing 31-6 provides the skeleton of a larger application of this control structure. The page contains two separate arrays of different product categories. Each product has its name and price stored in its respective array. A `select` list displays the product names. After a user chooses a product, the script looks up the product name in the appropriate array and displays the price.

The trick behind this application is the values assigned to each product in the select list. While the displayed text is the product name, the `value` attribute of each `<option>` tag is the array category for the product. That value is the expression used to decide which branch to follow. Notice, too, that I assign a label to the entire switch construction. The purpose of that is to let the deeply nested repeat loops for each case completely bail out of the switch construction (via a labeled `break` statement) whenever a match is made. You can extend this example to any number of product category arrays with additional `case` statements to match.

Listing 31-6: **The switch Construction in Action**

```
<html>
   <head>
      <title>Switch Statement and Labeled Break</title>
      <script type="text/javascript">
      // build two product arrays, simulating two database tables
      function product(name, price) {
         this.name = name;
         this.price = price;
      }
      var ICs = new Array();
      ICs[0] = new product("Septium 900MHz","$149");
      ICs[1] = new product("Septium Pro 1.0GHz","$249");
      ICs[2] = new product("Octium BFD 750MHz","$329");
      var snacks = new Array();
      snacks[0] = new product("Rays Potato Chips","$1.79");
      snacks[1] = new product("Cheezey-ettes","$1.59");
      snacks[2] = new product("Tortilla Flats","$2.29");

      // lookup in the 'table' associated with the product
      function getPrice(selector) {
         var chipName = selector.options[selector.selectedIndex].text;
         var outField = document.forms[0].cost;
         master:
         switch(selector.options[selector.selectedIndex].value) {
         case "ICs":
            for (var i = 0; i < ICs.length; i++) {
               if (ICs[i].name == chipName) {
                  outField.value = ICs[i].price;
                  break master;
               }
            }
            break;
         case "snacks":
            for (var i = 0; i < snacks.length; i++) {
               if (snacks[i].name == chipName) {
                  outField.value = snacks[i].price;
                  break master;
               }
            }
```

Continued

Listing 31-6 *(continued)*

```
            break;
         default:
            outField.value = "Not Found";
         }
      }
      </script>
   </head>
   <body>
      <b>Branching with the switch Statement</b>
      <hr />
      Select a chip for lookup in the chip price tables:
      <form>
         Chip:<select name="chips" onchange="getPrice(this)">
            <option></option>
            <option value="ICs">Septium 900MHz</option>
            <option value="ICs">Septium Pro 1.0GHz</option>
            <option value="ICs">Octium BFD 750MHz</option>
            <option value="snacks">Rays Potato Chips</option>
            <option value="snacks">Cheezey-ettes</option>
            <option value="snacks">Tortilla Flats</option>
            <option>Poker Chipset</option>
         </select>  Price:<input type="text" name="cost" size="10" />
      </form>
   </body>
</html>
```

Exception Handling

The subject of exception handling is relatively new to JavaScript. Formalized in Edition 3 of ECMA-262, parts of the official mechanism are implemented in IE5, with more complete implementations in IE6 and NN6, and of course in Mozilla and Safari.

Exceptions and errors

If you've done any scripting, you are certainly aware of JavaScript errors, whether they be from syntax errors in your code, or what are known as *runtime* errors — errors that occur while scripts are processing information. Ideally, a program should be aware of when an error occurs and handle it as gracefully as possible. This self-healing can prevent lost data (usually not a big problem in Web applications) and prevent users from seeing the ugliness of error messages. Chapter 16 covers the `onerror` event handler (and `window.onerror` property), which were early attempts at letting scripts gain a level of control over runtime errors. This event-driven mechanism works on a global level (that is, in the `window` object) and processes every error that occurs throughout the page. This event handler ends up being used primarily as a last-ditch defense against displaying any error message to the user and is a long way from what programmers consider to be exception handling.

In the English language, the term "exception" can mean the same as something out of the ordinary, or something abnormal. This definition seems quite distant from the word "error," which usually means a mistake. In the realm of programming languages, however, the two words tend to be used interchangeably, and the difference between the two depends primarily on one's point of view.

Consider, for example, a simple script whose job is to multiply numbers that the user enters into two text fields on the page. The script is supposed to display the results in a third text box. If the script contains no data entry validation, JavaScript will attempt to multiply whatever values are entered into the text boxes. If the user enters two numbers, JavaScript is smart enough to recognize that even though the `value` properties of the two input text fields are strings, the strings contain numbers that can be converted to number types for the proper multiplication. Without complaint, the product of the two numbers gets calculated and displayed into the results.

But what if the user types a letter into one of the text boxes? Again, without any entry validation in the script, JavaScript has a fixed way of responding to such a request: The result of the multiplication operation is the `NaN` (not a number) constant. If you are an untrained user, you have no idea what `NaN` means, but your experience with computers tells you that some kind of error has occurred. You may blame the computer or you may blame yourself.

To shift the point of view to the programmer, however, the script was designed to be run by a user who never makes a typing mistake, intentional or not. That, of course, is not very good programming practice. Users make mistakes. Therefore, anticipating user input that is not what would be expected is the programmer's job — input that is an exception to the rules your program wants to operate by. You must include some additional code that handles the exceptions gracefully so as to not confuse the user with unintelligible output and perhaps even help the user repair the input to get a result. This extra programming code handles the undesirable and erroneous input.

As it turns out, JavaScript and the W3C Document Object Model liberally mix terms of exception and error within the vocabulary used to handle exceptions. As you see shortly, an exception creates an *error object*, which contains information about the exception. It is safe to say that you can think of exceptions and errors as the same things.

The exception mechanism

Newcomers to JavaScript (or any programming environment, for that matter) may have a difficult time at first creating a mental model of how all this stuff runs within the context of the browser. It may be easy enough to understand how pages load and create object models, and how event handlers (or listeners in the W3C DOM terminology) cause script functions to run. But a lot of action also seems to be going on in the background. For example, the event object that is generated automatically with each event action (see Chapter 25) seems to sit "somewhere" while event handler functions run so that they can retrieve details about the event. After the functions finish their processing, the event object disappears, without even leaving behind a Cheshire Cat smile. Mysterious.

Browsers equipped for exception handling have more of this "stuff" running in the background, ready for your scripts when you need it. Because you have certainly viewed the details of at least one scripting error, you have already seen some of the exception-handling mechanism that is built into browsers. If a script error occurs, the browser creates in its memory an error object, whose properties contain details about the error. The precise details (described later in this chapter) vary from one browser brand to the next, but what you see in the error details readout is the default way the browser handles exceptions/errors. As browsers have matured, their makers have gone to great lengths to tone down the intrusion of script errors. For example in NN4+, errors appear in a separate JavaScript Console window (which must be invoked in NN4 by typing `javascript:` into the Location field; or opened directly via the Tools menu in NN6+ and Mozilla). In IE4+ for Windows, the status bar comes into play again, as the icon at the bottom-left corner turns into an alert icon: Double-clicking the icon displays more information about the error. MacIE users can turn off scripting error alerts altogether. Safari 1.0 doesn't divulge any script errors (see Chapter 45 on the CD-ROM to engage this feature).

True exception handling, however, goes further than just displaying error messages. It also provides a uniform way to let scripts guard against unusual occurrences. Ideally, the mechanism makes sure that *all* runtime errors get funneled through the same mechanism to help simplify the scripting of exception handling. The mechanism is also designed to be used intentionally as a way for your own code to generate errors in a uniform way so that other parts of your scripts can handle them quietly and intelligently. In other words, you can use the exception handling mechanism as a kind of "back channel" to communicate from one part of your scripts to another.

The JavaScript exception handling mechanism is built around two groups of program execution statements. The first group consists of the `try-catch-finally` statement triumvirate; the second group is the single `throw` statement.

Using try-catch-finally Constructions

The purpose of the `try-catch-finally` group of related statements is to provide a controlled environment in which script statements that may encounter runtime errors can run, such that if an exception occurs, your scripts can act upon the exception without alarming the rest of the browser's error mechanisms. Each of the three statements precedes a block of code in the following syntax:

```
try {
   statements to run
}
catch (errorInfo) {
   statements to run if exception occurs in try block
}
finally {
   statements to run whether or not an exception occurred [optional]
}
```

Each `try` block must be mated with a `catch` and/or `finally` block at the same nesting level, with no intervening statements. For example, a function can have a one-level `try-catch` construction inside it as follows:

```
function myFunc() {
   try {
      statements
   }
   catch (e) {
      statements
   }
}
```

But if there were another `try` block nested one level deeper, a balancing `catch` or `finally` block would also have to be present at that deeper level:

```
function myFunc() {
   try {
      statements
      try {
         statements
      }
      catch (e) {
         statements
      }
```

```
    }
    catch (e) {
        statements
    }
}
```

The statements inside the `try` block include statements that you believe are capable of generating a runtime error because of user input errors, the failure of some page component to load, or a similar error. The presence of the `catch` block prevents errors from appearing in the browser's regular script error reporting system (for example, the JavaScript Console of NN6+ and Mozilla).

An important term to know about exception handling of this type is *throw*. The convention is that when an operation or method call triggers an exception, it is said to "throw an exception." For example, if a script statement attempts to invoke a method of a string object, but that method does not exist for the object (perhaps you mistyped the method name), JavaScript throws an exception. Exceptions have names associated with them — a name that sometimes, but not always, reveals important information about the exception. In the mistyped method example just cited, the name of that exception is a `TypeError` (yet more evidence of how "exception" and "error" become intertwined).

The JavaScript language (in IE5+/NN6+/W3C) is not the only entity that can throw exceptions. The W3C DOM also defines categories of exceptions for DOM objects. For example, according to the Level 2 specification, the `appendChild()` method (see Chapter 15) can throw (or *raise*, in the W3C terminology) one of three exceptions:

Exception Name	*When Thrown*
`HIERARCHY_REQUEST_ERR`	If the current node is of a type that does not allow children of the type of the *`newChild`* node, or if the node to append is one of this node's ancestors
`WRONG_DOCUMENT_ERR`	If *`newChild`* was created from a different document than the one that created the current node
`NO_MODIFICATION_ALLOWED_ERR`	If the current node is read-only

Because the `appendChild()` method is capable of throwing exceptions, a JavaScript statement that invokes this method should ideally be inside a `try` block. If an exception is thrown, then script execution immediately jumps to the `catch` or `finally` block associated with the `try` block. Execution does not come back to the `try` block.

A `catch` block has special behavior. Its format looks similar to a function in a way, because the `catch` keyword is followed by a pair of parentheses and an arbitrary variable that is assigned a reference to the error object whose properties are filled by the browser when the exception occurs. One of the properties of that error object is the name of the error. Therefore, the code inside the `catch` block can examine the name of the error and perhaps include some branching code to take care of a variety of different errors that are caught.

To see how this construction may look in code, look at a hypothetical generic function whose job is to create a new element and append it to some other node. Both the type of element to be created and a reference to the parent node are passed as parameters. To take care of potential misuses of this function through the passage of improper parameter values, it includes extra error handling to treat all possible exceptions from the two DOM methods: `createElement()` and `appendChild()`. Such a function looks like Listing 31-7.

Listing 31-7: **A Hypothetical try-catch Routine**

```
// generic appender
function attachToEnd(theNode, newTag) {
   try {
      var newElem = document.createElement(newTag);
      theNode.appendChild(newElem);
   }
   catch (e) {
      switch (e.name) {
      case "INVALID_CHARACTER_ERR" :
         statements to handle this createElement() error
         break;
      case "HIERARCHY_REQUEST_ERR" :
         statements to handle this appendChild() error
         break;
      case "WRONG_DOCUMENT_ERR" :
         statements to handle this appendChild() error
         break;
      case "NO_MODIFICATION_ALLOWED_ERR" :
         statements to handle this appendChild() error
         break;
      default:
         statements to handle any other error
      }
      return false;
   }
   return true;
}
```

The single `catch` block in Listing 31-7 executes only if one of the statements in the `try` block throws an exception. The exceptions may be not only one of the four specific ones named in the `catch` block but also syntax or other errors that could occur inside the `try` block. That's why you have a last-ditch case to handle truly unexpected errors. Your job as scripter is to not only anticipate errors but also to provide clean ways for the exceptions to be handled, whether they be through judiciously worded alert dialog boxes or perhaps even some self-repair. For example, in the case of the invalid character error for `createElement()`, your script may attempt to salvage the data passed to the `attachToEnd()` function and reinvoke the method passing `theNode` value as-is and the repaired value originally passed to `newTag`. If your repairs were successful, the `try` block would execute without error and carry on with the user's being completely unaware that a nasty problem had been averted.

A `finally` block contains code that always executes after a `try` block, whether or not the `try` block succeeds without throwing an error. Unlike the `catch` block, a `finally` block does not receive an error object as a parameter, so it operates very much in the dark about what transpires inside the `try` block. If you include both `catch` and `finally` blocks after a `try` block, the execution path depends on whether an exception is thrown. If no exception is thrown, the `finally` block executes after the last statement of the `try` block runs. But if the `try` block throws an exception, program execution runs first to the `catch` block. After all processing within the `catch` block finishes, the `finally` block executes. In development environments that give programmers complete control over resources, such as memory allocation, a `finally` block may be used to delete some temporary items generated in the `try` block, whether or not an exception occurs in the `try` block. Currently, JavaScript has less need for that kind of maintenance, but you should be aware of the program execution possibilities of the `finally` block in the `try-catch-finally` context.

Real-life exceptions

The example shown in Listing 31-7 is a bit idealized. The listing assumes that the browser dutifully reports every W3C DOM exception precisely as defined in the formal specification. Unfortunately, that's not how it is (yet) in browsers through IE6 and NN7. Both browsers implement additional error naming conventions and layers between actual DOM exceptions and what gets reported with the error object at the time of the exception.

If you think these discrepancies make cross-browser exception handling difficult, you're right. Even simple errors are reported differently among the two major browser brands and the W3C DOM specification. Until the browsers exhibit a greater unanimity in exception reporting, the smoothest development road will be for those scripters who have the luxury of writing for one of the browser platforms, such as IE6 for Windows or NN7.

That said, however, one aspect of exception handling can still be used in both IE5+ and NN6+/W3C. You can take advantage of `try-catch` constructions to throw your own exceptions — a practice that is quite common in advanced programming environments.

Throwing Exceptions

The last exception handling keyword not covered yet — `throw` — makes it possible to utilize exception-handling facilities for your own management of processes, such as data entry validation. At any point inside a `try` block, you can manually throw an exception that gets picked up by the associated `catch` block. The details of the specific exception are up to you.

Syntax for the `throw` statement is as follows:

```
throw value;
```

The value you throw can be of any type, but good practice suggests that the value be an error object (described more fully later in this chapter). Whatever value you throw is assigned to the parameter of the `catch` block. Look at the following two examples. In the first, the value is a string message; in the second, the value is an error object.

Listing 31-8 presents one input text box for a number between 1 and 5. Clicking a button looks up a corresponding letter in an array and displays the letter in a second text box. The lookup script has two simple data validation routines to make sure the entry is a number and is in the desired range. Error checking here is done manually by script. If either of the error conditions occurs, `throw` statements force execution to jump to the `catch` block. The `catch` block assigns the incoming `string` parameter to the variable `e`. The design here assumes that the message being passed is text for an alert dialog box. Not only does a single `catch` block take care of both error conditions (and conceivably any others to be added later), but the `catch` block runs within the same variable scope as the function, so that it can use the reference to the input text box to focus and select the input text if there is an error.

Listing 31-8: **Throwing String Exceptions**

```
<html>
  <head>
     <title>Throwing a String Exception</title>
     <script type="text/javascript">
     var letters = new Array("A","B","C","D","E");
```

Continued

Listing 31-8 *(continued)*

```
      function getLetter(fld) {
         try {
            var inp = parseInt(fld.value, 10);
            if (isNaN(inp)) {
               throw "Entry was not a number.";
            }
            if (inp < 1 || inp > 5) {
               throw "Enter only 1 through 5.";
            }
            fld.form.output.value = letters[inp - 1];
         }
         catch (e)  {
            alert(e);
            fld.form.output.value = "";
            fld.focus();
            fld.select();
         }
      }
      </script>
   </head>
   <body>
      <h1>Throwing a String Exception</h1>
      <hr />
      <form>
         Enter a number from 1 to 5: <input type="text" name="input"
         size="5" /> <input type="button" value="Get Letter"
         onclick="getLetter(this.form.input)" /> Matching Letter is:<input
         type="text" name="output" size="5" />
      </form>
   </body>
</html>
```

The flaw with Listing 31-8 is that if some other kind of exception were thrown inside the `try` block, the value passed to the `catch` block would be an error object, not a string. The alert dialog box displayed to the user would be meaningless. Therefore, it is better to be uniform in your `throw-catch` constructions and pass an error object.

Listing 31-9 is an updated version of Listing 31-8, demonstrating how to create an error object that gets sent to the `catch` block via `throw` statements.

Listing 31-9: **Throwing an Error Object Exception**

```
<html>
   <head>
      <title>Throwing an Error Object Exception</title>
      <script type="text/javascript">
      var letters = new Array("A","B","C","D","E");
      function getErrorObj(msg) {
         var err = new Error(msg);
         return err;
      }
```

```
      function getLetter(fld) {
         try {
            var inp = parseInt(fld.value, 10);
            if (isNaN(inp)) {
               throw getErrorObj("Entry was not a number.");
            }
            if (inp < 1 || inp > 5) {
               throw getErrorObj("Enter only 1 through 5.");
            }
            fld.form.output.value = letters[inp - 1];
         }
         catch (e)  {
            alert(e.message);
            fld.form.output.value = "";
            fld.focus();
            fld.select();
         }
      }
      </script>
   </head>
   <body>
      <h1>Throwing an Error Object Exception</h1>
      <hr />
      <form>
         Enter a number from 1 to 5: <input type="text" name="input"
         size="5" /> <input type="button" value="Get Letter"
         onclick="getLetter(this.form.input)" /> Matching Letter is:<input
         type="text" name="output" size="5" />
      </form>
   </body>
</html>
```

The only difference to the `catch` block is that it now reads the `message` property of the incoming error object. This means that if some other exception is thrown inside the `try` block, the browser-generated message will be displayed in the alert dialog box.

In truth, however, the job really isn't complete. In all likelihood, if a browser-generated exception is thrown, the message in the alert dialog box won't mean much to the user. The error message will probably be some kind of syntax or type error — the kind of meaningless error message you often get from your favorite operating system. A better design is to branch the `catch` block so that "intentional" exceptions thrown by your code are handled through the alert dialog box messages you've put there, but other types are treated differently. To accomplish this, you can take over one of the other properties of the error object — `name` — so that your `catch` block treats your custom messages separately.

In Listing 31-10, the `getErrorObj()` function adds a custom value to the `name` property of the newly created error object. The name you assign can be any name, but you want to avoid exception names used by JavaScript or the DOM. Even if you don't know what all of those are, you can probably conjure up a suitably unique name for your error. Down in the `catch` block, a `switch` construction branches to treat the two classes of errors differently. In this simplified example, about the only possible problem other than the ones being trapped for explicitly in the `try` block would be some corruption to the page during downloading. Therefore, for this example, the branch for all other errors simply asks that the user reload the page and try again. The point is, however, that you can have as many classifications of custom and system errors as you want and handle them in a single `catch` block accordingly.

throw

Listing 31-10: **A Custom Object Exception**

```
<html>
   <head>
      <title>Throwing a Custom Error Object Exception</title>
      <script type="text/javascript">
      var letters = new Array("A","B","C","D","E");
      function getErrorObj(msg) {
         var err = new Error(msg);
         err.name = "MY_ERROR";
         return err;
      }
      function getLetter(fld) {
         try {
            var inp = parseInt(fld.value, 10);
            if (isNaN(inp)) {
               throw getErrorObj("Entry was not a number.");
            }
            if (inp < 1 || inp > 5) {
               throw getErrorObj("Enter only 1 through 5.");
            }
            fld.form.output.value = letters[inp - 1];
         }
         catch (e)  {
            switch (e.name) {
            case "MY_ERROR" :
               alert(e.message);
               fld.form.output.value = "";
               fld.focus();
               fld.select();
               break;
            default :
               alert("Reload the page and try again.");
            }
         }
      }
      </script>
   </head>
   <body>
      <h1>Throwing a Custom Error Object Exception</h1>
      <hr />
      <form>
         Enter a number from 1 to 5: <input type="text" name="input"
         size="5" /> <input type="button" value="Get Letter"
         onclick="getLetter(this.form.input)" /> Matching Letter is:<input
         type="text" name="output" size="5" />
      </form>
   </body>
</html>
```

If you want to see how the alternative branch of Listing 31-10 looks, copy the listing file from the CD-ROM to your hard disk and modify the last line of the `try` block so that one of the letters is dropped from the name of the array:

```
fld.form.output.value = letter[inp - 1];
```

This may simulate the faulty loading of the page. If you enter one of the allowable values, the reload alert appears, rather than the actual message of the error object: letter is undefined. Your users will thank you.

All that's left now on this subject are the details on the error object.

Error Object

Properties	*Methods*
`Error.prototype`	*`errorObject`*`.toString()`
`errorObject``.constructor`	
`errorObject``.description`	
`errorObject``.filename`	
`errorObject``.lineNumber`	
`errorObject``.message`	
`errorObject``.name`	
`errorObject.number`	

Syntax

Creating an error object:

```
var myError = new Error("message");
var myError = Error("message");
```

Accessing static `Error` object property:

```
Error.property
```

Accessing error object properties and methods:

```
errorObject.property | method([parameters])
```

Compatibility: WinIE5+, MacIE-, NN6+, Moz1+, Safari1+

About this object

An error object instance is created whenever an exception is thrown or when you invoke either of the constructor formats for creating an error object. Properties of the error object instance contain information about the nature of the error so that `catch` blocks can inspect the error and process error handling accordingly.

IE5 implemented an error object in advance of the ECMA-262 formal error object, and the IE5 version ended up having its own set of properties that are not part of the ECMA standard. Those proprietary properties are still part of IE5.5+, which includes the ECMA properties as well. NN6, on the other hand, started with the ECMA properties and adds two proprietary properties of its own. The browser uses these additional properties in its own script error reporting. The unfortunate bottom line for cross-browser developers is that no properties in common among all browsers support the error object. However, two common denominators (`name` and `message`) are between IE5.5+ and NN6+.

As described earlier in this chapter, you are encouraged to create an error object whenever you use the `throw` statement for your own error control. See the discussion surrounding Listing 31-9 about handling missing properties in IE.

Properties

constructor

(See `string.constructor` in Chapter 27)

description

Value: String. Read/Write
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `description` property contains a descriptive string that provides some level of detail about the error. For errors thrown by the browser, the description is the same text that appears in the script error dialog box in IE. Although this property continues to be supported, the `message` property is preferred.

Related Items: `message` property.

fileName
lineNumber

Value: String. Read/Write
Compatibility: WinIE-, MacIE-, NN6+, Moz-, Safari-

The NN6 browser uses the `fileName` and `lineNumber` properties of an error object for its own internal script error processing — these values appear as part of the error messages that are listed in the JavaScript Console. The `fileName` is the URL of the document causing the error; the `lineNumber` is the source code line number of the statement that threw the exception. These properties are exposed to JavaScript, as well, so that your error processing may use this information if it is meaningful to your application.

See the discussion of the `window.error` property in Chapter 16 for further ideas on how to use this information for bug reporting from users.

Related Items: `window.error` property.

message

Value: String. Read/Write
Compatibility: WinIE5.5+, MacIE-, NN6+, Moz1+, Safari1+

The `message` property contains a descriptive string that provides some level of detail about the error. For errors thrown by the browser, the message is the same text that appears in the script error dialog box in IE and the JavaScript Console in NN6+. By and large, these messages are more meaningful to scripters than to users. Unfortunately, there are no standards for the wording of a message for a given error. Therefore, it is hazardous at best to use the message content in a `catch` block as a means of branching to handle particular kinds of errors. You may get by with this approach if you are developing for a single browser platform, but you have no assurances that the text of a message for a particular exception may not change in future browser versions.

Custom messages for errors that your code explicitly throws can be in user-friendly language if you intend to display such messages to users. See Listings 31-8 through 31-10 for examples of this usage.

Related Items: `description` property.

`name`

Value: String. Read/Write
Compatibility: WinIE5.5+, MacIE-, NN6+, Moz1+, Safari1+

The `name` property generally contains a word that identifies the type of error that has been thrown. The most general kind of error (and the one that is created via the `new Error()` constructor) has a name `Error`. But JavaScript errors can be of several varieties: `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, and `URIError`. Some of these error types are not necessarily intended for exposure to scripters (they're used primarily in the inner workings of the JavaScript engine), but some browsers do expose them. Unfortunately, there are some discrepancies as to the specific name supplied to this property for script errors.

When JavaScript is being used in a W3C-compatible browser, some DOM exception types are returned via the `name` property. But browsers frequently insert their own error types for this property, and, as is common in this department, little uniformity exists among browser brands.

For custom exceptions that your code explicitly throws, you can assign names as you want. As shown in Listings 31-9 and 31-10, this information can assist a `catch` block in handling multiple categories of errors.

Related Items: `message` property.

`number`

Value: Number. Read/Write
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

IE5+ assigns unique numbers to each error description or message. The value of the `number` property must be massaged somewhat to retrieve a meaningful error description. Following is an example of how you must apply binary arithmetic to an error number to arrive at a meaningful result:

```
var errNum = errorObj.number & x0FFFF;
```

To find out what an error number means, just look it up on Microsoft's Developer Network (MSDN) site at `http://msdn.microsoft.com/library/en-us/script56/html/js56jsmscRunTimeErrors.asp`.

Related Items: `description` property.

Methods

`toString()`

Returns: String (see text).
Compatibility: WinIE5+, MacIE-, NN6+, Moz1+, Safari1+

The `toString()` method for an error object should return a string description of the error. In IE5+, however, the method returns a reference to the very same error object. In NN6+, the method returns the `message` property string, preceded by the string `Error:` (with a space after the colon). Most typically, if you want to retrieve a human-readable expression of an error object, read its `message` (or, in IE5+, `description`) property.

Related Items: `message` property.

✦ ✦ ✦

CHAPTER 32

JavaScript Operators

In This Chapter

Understanding operator categories

Exploring the role of operators in script statements

Recognizing operator precedence

JavaScript is rich in *operators*: words and symbols in expressions that perform operations on one or two values to arrive at another value. Any value on which an operator performs some action is called an *operand*. An expression may contain one operand and one operator (called a *unary operator*) or two operands separated by one operator (called a *binary operator*). Many of the same symbols are used in a variety of operators. The combination and order of those symbols are what distinguish their powers.

The vast majority of JavaScript operators have been in the language since the very beginning. But, as you may expect from an evolving language, some new entries have been added to the lexicon. In the rest of this chapter, compatibility charts typically govern an entire category of operator. If there are version anomalies for a particular operator within a category, they are covered in the text.

Operator Categories

To help you grasp the range of JavaScript operators, I group them into seven categories. I assign a wholly untraditional name to the second group — but a name that I believe better identifies its purpose in the language. Table 32-1 shows the operator types.

Table 32-1: JavaScript Operator Categories

Type	*What It Does*
Comparison	Compares the values of two operands, deriving a result of either true or false (used extensively in condition statements for `if...else` and `for` loop constructions)
Connubial	Joins together two operands to produce a single value that is a result of an arithmetical or other operation on the two

Continued

Table 32-1 *(continued)*

Type	*What It Does*
Assignment	Stuffs the value of the expression of the right-hand operand into a variable name on the left-hand side, sometimes with minor modification, as determined by the operator symbol
Boolean	Performs Boolean arithmetic on one or two Boolean operands
Bitwise	Performs arithmetic or column-shifting actions on the binary (base-2) representations of two operands
Object	Helps scripts examine the heritage and capabilities of a particular object before they need to invoke the object and its properties or methods
Miscellaneous	A handful of operators that have special behaviors

Any expression that contains an operator evaluates to a value of some kind. Sometimes the operator changes the value of one of the operands; other times the result is a new value. Even this simple expression

```
5 + 5
```

shows two integer operands joined by the addition operator. This expression evaluates to `10`. The operator is what provides the instruction for JavaScript to follow in its never-ending drive to evaluate every expression in a script.

Doing an equality comparison on two operands that, on the surface, look very different is not at all uncommon. JavaScript doesn't care what the operands look like—only how they evaluate. Two very dissimilar-looking values can, in fact, be identical when they are evaluated. Thus, an expression that compares the equality of two values, such as

```
fred == 25
```

does, in fact, evaluate to `true` if the variable `fred` has the number `25` stored in it from an earlier statement.

Comparison Operators

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Anytime you compare two values in JavaScript, the result is a Boolean `true` or `false` value. You have a wide selection of comparison operators to choose from, depending on the kind of test you want to apply to the two operands. Table 32-2 lists all comparison operators.

Table 32-2: JavaScript Comparison Operators

Syntax	*Name*	*Operand Types*	*Results*
==	Equals	All	Boolean
!=	Does not equal	All	Boolean
===	Strictly equals	All	Boolean (IE4+, NN4+, W3C)
!==	Strictly does not equal	All	Boolean (IE4+, NN4+, W3C)
>	Is greater than	All	Boolean

Syntax	*Name*	*Operand Types*	*Results*
>=	Is greater than or equal to	All	Boolean
<	Is less than	All	Boolean
<=	Is less than or equal to	All	Boolean

For numeric values, the results are the same as those you'd expect from your high school algebra class. Some examples follow, including some that may not be obvious.

```
10 == 10            // true
10 == 10.0          // true
9 != 10             // true
9 > 10              // false
9.99 <= 9.98        // false
```

Strings can also be compared on all of these levels:

```
"Fred" == "Fred"    // true
"Fred" == "fred"    // false
"Fred" > "fred"     // false
"Fran" < "Fred"     // true
```

To calculate string comparisons, JavaScript converts each character of a string to its ASCII value. Each letter, beginning with the first of the left-hand operator, is compared to the corresponding letter in the right-hand operator. With ASCII values for uppercase letters being less than those of their lowercase counterparts, an uppercase letter evaluates to being less than its lowercase equivalent. JavaScript takes case-sensitivity very seriously.

Values for comparison can also come from object properties or values passed to functions from event handlers or other functions. A common string comparison used in data-entry validation is the one that sees if the string has anything in it:

```
form.entry.value != ""    // true if something is in the field
```

Equality of Disparate Data Types

For all versions of JavaScript before 1.2, when your script tries to compare string values consisting of numerals and real numbers (for example, `"123" == 123` or `"123" != 123`), JavaScript anticipates that you want to compare apples to apples. Internally it does some data type conversion that does not affect the data type of the original values (for example, if the values are in variables). But the entire situation is more complex, because other data types, such as objects, need to be dealt with. Therefore, prior to JavaScript 1.2, the rules of comparison are as shown in Table 32-3.

Table 32-3: Equality Comparisons for JavaScript 1.0 and 1.1

Operand A	*Operand B*	*Internal Comparison Treatment*
Object reference	Object reference	Compare object reference evaluations
Any data type	Null	Convert nonnull to its object type and compare against null
Object reference	String	Convert object to string and compare strings
String	Number	Convert string to number and compare numbers

Comparison Operators

The logic to what goes on in equality comparisons from Table 32-3 requires a lot of forethought on the scripter's part, because you have to be very conscious of the particular way data types may or may not be converted for equality evaluation (even though the values themselves are not converted). In this situation, supplying the proper conversion where necessary in the comparison statement is best. This ensures that what you want to compare — for example, the string versions of two values or the number versions of two values — is compared, rather than leaving the conversion up to JavaScript.

Backward-compatible conversion from a number to string entails concatenating an empty string to a number:

```
var a = "09";
var b = 9;
a == "" + b;  // result: false, because "09" does not equal "9"
```

For converting strings to numbers, you have numerous possibilities. The simplest is subtracting zero from a numeric string:

```
var a = "09";
var b = 9;
a-0 == b;  // result: true because number 9 equals number 9
```

You can also use the `parseInt()` and `parseFloat()` functions to convert strings to numbers:

```
var a = "09";
var b = 9;
parseInt(a, 10) == b;  // result: true because number 9 equals number 9
```

To clear up the ambiguity of JavaScript's equality internal conversions, JavaScript 1.2 uses two more operators to force the equality comparison to be extremely literal in its comparison. The strictly equals (`===`) and strictly does not equal (`!==`) operators compare both the data type and value. The only time the `===` operator returns true is if the two operands are of the same data type (for example, both are numbers) and the same value. Therefore, no number is ever automatically equal to a string version of that same number. Data and object types must match before their values are compared.

JavaScript 1.2 also provides some convenient global functions for converting strings to numbers and vice versa: `String()` and `Number()`. To demonstrate these methods, the following examples use the `typeof` operator to show the data type of expressions using these functions:

```
typeof 9;             // result: number
type of String(9);    // result: string
type of "9";          // result: string
type of Number("9");  // result: number
```

None of these functions alters the data type of the value being converted. But the value of the function is what gets compared in an equality comparison:

```
var a = "09";
var b = 9;
a == String(b);   // result: false, because "09" does not equal "9"
typeof b;         // result: still a number
Number(a) == b;   // result: true, because 9 equals 9
typeof a;         // result: still a string
```

This discussion should impress upon you the importance of considering data types when testing the equality of two values.

Connubial Operators

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Connubial operators is my terminology for those operators that join two operands to yield a value related to the operands. Table 32-4 lists the connubial operators in JavaScript.

Table 32-4: JavaScript Connubial Operators

Syntax	*Name*	*Operand Types*	*Results*
+	Plus	Integer, float, string	Integer, float, string
-	Minus	Integer, float	Integer, float
*	Multiply	Integer, float	Integer, float
/	Divide	Integer, float	Integer, float
%	Modulo	Integer, float	Integer, float
++	Increment	Integer, float	Integer, float
--	Decrement	Integer, float	Integer, float
+*val*	Positive	Integer, float, string	Integer, float
-val	Negation	Integer, float, string	Integer, float

The four basic arithmetic operators for numbers are straightforward. The plus operator also works on strings to join them together, as in

```
"Howdy " + "Doody" // result = "Howdy Doody"
```

In object-oriented programming terminology, the plus sign is considered *overloaded*, meaning that it performs a different action depending on its context. Remember, too, that string concatenation does not do anything on its own to monitor or insert spaces between words. In the preceding example, the space between the names is part of the first string.

Modulo arithmetic is helpful for those times when you want to know if one number divides evenly into another. You used it in an example in Chapter 31 to figure out if a particular year was a leap year. Although some other leap year considerations exist for the turn of each century, the math in the example simply checked whether the year was evenly divisible by four. The result of the modulo math is the remainder of division of the two values: When the remainder is 0, one divides evenly into the other. Here are some samples of years evenly divisible by four:

```
2002 % 4    // result = 2
2003 % 4    // result = 3
2004 % 4    // result = 0 (Bingo! Leap year!)
```

Thus, I used this modulo operator in a condition statement of an `if. . .else` structure:

```
var howMany = 0;
today = new Date();
var theYear = today.getYear();
if (theYear % 4 == 0) {
    howMany = 29;
} else {
    howMany = 28;
}
```

Connubial Operators

Some other languages offer an operator that results in the integer part of a division problem solution: integral division, or `div`. Although JavaScript does not have an explicit operator for this behavior, you can re-create it reliably if you know that your operands are always positive numbers. Use the `Math.floor()` or `Math.ceil()` methods with the division operator, as in

```
Math.floor(4/3);    // result = 1
```

In this example, `Math.floor()` works only with values greater than or equal to `0`; `Math.ceil()` works with values less than `0`.

The increment operator (++) is a *unary* operator (only one operand) and displays two different behaviors, depending on the side of the operand on which the symbols lie. Both the increment and decrement (`--`) operators can be used in conjunction with assignment operators, which I cover next.

As its name implies, the increment operator increases the value of its operand by one. But in an assignment statement, you have to pay close attention to precisely when that increase takes place. An assignment statement stuffs the value of the right operand into a variable on the left. If the ++ operator is located in front of the right operand (prefix), the right operand is incremented before the value is assigned to the variable; if the ++ operator is located after the right operand (postfix), the previous value of the operand is sent to the variable before the value is incremented. Follow this sequence to get a feel for these two behaviors:

```
var a = 10;     // initialize a to 10
var z = 0;      // initialize z to zero
z = a;          // a = 10, so z = 10
z = ++a;        // a becomes 11 before assignment, so a = 11 and z becomes 11
z = a++;        // a is still 11 before assignment, so z = 11; then a becomes 12
z = a++;        // a is still 12 before assignment, so z = 12; then a becomes 13
```

The decrement operator behaves the same way, except that the value of the operand decreases by one. Increment and decrement operators are used most often with loop counters in `for` and `while` loops. The simpler ++ or `--` symbology is more compact than reassigning a value by adding `1` to it (such as, `z = z + 1` or `z += 1`). Because these are unary operators, you can use the increment and decrement operators without an assignment statement to adjust the value of a counting variable within a loop:

```
function doNothing() {
   var i = 1;
   while (i < 20) {
      ++i;
   }
   alert(i); // breaks out at i = 20
}
```

The last pair of connubial operators are also unary operators (operating on one operand). Both the positive and negation operators can be used as shortcuts to the `Number()` global function, converting a string operand consisting of number characters to a number data type. The string operand is not changed, but the operation returns a value of the number type, as shown in the following sequence:

```
var a = "123";
var b = +a;       // b is now 123
typeof a;         // result: string
typeof b;         // result: number
```

The negation operator (`-val`) has additional power. By placing a minus sign in front of any numeric value (no space between the symbol and the value), you instruct JavaScript to evaluate a positive value as its corresponding negative value, and vice versa. The operator does

not change the operand's value, but the expression returns the modified value. The following example provides a sequence of statements to demonstrate:

```
var x = 2;
var y = 8;
var z = -x;          // z equals -2, but x still equals 2
z = -(x + y);        // z equals -10, but x still equals 2 and y equals 8
z = -x + y;          // z equals 6, but x still equals 2 and y equals 8
```

To negate a Boolean value, see the Not (!) operator in the discussion of Boolean operators.

Assignment Operators

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Assignment statements are among the most common statements you write in your JavaScript scripts. These statements appear everywhere you copy a value or the results of an expression into a variable for further manipulation of that value.

You assign values to variables for many reasons, even though you could probably use the original values or expressions several times throughout a script. Here is a sampling of reasons why you should assign values to variables:

- ✦ Variable names are usually shorter
- ✦ Variable names can be more descriptive
- ✦ You may need to preserve the original value for later in the script
- ✦ The original value is a property that cannot be changed
- ✦ Invoking the same method several times in a script is not efficient

Newcomers to scripting often overlook the last reason. For instance, if a script is writing HTML to a new document, it's more efficient to assemble the string of large chunks of the page into one variable before invoking the `document.write()` method to send that text to the document. This approach is more efficient than literally sending out one line of HTML at a time with multiple `document.writeln()` method statements. Table 32-5 shows the range of assignment operators in JavaScript.

Table 32-5: JavaScript Assignment Operators

Syntax	*Name*	*Example*	*Means*
=	Equals	x = y	x = y
+=	Add by value	x += y	x = x + y
-=	Subtract by value	x -= y	x = x - y
*=	Multiply by value	x *= y	x = x * y
/=	Divide by value	x /= y	x = x / y
%=	Modulo by value	x %= y	x = x % y
<<=	Left shift by value	x <<= y	x = x << y
>=	Right shift by value	x >= y	x = x > y
>>=	Zero fill by value	x >>= y	x = x >> y

Continued

Table 32-5 *(continued)*

Syntax	*Name*	*Example*	*Means*
`>>>=`	Right shift by value	`x >>>= y`	`x = x >>> y`
`&=`	Bitwise `and` by value	`x &= y`	`x = x & y`
`|=`	Bitwise `or` by value	`x |= y`	`x = x | y`
`^=`	Bitwise XOR by value	`x ^= y`	`x = x ^ y`

As clearly demonstrated in the top group (see "Bitwise Operators" later in the chapter for information on the bottom group), assignment operators beyond the simple equals sign can save some characters in your typing, especially when you have a series of values that you're trying to bring together in subsequent statements. You've seen plenty of examples in previous chapters, where you used the add-by-value operator (+=) to work wonders with strings as you assemble a long string variable that you eventually send to a `document.write()` method. Look at this variation of a segment of Listing 30-3, where you could use JavaScript to create the HTML content of a `select` element on-the-fly:

```
var elem = ""; // start assembling next part of page and form
elem += "<p>Select a regional office: ";
elem += "<select name='offices' onchange='getData(this.form)'>";
// build options list from array office names
for (var i = 0; i < regionalOffices.length; i++) {
   elem += "<option";         // option tags
   if (i == 0) {              // pre-select first item in list
      elem += " selected='selected'";
   }
   elem += ">" + regionalOffices[i];
}
elem += "</select></p>";    // close select item tag
document.write(elem);       // write element to the page
```

The script segment starts with a plain equals assignment operator to initialize the `elem` variable as an empty string. In many of the succeeding lines, you use the add-by-value operator to tack additional string values onto whatever is in the `elem` variable at the time. Without the add-by-value operator, you are forced to use the plain equals assignment operator for each line of code to concatenate new string data to the existing string data. In that case, the first few lines of code look as shown:

```
var elem = ""; // start assembling next part of page and form
elem = elem + "<p>Select a regional office: ";
elem = elem + "<select name='offices' onchange='getData(this.form)'>";
```

Within the `for` loop, the repetition of `elem +` makes the code very difficult to read, trace, and maintain. These enhanced assignment operators are excellent shortcuts that you should use at every turn.

Boolean Operators

Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

Because a great deal of programming involves logic, it is no accident that the arithmetic of the logic world plays an important role. You've already seen dozens of instances where programs make all kinds of decisions based on whether a statement or expression is the Boolean

value `true` or `false`. What you haven't seen much of yet is how to combine multiple Boolean values and expressions — a quality that scripts with slightly above average complexity may need to have in them.

In the various condition expressions required throughout JavaScript (such as in an `if` construction), the condition that the program must test for may be more complicated than, say, whether a variable value is greater than a certain fixed value or whether a field is not empty. Look at the case of validating a text field entry for whether the entry contains all the numbers that your script may want. Without some magical JavaScript function to tell you whether or not a string consists of all numbers, you have to break apart the entry character by character and examine whether each character falls within the range of 0 through 9. But that examination actually comprises two tests: You can test for any character whose ASCII value is less than `0` or greater than `9`. Alternatively, you can test whether the character is greater than or equal to `0` and is less than or equal to `9`. What you need is the bottom-line evaluation of both tests.

Boolean math

That's where the wonder of Boolean math comes into play. With just two values — `true` and `false` — you can assemble a string of expressions that yield Boolean results and then let Boolean arithmetic figure out whether the bottom line is `true` or `false`.

But you don't add or subtract Boolean values the same way you add or subtract numbers. Instead, you use one of three JavaScript Boolean operators at your disposal. Table 32-6 shows the three operator symbols. In case you're unfamiliar with the characters in the table, the symbols for the Or operator are created by typing Shift-backslash.

Table 32-6: JavaScript Boolean Operators

Syntax	*Name*	*Operands*	*Results*
&&	And	Boolean	Boolean
||	Or	Boolean	Boolean
!	Not	One Boolean	Boolean

Using Boolean operators with Boolean operands gets tricky if you're not used to it, so I have you start with the simplest Boolean operator: Not. This operator requires only one operand. The Not operator precedes any Boolean value to switch it back to the opposite value (from `true` to `false`, or from `false` to `true`). For instance:

```
!true         // result = false
!(10 > 5)     // result = false
!(10 < 5)     // result = true
!(document.title == "Flintstones")    // result = true
```

As shown here, enclosing the operand of a Not expression inside parentheses is always a good idea. This forces JavaScript to evaluate the expression inside the parentheses before flipping it around with the Not operator.

The And (`&&`) operator joins two Boolean values to reach a `true` or `false` value based on the results of both values. This brings up something called a *truth table,* which helps you visualize all the possible outcomes for each value of an operand. Table 32-7 is a truth table for the And operator.

Table 32-7: Truth Table for the And Operator

Left Operand	*And Operator*	*Right Operand*	*Result*
True	&&	True	True
True	&&	False	False
False	&&	True	False
False	&&	False	False

Only one condition yields a `true` result: Both operands must evaluate to `true`. Which side of the operator a `true` or `false` value lives doesn't matter. Here are examples of each possibility:

```
5 > 1 && 50 > 10    // result = true
5 > 1 && 50 < 10    // result = false
5 < 1 && 50 > 10    // result = false
5 < 1 && 50 < 10    // result = false
```

In contrast, the Or (`||`) operator is more lenient about what it evaluates to `true`. The reason is that if one or the other (or both) operands is `true`, the operation returns `true`. The Or operator's truth table is shown in Table 32-8.

Table 32-8: Truth Table for the Or Operator

Left Operand	*Or Operator*	*Right Operand*	*Result*
True	||	True	True
True	||	False	True
False	||	True	True
False	||	False	False

Therefore, if a `true` value exists on either side of the operator, a `true` value is the result. Take the previous examples and swap the And operators with Or operators so that you can see the Or operator's impact on the results:

```
5 > 1 || 50 > 10    // result = true
5 > 1 || 50 < 10    // result = true
5 < 1 || 50 > 10    // result = true
5 < 1 || 50 < 10    // result = false
```

Only when both operands are `false` does the Or operator return `false`.

Boolean operators at work

Applying Boolean operators to JavaScript the first time just takes a little time and some sketches on a pad of paper to help you figure out the logic of the expressions. Earlier I talked about using a Boolean operator to see whether a character fell within a range of ASCII values for data-entry validation. Listing 32-1 is a function discussed in more depth in Chapter 43 on the CD-ROM. This function accepts any string and sees whether each character of the string has an ASCII value less than `0` or greater than `9` — meaning that the input string is not a number.

Listing 32-1: Is the Input String a Number?

```
function isNumber(inputStr) {
   for (var i = 0; i < inputStr.length; i++) {
      var oneChar = inputStr.substring(i, i + 1);
      if (oneChar < "0" || oneChar > "9") {
         alert("Please make sure entries are numerals only.");
         return false;
      }
   }
   return true;
}
```

Combining a number of JavaScript powers to read individual characters (substrings) from a `string` object within a `for` loop, the statement that you're interested in is the condition of the `if` construction:

```
(oneChar < "0" || oneChar > "9")
```

In one condition statement, you use the Or operator to test for both possibilities. If you check the Or truth table (Table 32-8), you see that this expression returns `true` if either one or both tests returns `true`. If that happens, the rest of the function alerts the user about the problem and returns a `false` value to the calling statement. Only if both tests within this condition evaluate to `false` for all characters of the string does the function return a `true` value.

From the simple Or operator, I go to the extreme, where the function checks — in one `condition` statement — whether a number falls within several numeric ranges. The script in Listing 32-2 comes from the array lookup application in Chapter 50 (on the CD-ROM), in which a user enters the first three digits of a U.S. Social Security number.

Listing 32-2: Is a Number within Discontiguous Ranges?

```
// function to determine if value is in acceptable range for this application
function inRange(inputStr) {
   num = parseInt(inputStr)
   if (num < 1 || (num > 586 && num < 596) || (num > 599 && num < 700) ||
      num > 728) {
      alert("Sorry, the number you entered is not part of our database.  Try
         another three-digit number.");
      return false;
   }
   return true;
}
```

By the time this function is called, the user's data entry has been validated enough for JavaScript to know that the entry is a number. Now the function must check whether the number falls outside of the various ranges for which the application contains matching data. The conditions that the function tests here are whether the number is

- ✦ Less than 1
- ✦ Greater than 586 and less than 596 (using the And operator)

- ✦ Greater than 599 and less than 700 (using the And operator)
- ✦ Greater than 728

Each of these tests is joined by an Or operator. Therefore, if any one of these conditions proves `true`, the whole `if` condition is `true`, and the user is alerted accordingly.

The alternative to combining so many Boolean expressions in one `condition` statement would be to nest a series of `if` constructions. But such a construction requires not only a great deal more code but also much repetition of the alert dialog box message for each condition that could possibly fail. The combined Boolean condition is, by far, the best way to go.

Bitwise Operators

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

For scripters, bitwise operations are an advanced subject. Unless you're dealing with external processes on CGIs or the connection to Java applets, it's unlikely that you will use bitwise operators. Experienced programmers who concern themselves with more specific data types (such as long integers) are quite comfortable in this arena, so I simply provide an explanation of JavaScript capabilities. Table 32-9 lists JavaScript bitwise operators.

Table 32-9: JavaScript's Bitwise Operators

Operator	*Name*	*Left Operand*	*Right Operand*
&	Bitwise And	Integer value	Integer value
|	Bitwise Or	Integer value	Integer value
^	Bitwise XOR	Integer value	Integer value
~	Bitwise Not	(None)	Integer value
<<	Left shift	Integer value	Shift amount
>>	Right shift	Integer value	Shift amount
>>>	Zero fill right shift	Integer value	Shift amount

The numeric value operands can appear in any of the JavaScript language's three numeric literal bases (decimal, octal, or hexadecimal). As soon as the operator has an operand, the value is converted to binary representation (32 bits long). For the first three bitwise operations, the individual bits of one operand are compared with their counterparts in the other operand. The resulting value for each bit depends on the operator:

- ✦ **Bitwise And:** `1` if both digits are `1`
- ✦ **Bitwise Or:** `1` if either digit is `1`
- ✦ **Bitwise Exclusive Or:** `1` if only one digit is a `1`

Bitwise Not, a unary operator, inverts the value of every bit in the single operand. The bitwise shift operators operate on a single operand. The second operand specifies the number of positions to shift the value's binary digits in the direction of the arrows of the operator symbols.

Example

For example, the left shift (`<<`) operator has the following effect:

```
4 << 2 // result = 16
```

The reason for this shifting is that the binary representation for decimal 4 is 00000100 (to eight digits, anyway). The left shift operator instructs JavaScript to shift all digits two places to the left, giving the binary result 00010000, which converts to 16 in decimal format. If you're interested in experimenting with these operators, use The Evaluator (Chapter 13) to evaluate sample expressions for yourself. More advanced books on C and C++ programming are also of help.

Object Operators

The next group of operators concern themselves with objects (including native JavaScript, DOM, and custom objects) and data types. Most of these have been implemented after the earliest JavaScript browsers, so each one has its own compatibility rating.

delete

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

Array objects do not contain a method to remove an element from the collection, nor do custom objects offer a method to remove a property. You can always empty the data in an array item or property by setting its value to an empty string or `null`, but the array element or property remains in the object. With the `delete` operator, you can completely remove the element or property.

There is special behavior about deleting an array item that you should bear in mind. If your array uses numeric indices, a deletion of a given index removes that index value from the total array but without collapsing the array (which would alter index values of items higher than the deleted item).

Example

For example, consider the following simple dense array:

```
var oceans = new Array("Atlantic", "Pacific", "Indian","Arctic");
```

This kind of array automatically assigns numeric indices to its entries for addressing later in constructions, such as `for` loops:

```
for (var i = 0; i < oceans.length; i++) {
   if (oceans[i] == form.destination.value) {
      statements
   }
}
```

If you then issue the statement

```
delete oceans[2];
```

the array undergoes significant changes. First, the third element is removed from the array. Note that the length of the array does not change. Even so, the index value `(2)` is removed from the array, such that schematically the array looks like the following:

```
oceans[0] = "Atlantic";
oceans[1] = "Pacific";
oceans[3] = "Arctic";
```

If you try to reference `oceans[2]` in this collection, the result is `undefined`.

The `delete` operator works best on arrays that have named indices. Your scripts will have more control over the remaining entries and their values, because they don't rely on what could be a missing entry of a numeric index sequence.

One aspect of this deletion action that JavaScript doesn't provide is absolute control over memory utilization. All garbage collection is managed by the JavaScript interpreter engine, which tries to recognize when items occupying memory are no longer needed, at which time the unused browser's application memory may be recovered. But you cannot force the browser to perform its garbage collection task.

in

Compatibility: WinIE5.5+, MacIE-, NN6+, Moz1+, Safari1+

The `in` operator lets a script statement inspect an object to see if it has a named property or method. The operand to the left of the operator is a string reference to the property or method (just the method name, without parentheses); the operand to the right of the operator is the object being inspected. If the object knows the property or method, the expression returns `true`. Thus, you can use the `in` operator in expressions used for conditional expressions.

Example

You can experiment with this operator in The Evaluator (Chapter 13). For example, to prove that the `write()` method is implemented for the `document` object, the expression you type into the top text box of The Evaluator is:

```
"write" in document
```

But compare the implementation of the W3C DOM `document.defaultView` property in IE5.5+ and modern W3C browsers:

```
"defaultView" in document
```

In NN6+, Mozilla, and Safari, the result is `true`, while in IE5.5 and IE6, the result is `false`.

Having this operator around for conditional expressions lets you go much beyond simple object detection for branching code. For example, if you intend to use `document.defaultView` in your script, you can make sure that the property is supported before referencing it (assuming your users all have browsers that know the `in` operator).

instanceof

Compatibility: WinIE5+, MacIE-, NN6+, Moz1+, Safari1+

The `instanceof` operator lets a script test whether an object is an instance of a particular JavaScript native object or DOM object. The operand to the left side of the operator is the value under test; the value to the right of the operand is a reference to the root class from which the value is suspected of being constructed.

For native JavaScript classes, the kinds of object references to the right of the operator include such static objects as `Date`, `String`, `Number`, `Boolean`, `Object`, `Array`, and `RegExp`. You sometimes need to be mindful of how native JavaScript classes can sometimes be children of other native classes, which means that a value may be an instance of two different static objects.

Example

For example, consider the following sequence (which you can follow along in The Evaluator):

```
a = new Array(1,2,3);
a instanceof Array;
```

The second statement yields a result of `true`, because the `Array` constructor was used to generate the object. But the JavaScript `Array` is, itself, an instance of the root `Object` object. Therefore both of the following statements evaluate to `true`:

```
a instanceof Object;
Array instanceof Object;
```

new

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Most JavaScript core objects have constructor functions built into the language. To access those functions, you use the `new` operator along with the name of the constructor. The function returns a reference to the object instance, which your scripts can then use to get and set properties or invoke object methods. For example, creating a `new` date object requires invoking the `Date` object's constructor, as follows:

```
var today = new Date();
```

Some object constructor functions require parameters to help define the object. Others, as in the case of the `Date` object, can accept a number of different parameter formats, depending on the format of date information you have to set the initial object. The `new` operator can be used with the following core language objects:

JavaScript 1.0	*JavaScript 1.1*	*JavaScript 1.2*	*JavaScript 1.5*
`Date`	`Array`	`RegExp`	`Error`
`Object`	`Boolean`		
(Custom object)	`Function`		
	`Image`		
	`Number`		
	`String`		

this

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

JavaScript includes an operator that allows script statements to refer to the very object in which they are located. The self-referential operator is `this`.

The most common application of the `this` operator is in event handlers that pass references of themselves to functions for further processing, as in

```
<input type="text" name="entry" onchange="process(this)" />
```

A function receiving the value assigns it to a variable that can be used to reference the sender, its properties, and its methods.

Example

Because the `this` operator references an object, that object's properties can be exposed with the aid of the operator. For example, to send the `value` property of a text input object to a

function, the `this` operator stands in for the current object reference and appends the proper syntax to reference the `value` property:

```
<input type="text" name="entry" onchange="process(this.value)" />
```

The `this` operator also works inside other objects, such as custom objects. When you define a constructor function for a custom object, using the `this` operator to define properties of the object and assign values to those properties is common practice. Consider the following example of an object creation sequence:

```
function bottledWater(brand, ozSize, flavor) {
   this.brand = brand;
   this.ozSize = ozSize;
   this.flavor = flavor;
}
var myWater = new bottledWater("Crystal Springs", 16, "original");
```

When the new object is created via the constructor function, the `this` operators define each property of the object and then assign the corresponding incoming value to that property. Using the same names for the properties and parameter variables is perfectly fine and makes the constructor easy to maintain.

By extension, if you assign a function as an object's property (to behave as a method for the object), the `this` operator inside that function refers to the object invoking the function, offering an avenue to the object's properties. For example, if I add the following function definition and statement to the `myWater` object created just above, the function can directly access the `brand` property of the object:

```
function adSlogan() {
   return "Drink " + this.brand + ", it's wet and wild!";
}
myWater.getSlogan = adSlogan;
```

When a statement invokes the `myWater.getSlogan()` method, the object invokes the `adSlogan()` function, but all within the context of the `myWater` object. Thus, the `this` operator applies to the surrounding object, making the brand property available via the `this` operator (`this.brand`).

Miscellaneous Operators

The final group of operators doesn't fit into any of the previous categories, but they are no less important.

,

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The comma operator indicates a series of expressions that are to be evaluated in left-to-right sequence. Most typically, this operator is used to permit multiple variable initializations. For example, you can combine the declaration of several variables in a single `var` statement, as follows:

```
var name, address, serialNumber;
```

Another situation where you could use this operator is within the expressions of a `for` loop construction. In the following example, two different counting variables are initialized and incremented at different rates. When the loop begins, both variables are initialized at zero

(they don't have to be, but this example starts that way); for each subsequent trip through the loop, one variable is incremented by one, while the other is incremented by 10:

```
for (var i=0, j=0; i < someLength; i++, j+10) {
   ...
}
```

Don't confuse the comma operator with the semicolon delimiter between statements.

? :

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

The conditional operator is a shortcut way of expressing an `if. . .else` conditional construction covered in Chapter 31. This operator is typically used in concert with an assignment operator to assign one of two values to a variable based on the result of a condition expression. The formal syntax for the conditional operator is:

```
condition ? expressionIfTrue : expressionIfFalse
```

If used with an assignment operator, the syntax is:

```
var = condition ? expressionIfTrue : expressionIfFalse;
```

No matter how you use the operator, the important point to remember is that an expression that contains this operator evaluates to one of the two expressions following the question mark symbol. In truth, either expression could invoke any JavaScript, including calling other functions or even nesting further conditional operators within one of the expressions to achieve the equivalent of nested `if. . .else` constructions. To assure proper resolution of nested conditionals, surround inner expressions with parentheses to make sure that they evaluate before the outer expression evaluates. As an example, the following statement assigns one of three strings to a variable depending on the date within a month:

```
var monthPart = (dateNum <= 10) ? "early" : ((dateNum <= 20) ?
   "middle" : "late");
```

When the statement is evaluated, the inner conditional expression at the right of the first colon is evaluated, returning either `middle` or `late`; then the outer conditional expression is evaluated, returning either `early` or the result of the inner conditional expression.

typeof

Compatibility: WinIE3+, MacIE3+, NN3+, Moz1+, Safari1+

Unlike most other operators, which are predominantly concerned with arithmetic and logic, the unary `typeof` operator defines the kind of value to which a variable or expression evaluates. Typically, this operator is used to identify whether a variable value is one of the following types: `number`, `string`, `boolean`, `object`, `function`, or `undefined`.

Example

Having this investigative capability in JavaScript is helpful because variables cannot only contain any one of those data types but can change their data type on the fly. Your scripts may need to handle a value differently based on the value's type. The most common use of the `typeof` property is as part of a condition. For example:

```
if (typeof myVal == "number") {
   myVal = parseInt(myVal);
}
```

typeof

The evaluated value of the `typeof` operation is, itself, a string.

`void`

Compatibility: WinIE3+, MacIE3+, NN3+, Moz1+, Safari1+

In all scriptable browsers you can use the `javascript:` pseudo-protocol to supply the parameter for `href` and `src` attributes in HTML tags, such as links. In the process, you have to be careful that the function or statement being invoked by the URL does not return or evaluate to any values. If a value comes back from such an expression, then that value or sometimes the directory of the client's hard disk often replaces the page content. To avoid this possibility, use the `void` operator in front of the function or expression being invoked by the `javascript:` URL.

Example

The best way to use this construction is to place the operator before the expression or function and separate them by a space, as in

```
javascript: void doSomething();
```

On occasion, you may have to wrap the expression inside parentheses after the `void` operator. Using parentheses is necessary only when the expression contains operators of a lower precedence than the `void` operator (see the following section, "Operator Precedence"). But don't automatically wrap all expressions in parentheses, because some browsers can experience problems with these. Even so, it is common practice to assign the following URL to the `href` attribute of an `a` link whose `onclick` event handler does all of the work:

```
href="javascript: void (0)"
```

The `void` operator makes sure the function or expression returns no value that the HTML attribute can use. Such a link's `onclick` event handler should also inhibit the natural behavior of a clicked link (for example, by evaluating to `return false`).

Operator Precedence

When you start working with complex expressions that hold a number of operators (for example, Listing 32-2), knowing the order in which JavaScript evaluates those expressions is vital. JavaScript assigns different priorities or weights to types of operators in an effort to achieve uniformity in the way it evaluates complex expressions.

In the following expression

```
10 + 4 * 5 // result = 30
```

JavaScript uses its precedence scheme to perform the multiplication before the addition — regardless of where the operators appear in the statement. In other words, JavaScript first multiplies `4` by `5` and then adds that result to `10` to get a result of `30`. That may not be the way you want this expression to evaluate. Perhaps your intention was to add the `10` and `4` first and then to multiply that sum by `5`. To make that happen, you have to override JavaScript's natural operator precedence. To do that, you must use parentheses to enclose an operator with lower precedence. The following statement shows how you adjust the previous expression to make it behave differently:

```
(10 + 4) * 5 // result = 70
```

That one set of parentheses has a great impact on the outcome. Parentheses have the highest precedence in JavaScript, and if you nest parentheses in an expression, the innermost set evaluates first.

For help in constructing complex expressions, refer to Table 32-10 for JavaScript's operator precedence. My general practice: When in doubt about complex precedence issues, I build the expression with lots of parentheses according to the way I want the internal expressions to evaluate.

Table 32-10: JavaScript Operator Precedence

Precedence Level	*Operator*	*Notes*
1	`()`	From innermost to outermost
	`[]`	Array index value
	`function()`	Any remote function call
2	`!`	Boolean Not
	`~`	Bitwise Not
	`-`	Negation
	`++`	Increment
	`--`	Decrement
	`new`	
	`typeof`	
	`void`	
	`delete`	Delete array or object entry
3	`*`	Multiplication
	`/`	Division
	`%`	Modulo
4	`+`	Addition
	`-`	Subtraction
5	`<<`	Bitwise shifts
	`>`	
	`>>`	
6	`<`	Comparison operators
	`<=`	
	`>`	
	`>=`	
7	`==`	Equality
	`!=`	
8	`&`	Bitwise And
9	`^`	Bitwise XOR
10	`|`	Bitwise Or
11	`&&`	Boolean And
12	`||`	Boolean Or
13	`?`	Conditional expression

Continued

Table 32-10 *(continued)*

Precedence Level	*Operator*	*Notes*
14	=	Assignment operators
	+=	
	-=	
	*=	
	/=	
	%=	
	<<=	
	>=	
	>>=	
	&=	
	^=	
	|=	
15	,	Comma (parameter delimiter)

This precedence scheme is devised to help you avoid being faced with two operators from the same precedence level that often appear in the same expression. When it happens (such as with addition and subtraction), JavaScript begins evaluating the expression from left to right.

One related fact involves a string of Boolean expressions strung together for a `condition` statement (see Listing 32-2). JavaScript follows what is called *short-circuit evaluation*. As the nested expressions are evaluated left to right, the fate of the entire condition can sometimes be determined before all expressions are evaluated. Anytime JavaScript encounters an And operator, if the left operand evaluates to `false`, the entire expression evaluates to `false` without JavaScript's even bothering to evaluate the right operand. For an Or operator, if the left operand is `true`, JavaScript short-circuits that expression to `true`. This feature can trip you up if you don't perform enough testing on your scripts: If a syntax error or other error exists in a right operand, and you fail to test the expression in a way that forces that right operand to evaluate, you may not know that a bug exists in your code. Users of your page, of course, will find the bug quickly. Do your testing to head bugs off at the pass.

Notice, too, that all math and string concatenation is performed prior to any comparison operators. This enables all expressions that act as operands for comparisons to evaluate fully before they are compared.

The key to working with complex expressions is to isolate individual expressions and to try them out by themselves, if you can. See additional debugging tips in Chapter 45 on the CD-ROM.

✦ ✦ ✦

CHAPTER 33

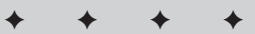
Functions and Custom Objects

In This Chapter

Creating function blocks

Passing parameters to functions

Creating your own objects



By now, you've seen dozens of JavaScript functions in action and probably have a pretty good feel for the way they work. This chapter provides the `function` object specification and delves into the fun prospect of creating objects in your JavaScript code. (That includes objects that have properties and methods, just like the big boys.)

Function Object

Properties	*Methods*	*Event Handlers*
`arguments`	`apply()`	
`arity`	`call()`	
`caller`	`toString()`	
`constructor`	`valueOf()`	
`length`		
`prototype`		

Syntax

Creating a function object:

```
function functionName([arg1,...[,argN]]) {
   statement(s)
}
var funcName = new Function(["argName1",...[,"argNameN"],
   "statement1;...[;statementN]"])
object.eventHandlerName = function([arg1,...[,argN]])
{statement(s)}
```

Accessing function object properties and methods:

```
functionObject.property | method([parameters])
```

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

About this object

JavaScript accommodates what other languages might call procedures, subroutines, and functions all in one type of structure: the *custom function*. A function may return a value (if programmed to do so with the `return` keyword), but it does not have to return any value. Except for JavaScript code that executes as the document loads, all deferred processing takes place in functions.

While you can create functions that are hundreds of lines long, I recommend you break up longer processes into shorter functions. Among the reasons for doing so: smaller chunks are easier to write and debug; building blocks make it easier to visualize the entire script; you can make functions generalizable and reusable for other scripts; and other parts of the script or other open frames can use the functions.

Learning how to write good, reusable functions takes time and experience. But the earlier you understand the importance of this concept, the more you will be on the lookout for good examples in other people's scripts on the Web.

Creating functions

The standard way of defining a function in your script means following a simple pattern and then filling in the details. The formal syntax definition for a function is:

```
function functionName( [arg1] ... [, argN]) {
   statement(s)
}
```

The task of assigning a function name helps you determine the precise scope of activity of the function. If you find that you can't reduce the planned task for the function to a simple one- to three-word name (which is then condensed into one contiguous sequence of characters for the *functionName*), perhaps you're asking the function to do too much. A better idea may be to break the job into two or more functions. As you start to design a function, be on the lookout for functions that you can call from the one you're writing. If you find yourself copying and pasting lines of code from one part of a function to another because you're performing the same operation in different spots within the function, it may be time to break that segment out into its own function.

You can also create what is called an *anonymous function* using the `new Function()` constructor. In reality, you assign a name to this "anonymous" function as follows:

```
var funcName = new Function(["argName1",...[,"argNameN"],
   "statement1;...[;statementN]"]);
```

This other way of building a function is particularly helpful when your scripts need to create a function after a document loads. All the components of a function are present in this definition. Each function parameter name is supplied as a string value, separated from each other by commas. The final parameter string consists of the statements that execute whenever the function is called. Separate each JavaScript statement with a semicolon, and enclose the entire sequence of statements inside quotes, as in the following:

```
var willItFit = new Function("width","height",
   "var sx = screen.availWidth; var sy = screen.availHeight;
   return (sx >= width && sy >= height)");
```

The `willItFit()` function takes two parameters; the body of the function defines two local variables (`sx` and `sy`) and then returns a Boolean value of `true` if the incoming parameters are smaller than the local variables. In traditional form, this function is defined as follows:

```
function willItFit(width, height) {
   var sx = screen.availWidth;
   var sy = screen.availHeight;
   return (sx >= width && sy >= height);
}
```

Once this function exists in the browser's memory, you can invoke it like any other function:

```
if (willItFit(400,500)) {
   statements to load image
}
```

One last function creation format is available in NN4+, IE4+, and W3C DOM browsers. This advanced technique, called a *lambda expression,* provides a shortcut for creating a reference to an anonymous function (truly anonymous because the function has no name that you can reference later). The common application of this technique is to assign function references to event handlers when the NN event object also must be passed. The following is an example of how to assign an anonymous function to an `onchange` event handler for a form control:

```
document.forms[0].age.onchange = function(event)
{isNumber(document.forms[0].age)}
```

Nesting functions

NN4+, IE4+, and W3C browsers also provide for nesting functions inside one another. In all prior scripting, each function definition is defined at the global level whereby every function is exposed and available to all other scripting. With nested functions, you can encapsulate the exposure of a function inside another and make that nested function private to the enclosing function. Of course I don't recommend reusing names in this fashion, but you can create nested functions with the same name inside multiple global level functions, as the following skeletal structure shows:

```
function outerA() {
   statements
   function innerA() {
      statements
   }
   statements
}
function outerB() {
   statements
   function innerA() {
      statements
   }
   function innerB() {
      statements
   }
   statements
}
```

A good time to apply a nested function is when a sequence of statements need to be invoked in multiple places within a large function but those statements have meaning only within the context of the larger function. In other words, rather than break out the repeated sequence as a separate global function, you keep it all within the scope of the larger function.

You can access a nested function only from statements in its containing function (and in any order). Moreover, all variables defined in the outer function (including parameter variables) are accessible to the inner function; but variables defined in an inner function are not accessible to

the outer function. See the section, "Variable scope: Globals and locals" later in this chapter for details on how variables are visible to various components of a script.

Function parameters

The function definition requires a set of parentheses after the *functionName*. If the function does not rely on any information arriving with it when invoked, the parentheses can be empty. But when some kind of data is arriving with a call to the function, you need to assign names to each parameter. Virtually any kind of value can be a parameter: strings, numbers, Booleans, and even complete object references such as a form or form element. Choose names for these variables that help you remember the content of those values; also, avoid reusing existing object names as variable names because it's easy to get confused when objects and variables with the same name appear in the same statements. You must avoid using JavaScript keywords (including the reserved words listed in Appendix B) and any global variable name defined elsewhere in your script. (See more about global variables in the following sections.)

JavaScript is forgiving about matching the number of parameters in the function definition with the number of parameters passed along from the calling statement. If you define a function with three parameters and the calling statement specifies only two, the third parameter variable value in that function is assigned a `null` value. For example:

```
function oneFunction(a, b, c) {
   statements
}
oneFunction("George","Gracie");
```

In the preceding example, the values of `a` and `b` inside the function are `"George"` and `"Gracie"`, respectively; the value of `c` is `null`.

At the opposite end of the spectrum, JavaScript also doesn't balk if you send more parameters from the calling statement than the number of parameter variables specified in the function definition. In fact, the language includes a mechanism — the `arguments` property — that you can add to your function to gather any extraneous parameters that should read your function.

Properties

`arguments`

Value: Array of arguments. Read-Only
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

When a function receives parameter values from the statement that invokes the function, those parameter values are silently assigned to the `arguments` property of the function object. This property is an array of the values, with each parameter value assigned to a zero-based index entry in the array — whether or not parameters are defined for it. You can find out how many parameters are sent by extracting *functionName*`.arguments.length`. For example, if four parameters are passed, *functionName*`.arguments.length` returns `4`. Then, you can use array notation (*functionName*`.arguments[`*i*`]`) to extract the values of any parameter(s) you want.

Theoretically, you never have to define parameter variables for your functions by extracting the desired arguments array entry instead. Well-chosen parameter variable names, however, are much more readable, so I recommend them over the `arguments` property in most cases. But you may run into situations in which a single function definition needs to handle multiple calls to the function when each call may have a different number of parameters. The function knows how to handle any arguments over and above the ones given names as parameter variables.

Example

See Listings 33-1 and 33-2 for a demonstration of both the `arguments` and `caller` properties.

arity

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

See the discussion of the `length` property later in this chapter.

caller

Value: Function object reference. Read-Only
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari-

When one function invokes another, a chain is established between the two primarily so that a returned value knows where to go. Therefore, a function invoked by another maintains a reference to the function that called it. Such information is automatically stored in a function object as the `caller` property. This relationship reminds me a bit of a subwindow's `opener` property, which points to the window or frame responsible for the subwindow's creation. The value is valid only while the called function is running at the request of another function; when a function isn't running, its `caller` property is `null`.

The value of the `caller` property is a reference to a function object, so you can inspect its `arguments` and `caller` properties (in case it was called by yet another function). Thus, a function can look back at a calling function to see what values it was passed.

The *`functionName`*`.caller` property reveals the contents of an entire function definition if the current function was called from another function (including an event handler). If the call for a function comes from a regular JavaScript statement not originating from inside a function, the *`functionName`*`.caller` property is `null`.

Example

To help you grasp all that these two properties yield, study Listing 33-1.

Listing 33-1: A Function's arguments and caller Properties

```
<html>
  <head>
     <title></title>
     <script type="text/javascript">
     function hansel(x,y) {
        var args = hansel.arguments;
        document.write("<p>hansel.caller is " + hansel.caller + "<br />");
        document.write("hansel.arguments.length is " +
           hansel.arguments.length + "<br />");
        for (var i = 0; i < args.length; i++) {
           document.write("argument " + i + " is " + args[i] + "<br />");
        }
        document.write("<\/p>");
     }
```

Continued

Listing 33-1 *(continued)*

```
      function gretel(x,y,z) {
         today = new Date();
         thisYear = today.getFullYear();
         hansel(x,y,z,thisYear);
      }
      </script>
   </head>
   <body>
      <script type="text/javascript">
      hansel(1, "two", 3);
      gretel(4, "five", 6, "seven");
      </script>
   </body>
</html>
```

When you load this page, the following results appear in the browser window (although the `caller` property values show `undefined` for Safari):

```
hansel.caller is null
hansel.arguments.length is 3
argument 0 is 1
argument 1 is two
argument 2 is 3

hansel.caller is function gretel(x, y, z) { today = new Date(); thisYear =
today.getFullYear(); hansel(x, y, z, thisYear); }
hansel.arguments.length is 4
argument 0 is 4
argument 1 is five
argument 2 is 6
argument 3 is 2004 (or whatever the current year is)
```

As the document loads, the `hansel()` function is called directly in the body script. It passes three arguments, even though the `hansel()` function defines only two. The `hansel.arguments` property picks up all three arguments just the same. The main body script then invokes the `gretel()` function, which, in turn, calls `hansel()` again. But when `gretel()` makes the call, it passes four parameters. The `gretel()` function picks up only three of the four arguments sent by the calling statement. It also inserts another value from its own calculations as an extra parameter to be sent to `hansel()`. The `hansel.caller` property reveals the entire content of the `gretel()` function, whereas `hansel.arguments` picks up all four parameters, including the year value introduced by the `gretel()` function.

constructor

(See `string.constructor` in Chapter 27)

length

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

As the `arguments` property of a function proves, JavaScript is very forgiving about matching the number of parameters passed to a function with the number of parameter variables defined

for the function. But a script can examine the `length` property of a function object to see precisely how many parameter variables are defined for a function. A reference to the property starts with the function name representing the object. For example, consider the following function definition shell:

```
function identify(name, rank, serialNum) {
   ...
}
```

A script statement anywhere outside of the function can read the number of parameters with the reference:

```
identify.length
```

The value of the property in the preceding example is `3`. The `length` property supercedes the NN-only `arity` property.

prototype

(See `Array.prototype` in Chapter 30)

Methods

apply([*thisObj*[, *argumentsArray*]])
call([*thisObj*[, *arg1*[, *arg2*[,...*argN*]]]])

Returns: Nothing.
Compatibility: WinIE5.5+, MacIE-, NN6+, Moz1+, Safari1+

The `apply()` and `call()` methods of a function object invoke the function. This may seem redundant to the normal way in which script statements invoke functions by simply naming the function, following it with parentheses, passing parameters, and so on. The difference with these methods is that you can invoke the function if your script has only a reference to the function. For example, if your script defines a function via the `new Function()` constructor (or other anonymous shortcut supported by the browser), you receive a reference to the function as a result of the constructor. To invoke the function later using only that reference (presumably preserved in a global variable), use either the `apply()` or `call()` method. Both of these methods achieve the same result, but choosing one method over the other depends on the form in which the function's parameters are conveyed (more about that in a moment).

The first parameter of both methods is a reference to the object that the function treats as the current object. For garden-variety functions defined in your script, use the keyword `this`, which means that the function's context becomes the current object (just like a regular function). In fact, if there are no parameters to be sent to the function, you can omit parameters to both methods altogether.

The object reference comes into play when the function being invoked is one that is normally defined as a method to a custom object. (I cover some of these concepts later in this chapter, so you may need to return here after you are familiar with custom objects.)

Example

Consider the following code that generates a custom object and assigns a method to the object to display an alert about properties of the object:

```
// function to be invoked as a method from a 'car' object
function showCar() {
   alert(this.make + " : " + this.color);
```

```
}
// 'car' object constructor function
function car(make, color) {
   this.make = make;
   this.color = color;
   this.show = showCar;
}
// create instance of a 'car' object
var myCar = new car("Ford", "blue");
```

The normal way of getting the `myCar` object to display an alert about its properties is:

```
myCar.show();
```

At that point, the `showCar()` function runs, picking up the current `car` object as the context for the `this` references in the function. In other words, when the `showCar()` function runs as a method of the object, the function treats the object as the "current object."

With the `call()` or `apply()` methods, however, you don't have to bind the `showCar()` function to the `myCar` object. You can omit the statement in the `car()` constructor that assigns the `showCar` function to a method name for the object. Instead, a script can invoke the `showCar()` method and instruct it to treat `myCar` as the current object:

```
showCar.call(myCar);
```

The `showCar()` function operates just as before, and the object reference in the `call()` method's first parameter slot is treated as the current object for the `showCar()` function.

As for succeeding parameters, the `apply()` method's second parameter is an array of values to be passed as parameters to the current function. The order of the values must match the order of parameter variables defined for the function. The `call()` method, on the other hand, enables you to pass individual parameters in a comma-delimited list. Your choice depends on how the parameters are carried along in your script. If they're already in array form, use the `apply()` method; otherwise, use the `call()` method. The (ECMA) recommended way to invoke a function through this mechanism when no parameters need to be passed is via the `call()` method.

`toString()`
`valueOf()`

Returns: String.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

Scripts rarely, if ever, summon the `toString()` and `valueOf()` methods of a function object. They work internally to allow debugging scripts to display a string version of the function definition. For example, when you enter the name of a function defined in The Evaluator (see Chapter 13) into the top text box, JavaScript automatically converts the function to a string so that its "value" can be displayed in the Results box. Using these methods or parsing the text they return has little, if any, practical application.

Function Application Notes

Understanding the ins and outs of JavaScript functions is the key to successful scripting, especially for complex applications. Additional topics covered in this chapter include the ways to invoke functions, variable scope in and around functions, recursion, and the design of reusable functions.

Invoking functions

A function doesn't perform any work until a script calls it by name or reference. Scripts invoke functions (that is, get functions to do something) via four routes: document object event handlers; JavaScript statements; `href` attributes pointing to a `javascript:` URL; and the more recent `call()` and `apply()` methods of function objects. The one approach not discussed at length yet in this book is the `javascript:` URL (some say pseudo-URL).

Several HTML tags have `href` attributes that normally point to Internet URLs for navigating to another page or loading a MIME file that requires a helper application or plug-in. These HTML tags are usually tags for clickable objects, such as links and client-side image map areas.

A JavaScript-enabled browser has a special, built-in URL pseudo-protocol — `javascript:` — that lets the `href` attribute point to a JavaScript function or method rather than to a URL out on the Net. For example, it is common practice to use the `javascript:` URL to change the contents of two frames from a single link. Because the `href` attribute is designed to point to only a single URL, you'd be out of luck without a convenient way to put multiframe navigation into your hands. Implement multiframe navigation by writing a function that sets the `location.href` properties of the two frames; then invoke that function from the `href` attribute. The following example shows what the script may look like:

```
function loadPages() {
   parent.frames[1].location.href = "page2.html";
   parent.frames[2].location.href = "instrux2.html";
}
...
<a href="javascript:loadPages()">Next</a>
```

These kinds of function invocations can include parameters, and the functions can do anything you want. One potential side effect to watch out for occurs when the function returns a value (perhaps the function is also invoked from other script locations where a returned value is expected). Because the `href` attribute sets the `target` window to whatever the attribute evaluates to, the returned value is assigned to the `target` window — probably not what you want.

To prevent the assignment of a returned value to the `href` attribute, prefix the function call with the `void` operator:

```
<a href="javascript:void loadPages()">
```

If you don't want the `href` attribute to do anything (that is, let the `onclick` event handler do all the work), assign a blank function after the operator:

```
<a href="javascript:void (0)">
```

Experienced programmers of many other languages recognize this operator as a way of indicating that no values are returned from a function or procedure. The operator has that precise functionality here, but in a nontraditional location.

Variable scope: Globals and locals

A variable can have two scopes in JavaScript. As you might expect, any variable initialized within the main flow of a script (not inside a function) is a *global variable* in that any statement in the same document's script can access it by name. You can, however, also initialize variables inside a function (in a `var` statement) so the variable name applies only to statements inside that function. By limiting the scope of the variable to a single function, you can reuse the same variable name in multiple functions thereby enabling the variables to carry very different information in each function. Listing 33-2 demonstrates the various possibilities.

Listing 33-2: **Variable Scope Workbench Page**

```
<html>
  <head>
     <title>Variable Scope Trials</title>
     <script type="text/javascript">
     var headGlobal = "Gumby";
     function doNothing() {
        var headLocal = "Pokey";
        return headLocal;
     }
     </script>
  </head>
  <body>
     <script type="text/javascript">
     // two global variables
     var aBoy = "Charlie Brown";
     var hisDog = "Snoopy";
     function testValues() {
        var hisDog = "Gromit";  // initializes local version of "hisDog"
        var page = "";
        page += "headGlobal is: " + headGlobal + "<br />";
        // page += "headLocal is: " + headLocal + "<br />" // won't run:
        // ...headLocal not defined
        page += "headLocal value returned from head function is: " +
           doNothing() + "<br />";
        page += " aBoy is: " + aBoy + "<br />"; // picks up global
        page += "local version of hisDog is: " + hisDog + "<br />"; // "sees"
        // ...only local version
        document.write(page);
     }
     testValues();
     document.write("global version of hisDog is intact: " + hisDog);
     </script>
  </body>
</html>
```

In this page, you define a number of variables — some global, others local — that are spread out in the document's Head and Body sections. When you load this page, it runs the `testValues()` function, which accounts for the current values of all the variable names. The script then follows up with one more value extraction that was masked in the function. The results of the page look like this:

```
headGlobal is: Gumby
headLocal value returned from head function is: Pokey
aBoy is: Charlie Brown
local version of hisDog is: Gromit
global version of hisDog is intact: Snoopy
```

Examine the variable initialization throughout this script. In the Head, you define the first variable (`headGlobal`) as a global style outside of any function definition. The `var` keyword for the global variable is optional but often helpful for enabling you to see at a glance where you initialize your variables. You then create a short function, which defines a variable (`headLocal`) that only statements in the function can use.

In the Body, you define two more global variables: `aBoy` and `hisDog`. Inside the Body's function (for purposes of demonstration), you reuse the `hisDog` variable name. By initializing `hisDog` with the `var` statement inside the function, you tell JavaScript to create a separate variable whose scope is only within the function. This initialization does not disturb the global variable of the same name. It can, however, make things confusing for you as the script author.

Statements in this script attempt to collect the values of variables scattered around the script. Even from within this script, JavaScript has no problem extracting global variables directly — including the one defined in the Head. But JavaScript cannot get the local variable defined in the other function — that `headLocal` variable is private to its own function. Trying to run a script that references that variable value will result in an error message saying that the variable name is not defined. In the eyes of everyone else outside of the `doNothing()` function, that's true. If you really need that value, you can have the function return the value to a calling statement as you do in the `testValues()` function.

Near the end of the function, the script reads the `aBoy` global value without a hitch. But because you initialized a separate version of `hisDog` inside that function, only the localized version is available to the function. If you reassign a global variable name inside a function, you cannot access the global version from inside that function.

As proof that the global variable — whose name was reused inside the `testValues()` function — remains untouched, the script writes that value to the end of the page for all to see. Charlie Brown and his dog are reunited.

A benefit of this variable-scoping scheme is that you can reuse "throw-away" variable names in any function you like. For instance, you can use the `i` loop counting variable in every function that employs loops. (In fact, you can reuse it in multiple `for` loops of the same function because the `for` loop reinitializes the value at the start of the loop.) If you pass parameters to a function, you can assign to those parameter variables the same names to aid in consistency. For example, a common practice is to pass an entire form object reference as a parameter to a function (using a `this.form` parameter in the event handler). For every function that catches one of these objects, you can use the variable name `form` in the parameter:

```
function doSomething(form) {
   statements
}
...
<input type="button" value="Do Something" onclick="doSomething(this.form)" />
```

If five buttons on your page pass their form objects as parameters to five different functions, each function can assign `form` (or whatever you want to use) to that parameter value.

I recommend reusing variable names only for these "throwaway" variables. In this case, the variables are all local to functions, so the possibility of a mix-up with global variables does not exist. But the thought of reusing a global variable name as, say, a special case inside a function sends shivers up my spine. Such a tactic is doomed to cause confusion and error.

Some programmers devise naming conventions to avoid reusing global variables as local variables. A popular scheme puts a lowercase "g" in front of any global variable name. In the example from Listing 33-2, you can name the global variables

```
gHeadGlobal
gABoy
gHisDog
```

Then, if you define local variables, don't use the leading "g." A similar scheme involves using an underscore character (_) instead of a `g` in front of global variable names. Any scheme you employ to prevent the reuse of variable names in different scopes is fine as long as it does the job.

In a multiframe or multiwindow environment, your scripts can also access global variables from any other document currently loaded into the browser. For details about this level of access, see Chapter 16.

Variable scoping rules apply equally to nested functions in NN4+, IE4+, and W3C browsers. Any variables defined in an outer function (including parameter variables) are exposed to all functions nested inside. But if you define a new local variable inside a nested function, that variable is not available to the outer function. Instead, you can return a value from the nested function to the statement in the outer function that invokes the nested function.

Parameter variables

When a function receives data in the form of parameters, remember that the values may be copies of the data (in the case of run-of-the-mill data values) or references to real objects (such as a form object). In the latter case, you can change the object's modifiable properties in the function when the function receives the object as a parameter, as shown in the following example:

```
function validateCountry (form) {
   if (form.country.value == "") {
      form.country.value = "USA";
   }
}
```

Therefore, whenever you pass an object reference as a function parameter, be aware that the changes you make to that object in its "passed" form affect the real object.

As a matter of style, if my function needs to extract properties or results of methods from passed data (such as object properties or string substrings), I like to do that at the start of the function. I initialize as many variables as needed for each piece of data used later in the function. This task enables me to assign meaningful names to the data chunks, rather than rely on potentially long references within the working part of the function (such as using a variable like `inputStr` instead of `form.entry.value`).

Recursion in functions

Functions can call themselves — a process known as *recursion*. The classic example of programmed recursion is the calculation of the factorial (the factorial for a value of `4` is `4 * 3 * 2 * 1`), shown in Listing 33-3.

In the third line of this function, the statement calls itself, passing along a parameter of the next lower value of `n`. As this function executes, diving ever deeper into itself, JavaScript watches intermediate values and performs the final evaluations of the nested expressions. Be sure to test any recursive function carefully. In particular, make sure that the recursion is finite: that a limit exists for the number of times it can recurse. In the case of Listing 33-3, that limit is the initial value of `n`. Failure to watch out for this limit may cause the recursion to overpower the limits of the browser's memory and even lead to a crash.

Listing 33-3: **A JavaScript Function Utilizing Recursion**

```
function factorial(n) {
   if (n > 0) {
      return n * (factorial(n-1));
   } else {
```

```
      return 1;
   }
}
```

Turning functions into libraries

As you start writing functions for your scripts, be on the lookout for ways to make functions generalizable (written so that you can reuse the function in other instances, regardless of the object structure of the page). The likeliest candidates for this kind of treatment are functions that perform specific kinds of validation checks (see examples in Chapter 43 on the CD-ROM), data conversions, or iterative math problems.

To make a function generalizable, don't let it make any references to specific objects by name. Object names generally change from document to document. Instead, write the function so that it accepts a named object as a parameter. For example, if you write a function that accepts a `text` object as its parameter, the function can extract the object's data or invoke its methods without knowing anything about its enclosing form or name. Look again, for example, at the `factorial()` function in Listing 33-4 — but now as part of an entire document.

Listing 33-4: **Calling a Generalizable Function**

```
<html>
   <head>
      <title>Variable Scope Trials</title>
      <script type="text/javascript">
      function factorial(n) {
         if (n > 0) {
            return n * (factorial(n - 1));
         } else {
            return 1;
         }
      }
      </script>
   </head>
   <body>
      <form>
         Enter an input value: <input type="text" name="input" value="0" />
         <p><input type="button" value="Calc Factorial"
            onclick="this.form.output.value =
            factorial(this.form.input.value)" /></p>
         <p>Results: <input type="text" name="output" /></p>
      </form>
   </body>
</html>
```

This function is designed to be generalizable, accepting only the input value (`n`) as a parameter. In the form, the `onClick` event handler of the button sends only the input value from one of the form's fields to the `factorial()` function. The returned value is assigned to the output field of the form. The `factorial()` function is totally ignorant about forms, fields, or buttons in this document. If I need this function in another script, I can copy and paste it into that script knowing that it has been pretested. Any generalizable function is part of my personal library of scripts — from which I can borrow — and saves me time in future scripting tasks.

You cannot always generalize a function. Somewhere along the line in your scripts, you must have references to JavaScript or custom objects. But if you find that you're frequently writing functions that perform the same kind of actions, see how you can generalize the code and put the results in your library of ready-made functions. You should also consider placing these reusable library functions in an external `.js` library file. See Chapter 13 for details on this convenient way to share utility functions among many documents.

Custom Objects

In all the previous chapters of this book, you've seen how conveniently the browser document object models organize all the information about the browser window and its document. What may not be obvious from the scripting you've done so far is that JavaScript enables you to create your own objects in memory — objects with properties and methods. These objects are not user-interface elements on the page but rather the kinds of objects that may contain data and script functions (behaving as methods) whose results the user can see displayed in the browser window.

You actually had a preview of this power in Chapter 30's discussion about arrays. An array, you recall, is an ordered collection of data. An object typically contains different kinds of data. It doesn't have to be an ordered collection of data — although your scripts can use objects in constructions that strongly resemble arrays. Moreover, you can attach any number of custom functions as methods for that object. You are in total control of the object's structure, data, and behavior.

An example — planetary objects

Building on your familiarity with the planetary data array created in Chapter 30, this chapter shows you how convenient it is to use the data when it is constructed in the form of custom objects. The application goal for the extended example in this section is to present a pop-up list of the nine planets of the solar system and display data about the selected planet. From a user-interface perspective (and for more exposure to multiframe environments), the resulting data displays in a separate frame of a two-frame window. This means your object method builds HTML on the fly and plugs it into the display frame. If you implement this application strictly for IE4+, NN6+, and W3C browsers, you can apply the same data to reconstruct the displayed table data for each user selection. The example as shown, however, is fully backward compatible for all scriptable browsers.

In this chapter, instead of building arrays to hold the data, you build objects — one object for each planet. The design of your object has five properties and one method. The properties of each planet are: name, diameter, distance from the sun, year length, and day length. To assign more intelligence to these objects, you give each of them the capability to display their data in the lower frame of the window. You can conveniently define one function that knows how to behave with any of these planet objects, rather than having to define nine separate functions.

Listing 33-5 shows the source code for the document that creates the frameset for your planetary explorations; Listing 33-6 shows the entire HTML page for the object-oriented planet document, which appears in the top frame.

Listing 33-5: **Framesetting Document for a Two-Frame Window**

```
<html>
  <head>
     <title>Solar System Viewer</title>
```

```
      <script type="text/javascript">
      function blank() {
         return "<html><body><\/body><\/html>";
      }
      </script>
   </head>
   <frameset rows="50%,50%"
   onload="Frame1.doDisplay(Frame1.document.forms[0].planetsList)">
      <frame name="Frame1" src="lst33-06.htm" />
      <frame name="Frame2" src="javascript:parent.blank()" />
   </frameset>
</html>
```

One item to point out in Listing 33-5 is that because the lower frame isn't filled until the upper frame's document loads, you need to assign some kind of URL for the `src` attribute of the second frame. Rather than add the extra transaction and file burden of a blank HTML document, here you use the `javascript:` URL to invoke a function. In this instance, I want the value returned from the function (a blank HTML page) to be reflected into the target frame (no `void` operator here). This method provides the most efficient way of creating a blank frame in a frameset.

Listing 33-6: **Object-Oriented Planetary Data Presentation**

```
<html>
   <head>
      <title>Our Solar System</title>
      <script type="text/javascript">
      // method definition
      function showPlanet() {
         var result = "<html><body><center><table border='2'>";
         result += "<caption align='top'>Planetary data for: <b>" + this.name +
            "</b></caption>";
         result += "<tr><td align='right'>Diameter:</td><td>" + this.diameter +
            "</td></tr>";
         result += "<tr><td align='right'>Distance from Sun:</td><td>" +
            this.distance + "</td></tr>";
         result += "<tr><td align='right'>One Orbit Around Sun:</td><td>" +
            this.year + "</td></tr>";
         result += "<tr><td align='right'>One Revolution (Earth
            Time):</td><td>" + this.day + "</td></tr>";
         result += "</table></center></body></html>";
         // display results in a second frame of the window
         parent.Frame2.document.write(result);
         parent.Frame2.document.close();
      }

      // definition of planet object type;
      // 'new' will create a new instance and stuff parameter data into object
      function planet(name, diameter, distance, year, day) {
         this.name = name;
         this.diameter = diameter;
         this.distance = distance;
         this.year = year;
```

Continued

Listing 33-6 *(continued)*

```
        this.day = day;
        this.showPlanet = showPlanet; // make showPlanet() function a method
        // ...of planet
     }

     // create new planet objects, and store in a series of variables
     var Mercury = new planet("Mercury","3100 miles", "36 million miles",
        "88 days", "59 days");
     var Venus = new planet("Venus", "7700 miles", "67 million miles",
        "225 days", "244 days");
     var Earth = new planet("Earth", "7920 miles", "93 million miles",
        "365.25 days","24 hours");
     var Mars = new planet("Mars", "4200 miles", "141 million miles",
        "687 days", "24 hours, 24 minutes");
     var Jupiter = new planet("Jupiter","88,640 miles","483 million miles",
        "11.9 years", "9 hours, 50 minutes");
     var Saturn = new planet("Saturn", "74,500 miles","886 million miles",
        "29.5 years", "10 hours, 39 minutes");
     var Uranus = new planet("Uranus", "32,000 miles",
        "1.782 billion miles","84 years", "23 hours");
     var Neptune = new planet("Neptune","31,000 miles",
        "2.793 billion miles","165 years", "15 hours, 48 minutes");
     var Pluto = new planet("Pluto", "1500 miles", "3.67 billion miles",
        "248 years", "6 days, 7 hours");

     // called from push button to invoke planet object method
     function doDisplay(popup) {
        i = popup.selectedIndex;
        eval(popup.options[i].text + ".showPlanet()");
     }
     </script>
  </head>
  <body>
     <h1>The Daily Planet</h1>
     <hr />
     <form>
        <p>Select a planet to view its planetary data: <select
        name='planetsList' onchange='doDisplay(this)'>
           <option>Mercury</option>
           <option>Venus</option>
           <option selected="selected">Earth</option>
           <option>Mars</option>
           <option>Jupiter</option>
           <option>Saturn</option>
           <option>Uranus</option>
           <option>Neptune</option>
           <option>Pluto</option>
        </select></p>
     </form>
  </body>
</html>
```

The first task in the Head is to define the function that becomes a method in each of the objects. You must do this task before scripting any other code that adopts the function as its method. Failure to define the function ahead of time results in an error — the function name is not defined. If you compare the data extraction methodology with the function in the array version, notice that the parameter for the index value is gone and the reference to each property begins with `this`. Later, I return to the custom method after giving you a look at the rest of the Head code.

Next comes the object constructor function, which performs several important tasks. For one, everything in this function establishes the structure of your custom object: the properties available for data storage and retrieval and any methods that the object can invoke. The name of the function is the name you use later to create new instances of the object. Therefore, choosing a name that truly reflects the nature of the object is important. And, because you probably want to stuff some data into the function's properties to get one or more instances of the object loaded and ready for the page's user, the function definition includes parameters for each of the properties defined in this object definition.

Inside the function, you use the `this` keyword to assign data that comes in as parameters to labeled properties. For this example, I use the same names for both the incoming parameter variables and the properties. That's primarily for convenience (and is very common practice), but you can assign any variable and property names you want and connect them any way you like. In the `planet()` constructor function, five property slots are reserved for every instance of the object whether or not any data actually is placed in every property (any unassigned slot has a value of `null`).

The last entry in the `planet()` constructor function is a reference to the `showPlanet()` function defined earlier. Notice that the assignment statement doesn't refer to the function with its parentheses — just to the function name. When JavaScript sees this assignment statement, it looks back through existing definitions (those functions defined ahead of the current location in the script) for a match. If it finds a function (as it does here), JavaScript knows to assign the function to the identifier on the left side of the assignment statement. In doing this task with a function, JavaScript automatically sets up the identifier as a method name for this object. As you do in every JavaScript method you encounter, you must invoke a method by using a reference to the object, a period, and the method name followed by a set of parentheses. You see that syntax in a minute.

The next long block of statements creates the individual objects according to the definition established in the `planet()` constructor. Similar to an array, an assignment statement and the keyword `new` create an object. I assign names that are not only the real names of planets (the `Mercury` object name is the Mercury planet object) but that also can come in handy later when the `doDisplay()` function extracts names from the pop-up list in search of a particular object's data.

The act of creating a new object sets aside space in memory (associated with the current document) for this object and its properties. In this script, you create nine object spaces, each with a different set of properties. Notice that no parameter is sent (or expected at the function) that corresponds to the `showPlanet()` method. Omitting that parameter here is fine because the specification of that method in the object definition means that the script automatically attaches the method to every version (instance) of the planet object that it creates.

The last function definition, `doDisplay()`, is invoked whenever the user makes a choice from the list of planets in the upper frame. This function is also invoked via the frameset's `onload` event handler so that an initial table is displayed from the default selected item (see Figure 33-1). Invoking the function from the upper frame's `onload` event handler can cause problems (such as the failure of the other frame) if the frameset is not completely loaded.

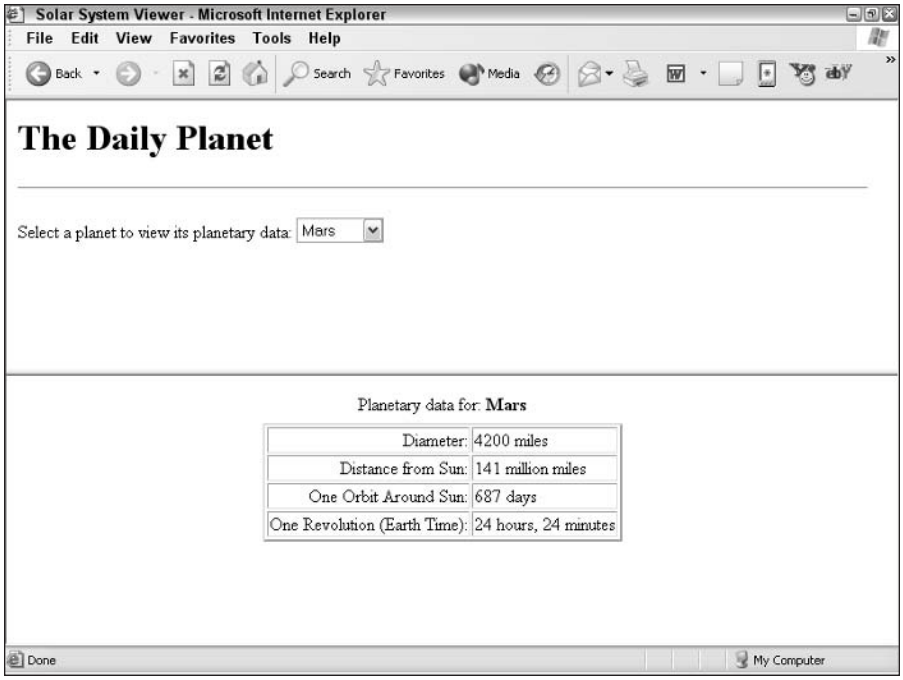


Figure 33-1: An external and internal face-lift for an earlier application.

The `onchange` event handler in the `select` list passes that `select` element's reference to the `doDisplay()` function. In that function, the `select` object is assigned to a variable called `popup` to help you visualize that the object is the pop-up list. The first statement extracts the index value of the selected item. Using that index value, the script extracts the text. But things get a little tricky because you need to use that text string as a variable name — the name of the planet — and append it to the call to the `showPlanet()` method. To make the disparate data types come together, use the `eval()` function. Inside the parentheses, extract the string for the planet name and concatenate a string that completes the reference to the object's `showPlanet()` method. The `eval()` function evaluates that string, which turns it into a valid method call. Therefore, if the user selects Jupiter from the pop-up list, the method call becomes `Jupiter.showPlanet()`.

Now it's time to look back to the `showPlanet()` function/method definition at the top of the script. When that method runs in response to a user selection of the planet Jupiter, the method's only scope is of the `Jupiter` object. Therefore, all references to `this.propertyName` in `showPlanet()` refer to `Jupiter` only. The only possibility for `this.name` in the `Jupiter` object is the value assigned to the `name` property for `Jupiter`. The same goes for the rest of the properties extracted in the function/method.

Creating an array of objects

In Listing 33-6, each of the planet objects is assigned to a global variable whose name is that of the planet. If the idea of custom objects is new to you, this idea probably doesn't sound so bad because it's easy to visualize each variable representing an object. But, as shown in the `doDisplay()` function, accessing an object by name requires use of the `eval()` function to

convert a string representation to a valid object reference. While it's not too important in this simple example, the `eval()` function is not particularly efficient in JavaScript. If you find yourself using an `eval()` function, look for ways to improve efficiency such that you can reference an object by string. The way to accomplish that streamlining for this application is to place the objects in an array whose index values are the planet names.

To assign the custom objects in Listing 33-6 to an array, first create an empty array and then assign the result of each object constructor call to an entry in the array. The modified code section looks like the following (formatted to fit this printed page):

```
// create array
var planets = new Array();
// populate array with new planet objects
planets["Mercury"] =
   new planet("Mercury","3100 miles", "36 million miles",
   "88 days", "59 days");
planets["Venus"] =
   new planet("Venus", "7700 miles", "67 million miles",
   "225 days", "244 days");
planets["Earth"] =
   new planet("Earth", "7920 miles", "93 million miles",
   "365.25 days","24 hours");
planets["Mars"] =
   new planet("Mars", "4200 miles", "141 million miles",
   "687 days", "24 hours, 24 minutes");
planets["Jupiter"] =
   new planet("Jupiter","88,640 miles","483 million miles",
   "11.9 years", "9 hours, 50 minutes");
planets["Saturn"] =
   new planet("Saturn", "74,500 miles","886 million miles",
   "29.5 years", "10 hours, 39 minutes");
planets["Uranus"] =
   new planet("Uranus", "32,000 miles","1.782 billion miles",
   "84 years", "23 hours");
planets["Neptune"] =
   new planet("Neptune","31,000 miles","2.793 billion miles",
   "165 years", "15 hours, 48 minutes");
planets["Pluto"] =
   new planet("Pluto", "1500 miles", "3.67 billion miles",
   "248 years", "6 days, 7 hours");
```

The supreme advantage to this approach comes in a modified `doDisplay()` function, which can use the string value from the `select` element directly without any conversion to an object reference:

```
// called from push button to invoke planet object method
function doDisplay(popup) {
   i = popup.selectedIndex;
   planets[popup.options[i].text].showPlanet();
}
```

The presence of so many similar objects cries out for their storage as an array. Because the names play a key role in their choice for this application, the named index values work best; in other situations, you may prefer to use numeric indexes to facilitate looping through the array.

Adding a custom method

You're approaching advanced subject matter at this point, so I merely mention and briefly demonstrate an additional power of defining and using custom objects. A custom object can

have a reference to another custom object as a property. Let's extend the planet example to help you understand the implications.

Say that you want to beef up the planet page with a photo of each planet. Each photo has a URL for the photo file; each photo also contains other information, such as the copyright notice and a reference number, which displays on the page for the user. One way to handle this additional information is to create a separate object definition for a photo database. Such a definition may look like this:

```
function photo(name, URL, copyright, refNum) {
   this.name = name;
   this.URL = URL;
   this.copyright = copyright;
   this.refNum = refNum;
}
```

You then need to create individual `photo` objects for each picture. One such definition may look like this:

```
mercuryPhoto = new photo("Planet Mercury", "/images/merc44.gif",
   "(c)1990 NASA", 28372);
```

Attaching a `photo` object to a `planet` object requires modifying the `planet` constructor function to accommodate one more property. The new `planet` constructor looks like this:

```
function planet(name, diameter, distance, year, day, photo) {
   this.name = name;
   this.diameter = diameter;
   this.distance = distance;
   this.year = year;
   this.day = day;
   this.showPlanet = showPlanet;
   this.photo = photo; // add photo property
}
```

Once the `photo` objects are created, you can then create each `planet` object by passing one more parameter — a `photo` object you want associated with that object:

```
// create new planet objects, and store in a series of variables
Mercury = new planet("Mercury", "3100 miles", "36 million miles",
   "88 days", "59 days", mercuryPhoto);
```

To access a property of a `photo` object, your scripts then have to assemble a reference that works its way through the connection with the `planet` object:

```
copyrightData = Mercury.photo.copyright;
```

The potential of custom objects of this type is enormous. For example, you can embed all the copy elements and image URLs for an online catalog in a single document. As the user selects items to view (or cycles through them in sequence), a new JavaScript-written page displays the information in an instant. This requires only the image to be downloaded — unless the image was precached, as described in the image object discussion in Chapter 18. In this case, everything works instantaneously — no waiting for page after page of catalog.

If, by now, you think you see a resemblance between this object-within-an-object construction and a relational database, give yourself a gold star. Nothing prevents multiple objects from having the same subobject as their properties — like multiple business contacts having the same company object property.

More ways to create objects

The examples in Listings 33-5 and 33-6 show a way of creating objects that works with all scriptable browsers. If your audience is limited to users with more modern browsers, additional ways of creating custom objects exist.

From NN3+ and IE4+ onward (including W3C browsers), you can use the `new Object()` constructor to generate a blank object. From that point on, you can define property and method names by simple assignment, as in the following:

```
var Earth = new Object();
Earth.diameter = "7920 miles";
Earth.distance = "93 million miles";
Earth.year = "365.25";
Earth.day = "24 hours";
Earth.showPlanet = showPlanet;  // function reference
```

When you create a lot of like-structured objects, the custom object constructor shown in Listing 33-6 is more efficient. But for single objects, the `new Object()` constructor is more efficient.

For more modern browsers, you can also benefit from a shortcut literal syntax for creating a new object. You can set pairs of property names and their values inside a set of curly braces, and you can assign the whole construction to a variable that becomes the object name. The following script shows how to organize this kind of object constructor:

```
var Earth = {diameter:"7920 miles", distance:"93 million miles", year:"365.25",
             day:"24 hours", showPlanet:showPlanet};
```

Colons link name-value pairs, and commas separate multiple name-value pairs. The value portion of a name-value pair can even be an array (using the `[...]` constructor shortcut) or a nested object (using another pair of curly braces). In fact, you can nest arrays and objects to your heart's content to create exceedingly complex objects. All in all, this is a very compact way to embed data in a page for script manipulation. If your CGI, XML, and database skills are up to the task, consider using a server program to convert XML data into this compact JavaScript version with each XML record being its own JavaScript object. For multiple records, assign the curly-braced object definitions to an array entry. Then your scripts on the client can iterate through the data and generate the HTML to display the data in a variety of forms and sorted according to different criteria (thanks to the JavaScript array-sorting powers).

Defining object property getters and setters

A future version of the ECMA-262 language specification will likely include a pair of facilities called *getter* and *setter*. Until such time as the formal syntax is finalized, you can begin to experiment with this technique in NN6+/Moz1+ using temporary syntax that adheres to the likely format (but intentionally uses different keywords until the standard is adopted). When the standard is adopted, a subsequent version of NN/Moz will include the standard keywords.

I introduced the idea of creating a getter and setter for an object briefly in Chapter 14, where the NN6 syntax style extended properties of some W3C DOM objects to include some of the Microsoft-specific (and very convenient) DOM syntax. Most notably, you can define a getter for any container to return an array of nested elements just like the IE-only `document.all` collection.

The purpose of a getter is to assign a new property to the prototype of an object and to define how the value returned by the property should be evaluated. A setter does the same, but it also defines how a new value assigned to the property should apply the value to the object. Both

definitions are written in the form of anonymous functions, such that reading or writing an object's property value can include sophisticated processing for either operation.

Getters and setters are assigned to the `prototype` property of an object, thus enabling you to customize native and DOM objects. The NN6 syntax fashions getters, setters, and methods of an object's prototype with the following syntax:

```
object.prototype.__defineGetter__("propName", function)
object.prototype.__defineSetter__("propName", function)
```

Note that the underscores before and after the method names are actually pairs of underscore characters (that is, _, _, `defineGetter`, _, _). This double underscore was chosen as a syntax that the ECMA standard will not use, so it will not conflict with the eventual syntax for this facility.

The first parameter of the method is the name of the property for which the getter or setter is defined. This can be an existing property name that you want to override. The second parameter can be a function reference; but more likely it will be an anonymous function defined in place. By using an anonymous function, you can take advantage of the context of the object for which the property is defined. For each property, define both a getter and setter — even if the property is meant to be read-only or write-only.

To see how this mechanism works, let's use the getter and setter shown in Chapter 14 to add an `innerText` property to HTML elements in NN6+. This property is read/write, so functions are defined for both the getter and setter. The getter definition is as follows:

```
HTMLElement.prototype.__defineGetter__("innerText", function () {
   var rng = document.createRange();
   rng.selectNode(this);
   return rng.toString();
})
```

The modified object is the basic `HTMLElement` object — the object that NN6+ uses to create instances of every HTML element for the page. After the preceding statement executes, every HTML element on the page inherits the new `innerText` property. Each time the `innerText` property is read for an element, the anonymous function in this getter executes. Thus, after a text range object is created, the range is set to the node that is the current element. This is an excellent example of how the context of the current object allows the use of the `this` keyword to refer to the very same object. Finally, the string version of the selected range is returned. It is essential that a getter function include a `return` statement and that the returned value is of the desired data type. Also take notice of the closing of the function's curly brace and the getter method's parenthesis.

By executing this function each time the property is read, the getter always returns the current state of the object. If content of the element has changed since the page loaded, you are still assured of getting the current text inside the element. This is far superior to simply running the statements inside this function once as the page loads to capture a static view of the element's text.

The corresponding setter definition is as follows:

```
HTMLElement.prototype.__defineSetter__("innerText", function (txt) {
   var rng = document.createRange();
   rng.selectNodeContents(this);
   rng.deleteContents();
   var newText = document.createTextNode(txt);
   this.appendChild(newText);
   return txt;
})
```

To assign a value to an object's property, the setter function requires that a parameter variable receive the assigned value. That parameter variable plays a role somewhere within the function definition. For this particular setter, the current object (`this`) also manipulates the text range object. The contents of the current element are deleted, and a text node comprising the text passed as a parameter is inserted into the element. To completely simulate the IE behavior of setting the `innerText` property, the text is returned. While setters don't always return values, this one does so that the expression that assigns a value to the `innerText` property evaluates to the new text.

If you want to create a read-only property, you still define a setter for the property but you also assign an empty function, as in:

```
Node.prototype.__defineSetter__("all", function() {})
```

This prevents assignment statements to a read-only property from generating errors. A write-only property should also have a getter that returns `null` or an empty string, as in:

```
HTMLElement.prototype.__defineGetter__("outerHTML", function() {return ""})
```

Because the getter and setter syntax shown here is unique to NN6, you must obviously wrap such statements inside object detection or browser version detection statements. And, to reiterate, this syntax will change in future browser versions once ECMA adopts the formal syntax.

Using custom objects

There is no magic to knowing when to use a custom object instead of an array in your application. The more you work with and understand the way custom objects work, the more likely you will think about your data-carrying scripts in these terms — especially if an object can benefit from having one or more methods associated with it. This avenue is certainly not one for beginners, but I recommend that you give custom objects more than a casual perusal once you gain some JavaScripting experience.

Object-Oriented Concepts

As stated several times throughout this book, JavaScript is object-based rather than object-oriented. Instead of adhering to the class, subclass, and inheritance schemes of object-oriented languages such as Java, JavaScript uses what is called *prototype inheritance*. This scheme works not only for native and DOM objects but also for custom objects.

Adding a prototype

A custom object is frequently defined by a constructor function, which typically parcels out initial values to properties of the object, as in the following example:

```
function car(plate, model, color) {
   this.plate = plate;
   this.model = model;
   this.color = color;
}
var car1 = new car("AB 123", "Ford", "blue");
```

NN4+, IE4+, and W3C browsers offer a handy shortcut, as well, to stuff default values into properties if none are provided (the supplied value is `null`, `0`, or an empty string). The `OR` operator (`||`) can let the property assignment statement apply the passed value, if present,

or a default value you hard-wire into the constructor. Therefore, you can modify the preceding function to offer default values for the properties:

```
function car(plate, model, color) {
   this.plate = plate || "missing";
   this.model = model || "unknown";
   this.color = color || "unknown";
}
var car1 = new car("AB 123", "Ford", "");
```

After the preceding statements run, the `car1` object has the following properties:

```
car1.plate    // value = "AB 123"
car1.model    // value = "Ford"
car1.color    // value = "unknown"
```

If you then add a new property to the constructor's `prototype` property, as in

```
car.prototype.companyOwned = true;
```

any `car` object you already created or are about to create automatically inherits the new `companyOwned` property and its value. You can still override the value of the `companyOwned` property for any individual `car` object. But if you don't override the property for instances of the `car` object, the `car` objects whose `companyOwned` property is not overridden automatically inherit any change to the `prototype.companyOwned` value. This has to do with the way JavaScript looks for `prototype` property values.

Prototype inheritance

Each time your script attempts to read or write a property of an object, JavaScript follows a specific sequence in search of a match for the property name. The sequence is as follows:

1. If the property has a value assigned to the current (local) object, this is the value to use.
2. If there is no local value, check the value of the property's prototype of the object's constructor.
3. Continue up the prototype chain until either a match of the property is found (with a value assigned to it) or the search reaches the native `Object` object.

Therefore, if you change the value of a constructor's `prototype` property and you do not override the property value in an instance of that constructor, JavaScript returns the current value of the constructor's `prototype` property.

Nested objects and prototype inheritance

When you begin nesting objects, especially when one object invokes the constructor of another, there is an added wrinkle to the prototype inheritance chain. Let's continue with the `car` object defined earlier. In this scenario, consider the `car` object to be akin to a root object that has properties shared among two other types of objects. One of the object types is a company fleet vehicle, which needs the properties of the root `car` object (`plate`, `model`, `color`) but also adds some properties of its own. The other object that shares the `car` object is an object representing a car parked in the company garage — an object that has additional properties regarding the parking of the vehicle. This explains why the `car` object is defined on its own.

Now look at the constructor function for the parking record, along with the constructor for the basic `car` object:

```
function car(plate, model, color) {
   this.plate = plate || "missing";
   this.model = model || "unknown";
   this.color = color || "unknown";
}
function carInLot(plate, model, color, timeIn, spaceNum) {
   this.timeIn = timeIn;
   this.spaceNum = spaceNum;
   this.carInfo = car;
   this.carInfo(plate, model, color);
}
```

The `carInLot` constructor not only assigns values to its unique properties (`timeIn` and `spaceNum`) but it also includes a reference to the `car` constructor arbitrarily assigned to a property called `carInfo`. This property assignment is merely a conduit that allows property values intended for the `car` constructor to be passed within the `carInLot` constructor function. To create a `carInLot` object, use a statement like the following:

```
var car1 = new carInLot("AA 123", "Ford", "blue", "10:02AM", "31");
```

After this statement, the `car1` object has the following properties and values:

```
car1.timeIn      // value = "10:02AM"
car1.spaceNum    // value = "31"
car1.carInfo     // value = reference to car object constructor function
car1.plate       // value = "AA 123"
car1.model       // value = "Ford"
car1.color       // value = "blue"
```

Let's say that five `carInLot` objects are created in the script (`car1` through `car5`). The prototype wrinkle comes into play if, for example, you assign a new property to the `car` constructor prototype:

```
car.prototype.companyOwned = true;
```

Even though the `carInLot` objects use the `car` constructor, the instances of `carInLot` objects do not have a prototype chain back to the `car` object. As the preceding code stands, even though you've added a `companyOwned` property to the `car` constructor, no `carInLot` object inherits that property (even if you were to create a new `carInLot` object after defining the new `prototype` property for `car`). To get the `carInLot` instances to inherit the `prototype.companyOwned` property, you must explicitly connect the prototype of the `carInLot` constructor to the `car` constructor prior to creating instances of `carInLot` objects:

```
carInLot.prototype = new car();
```

The complete sequence, then, is as follows:

```
function car(plate, model, color) {
   this.plate = plate || "missing";
   this.model = model || "unknown";
   this.color = color || "unknown";
}
function carsInLot(plate, model, color, timeIn, spaceNum) {
   this.timeIn = timeIn;
   this.spaceNum = spaceNum;
   this.carInfo = car;
   this.carInfo(plate, model, color);
}
```

```
carsInLot.prototype = new car();
var car1 = new carsInLot("123ABC", "Ford","blue","10:02AM", "32");
car.prototype.companyOwned = true;
```

After this stretch of code runs, the `car1` object has the following properties and values:

```
car1.timeIn             // value = "10:02AM"
car1.spaceNum           // value = "31"
car1.carInfo            // value = reference to car object constructor function
car1.plate              // value = "AA 123"
car1.model              // value = "Ford"
car1.color              // value = "blue"
car1.companyOwned       // value = true
```

NN4+ provides one extra, proprietary bit of syntax in this prototype world. The `__proto__` property (that's with double underscores before and after the word "proto") returns a reference to the object that is next up the prototype chain. For example, if you inspect the properties of `car1.__proto__` after the preceding code runs, you see that the properties of the object next up the prototype chain are as follows:

```
car1.__proto__.plate           // value = "AA 123"
car1.__proto__.model           // value = "Ford"
car1.__proto__.color           // value = "blue"
car1.__proto__.companyOwned    // value = true
```

This property can be helpful in debugging custom objects and prototype inheritance chain challenges, but the property is not part of the ECMA standard. Therefore, I discourage you from using the property in your production scripts.

Object Object

Properties	*Methods*
constructor	hasOwnProperty()
prototype	isPrototypeOf()
	propertyIsEnumerable()
	toLocaleString()
	toString()
	valueOf()

Syntax

Creating an object object:

```
function constructorName([arg1,...[,argN]]) {
    statement(s)
}
var objName = new constructorName(["argName1",...[,"argNameN"]);
var objName = new Object();
var objName = {propName1:propVal1[, propName2:propVal2[,...N]]}
```

Accessing an object object properties and methods:

```
objectReference.property | method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

About this object

While it might sound like doubletalk, the `Object` object is a vital native object in the JavaScript environment. It is the root object on which all other native objects — such as `Date`, `Array`, `String`, and the like — are based. This object also provides the foundation for creating custom objects, as described earlier in this chapter.

By and large, your scripts do not access the properties of the native `Object` object. The same is true for many of its methods, such as `toString()` and `valueOf()`, which internally allow debugging alert dialog boxes (and The Evaluator) to display something when referring to an object or its constructor.

You can use a trio of methods to perform some inspection of the prototype environment of an object instance. They are of interest primarily to advanced scripters who are building extensive, simulated object-oriented applications.

Methods

hasOwnProperty("*propName*")

Returns: Boolean.
Compatibility: WinIE5.5+, MacIE-, NN6+, Moz1+, Safari-

The `hasOwnProperty()` method returns `true` if the current object instance has the property defined in its constructor or in a related constructor function. But if this property is defined externally, as via assignment to the object's `prototype` property, the method returns `false`.

Using the example of the `car` and `carInLot` objects from earlier in this chapter, the following expressions evaluate to `true`:

```
car1.hasOwnProperty("spaceNum");
car1.hasOwnProperty("model");
```

Even though the `model` property is defined in a constructor that is invoked by another constructor, the property belongs to the `car1` object. The following statement, however, evaluates to `false`:

```
car1.hasOwnProperty("companyOwned");
```

This property is defined by way of the prototype of one of the constructor functions and is not a built-in property for the object instance.

isPrototypeOf(*objRef*)

Returns: Boolean.
Compatibility: WinIE5.5+, MacIE-, NN6+, Moz1+, Safari-

The `isPrototypeOf()` method is intended to reveal whether or not the current object has a prototype relation with an object passed as a parameter. In practice, the IE and NN/Moz versions of this method operate differently and return different results.

***objectObject*.propertyIsEnumerable()**

```
propertyIsEnumerable("propName")
```

Returns: Boolean.
Compatibility: WinIE5.5+, MacIE-, NN6+, Moz1+, Safari-

In the terminology of the ECMA-262 language specification, a value is enumerable if constructions such as the `for-in` property inspection loop (see Chapter 31) can inspect it. Enumerable properties include values such as arrays, strings, and virtually every kind of object. According to the ECMA specification, this method is not supposed to work its way up the prototype chain.

✦ ✦ ✦

CHAPTER 34

Global Functions and Statements

In This Chapter

Converting strings into object references

Creating URL-friendly strings

Adding comments to scripts

In addition to all the objects and other language constructs described in the preceding chapters of this reference part of the book, several language items need to be treated on a global scale. These items apply to no particular objects (or any object), and you can use them anywhere in a script. If you read earlier chapters, you were introduced to many of these functions and statements. This chapter serves as a convenient place to highlight these all-important items that are otherwise easily forgotten. At the end of the chapter, note the brief introduction to several objects that are built into the Windows-only versions of Internet Explorer.

This chapter begins with coverage of the following global functions and statements that are part of the core JavaScript language:

Functions	***Statements***
`decodeURI()`	`//` and `/*...*/` (comment)
`decodeURIComponent()`	`const`
`encodeURI()`	`var`
`encodeURIComponent()`	
`escape()`	
`eval()`	
`isFinite()`	
`isNaN()`	
`Number()`	
`parseFloat()`	
`parseInt()`	
`toString()`	
`unescape()`	
`unwatch()`	
`watch()`	

Global functions are not tied to the document object model. Instead, they typically enable you to convert data from one type to another type. The list of global statements is short, but a couple of them appear extensively in your scripting.

Functions

```
decodeURI("encodedURI")
decodeURIComponent("encodedURIComponent")
encodeURI("URIString")
encodeURIComponent("URIComponentString")
```

Returns: String.
Compatibility: WinIE5.5+, MacIE-, NN6+, Moz1+, Safari-

The ECMA-262 Edition 3 standard, as implemented in IE5.5+, NN6+, and Mozilla, provides utility functions that perform a more rigorous conversion of strings to valid URI strings and vice versa than was achieved earlier via the `escape()` and `unescape()` functions (described later in this chapter). The purpose of the encoding functions is to convert any string to a version that you can use as a Uniform Resource Identifier, such as a Web page address or an invocation of a server CGI script. While Latin alphanumeric characters pass through the encoding process untouched, you must use the encoding functions to convert some symbols and other Unicode characters to a form (hexadecimal representations of the character numbers) that the Internet can pass from place to place. The space character, for example, must be encoded to its hex version: `%20`.

Perhaps the biggest difference between the `encodeURI()` and `escape()` functions (and their `decodeURI()` and `unescape()` counterparts) is that the more modern versions do not encode a wide range of symbols that are perfectly acceptable as URI characters according to the syntax recommended in RFC2396 (`http://www.ietf.org/rfc/rfc2396.txt`). Thus, the following characters are not encoded via the `encodeURI()` function:

```
; / ? : @ & = + $ , - _ . ! ~ * ' ( ) #
```

Use the `encodeURI()` and `decodeURI()` functions only on complete URIs. Applicable URIs can be relative or absolute, but these two functions are wired especially so symbols that are part of the protocol (`://`), search string (`?` and `=`, for instance), and directory-level delimiters (`/`) are not encoded. The `decodeURI()` function should work with URIs that arrive from servers as page locations, but be aware that some server CGIs encode spaces into plus symbols (+) that are not decoded back to spaces by the JavaScript function. If the URIs your script needs to decode contain plus symbols in place of spaces, you need to run your decoded URI through a string replacement method to finish the job (regular expressions come in handy here). If you are decoding URI strings that your scripts encoded, use the decode functions only on URIs that were encoded via the corresponding encode function. Do not attempt to decode a URI that was created via the old `escape()` function because the conversion processes work according to different rules.

The difference between a URI and a URI component is that a *component* is a single piece of a URI, generally not containing delimiter characters. For example, if you use the `encodeURIComponent()` function on a complete URI, almost all of the symbols (other than things such as periods) are encoded into hexadecimal versions — including directory delimiters. Therefore, you should use the component-level conversion functions only on quite granular pieces of a URI. For example, if you assemble a search string that has a name-value pair, you can use the `encodeURIComponent()` function separately on the name and on the value. But if you use that function on the pair that is already in the form *`name=value`*, the function encodes the equal symbol to a hexadecimal equivalent.

Example

Use The Evaluator (Chapter 13) to experiment with the differences between encoding a full URI and a component and encoding and escaping a URI string. For example, compare the results of the following three statements:

```
escape("http://www.giantco.com/index.html?code=42")
encodeURI("http://www.giantco.com/index.html?code=42")
encodeURIComponent("http://www.giantco.com/index.html?code=42")
```

Because the sample URI string is valid as is, the `encodeURI()` version makes no changes. Experiment further by making the search string value into a string with a space, and see how each function treats that character.

escape("*URIString*" [,1])
unescape("*escapedURIString*")

Returns: String.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

If you watch the content of the Location field in your browser, you may occasionally see URLs that include a lot of `%` symbols plus some numbers. The format you see is *URL encoding* (more accurately called *URI encoding* — Uniform Resource Identifier rather than Uniform Resource Locator). This format allows even multiple word strings and nonalphanumeric characters to be sent as one contiguous string of a very low, common denominator character set. This encoding turns a character, such as a space, into its hexadecimal equivalent value preceded by a percent symbol. For example, the space character (ASCII value 32) is hexadecimal 20, so the encoded version of a space is `%20`.

All characters, including tabs and carriage returns, can be encoded in this way and sent as a simple string that can be decoded on the receiving end for reconstruction. You can also use this encoding to preprocess multiple lines of text that must be stored as a character string in databases. To convert a plain-language string to its encoded version, use the `escape()` method. This function returns a string consisting of the encoded version. For example:

```
var theCode = escape("Hello there");
    // result: "Hello%20there"
```

Most, but not all, non-alphanumeric characters are converted to escaped versions with the `escape()` function. One exception is the plus sign, which URLs use to separate components of search strings. If you must encode the plus symbol, too, then add the optional second parameter to the function to make the plus symbol convert to its hexadecimal equivalent (2B):

```
var a = escape("Adding 2+2");
    // result: "Adding%202+2
var a = escape("Adding 2+2",1);
    // result: "Adding%202%2B2
```

To convert an escaped string back into plain language, use the `unescape()` function. This function returns a string and converts all URL-encoded strings — including those encoded with the optional parameter.

The `escape()` function operates in a way that is approximately midway between the newer functions `encodeURI()` and `encodeComponentURI()`. The `escape()` function is best used on portions of URIs, such as the search string. If your scripts bounce back and forth between escaped and unescaped strings, be sure to balance the functions of the same type; use `unescape()` only on URI strings that are encoded via the `escape()` function.

Finally, be aware of slightly different behavior with regard to the @ symbol in various browsers. This character is not encoded in IE, but it is encoded (to %40) in NN.

eval("*string*")

Returns: Object reference.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Expression evaluation, as you probably are well aware by now, is an important concept to grasp in scripting with JavaScript (and programming in general). An expression evaluates to some value. But occasionally you need to force an additional evaluation on an expression to receive the desired results. The `eval()` function acts on a string value to force an evaluation of that string expression. Perhaps the most common application of the `eval()` function is to convert a string version of an object reference to a genuine object reference.

Example

The `eval()` function can evaluate any JavaScript statement or expression stored as a string. This includes string equivalents of arithmetic expressions, object value assignments, and object method invocation. I do not recommend that you rely on the `eval()` function, however, because this function is inherently inefficient (from the standpoint of performance). Fortunately, you may not need the `eval()` function to get from a string version of an object's name to a valid object reference. For example, if your script loops through a series of objects whose names include serial numbers, you can use the object names as array indices rather than use `eval()` to assemble the object references. The inefficient way to set the value of a series of fields named `data0`, `data1`, and so on, is as follows:

```
function fillFields() {
   var theObj;
   for (var i = 0; i < 10; i++) {
      theObj = eval("document.forms[0].data" + i);
      theObj.value = i;
   }
}
```

A more efficient way is to perform the concatenation within the index brackets for the object reference:

```
function fillFields() {
   for (var i = 0; i < 10; i++) {
      document.forms[0].elements["data" + i].value = i;
   }
}
```

Whenever you are about to use an `eval()` function, look for ways to use string index values of arrays of objects instead. The W3C DOM makes it even easier with the help of the `document.getElementById()` method, which takes a string as a parameter and returns a reference to the named object.

isFinite(*number*)

Returns: Boolean.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

It is unlikely that you will ever need the `isFinite()` function, but its purpose is to advise whether a number is beyond the absolute minimum or maximum values that JavaScript can handle. If a number is outside of that range, the function returns `false`. The parameter to the function must be a number data type.

isNaN(*expression*)

Returns: Boolean.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

For those instances in which a calculation relies on data coming from a text field or other string-oriented source, you frequently need to check whether the value is a number. If the value is not a number, the calculation may result in a script error.

Example

Use the `isNaN()` function to test whether a value is a number prior to passing the value onto the operation. The most common use of this function is to test the result of a `parseInt()` or `parseFloat()` function. If the strings submitted for conversion to those functions cannot be converted to a number, the resulting value is `NaN` (a special symbol indicating "not a number"). The `isNaN()` function returns `true` if the value is not a number.

A convenient way to use this function is to intercept improper data before it can do damage, as follows:

```
function calc(form) {
   var inputValue = parseInt(form.entry.value);
   if (isNaN(inputValue)) {
      alert("You must enter a number to continue.");
   } else {
      statements for calculation
   }
}
```

Probably the biggest mistake scripters make with this function is failing to observe the case of all the letters in the function name. The trailing uppercase "N" is easy to miss.

Number("*string*")
parseFloat("*string*")
parseInt("*string*" [,*radix*])

Returns: Number.
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

All three of these functions convert a string value into a numeric value. The `parseInt()` and `parseFloat()` functions are compatible across all versions of all browsers; the `Number()` function was introduced in NN4 and IE4.

Use the `Number()` function when your script is not concerned with the precision of the value and prefers to let the source string govern whether the returned value is a floating-point number or an integer. The function takes a single parameter — a string to convert to a number value.

The `parseFloat()` function also lets the string source value determine whether the returned value is a floating-point number or an integer. If the source string includes any non-zero value to the right of the decimal, the result is a floating-point number. But if the string value were, say, `"3.00"`, the returned value would be an integer value.

An extra, optional parameter for `parseInt()` enables you to define the number base for use in the conversion. If you don't specify a radix parameter, JavaScript tries to look out for you; but in doing so, JavaScript may cause some difficulty for you. The primary problem arises when the string parameter for `parseInt()` starts with a zero, which a text box entry or database field might do. In JavaScript, numbers starting with zero are treated as octal (base-8) numbers. Therefore, `parseInt("010")` yields the decimal value `8`.

When you apply the `parseInt()` function, always specify the radix of 10 if you are working in base-10 numbers. You can, however, specify any radix value from 2 through 36. For example, to convert a binary number string to its decimal equivalent, assign a radix of 2 as follows:

```
var n = parseInt("011",2);
    // result: 3
```

Similarly, you can convert a hexadecimal string to its decimal equivalent by specifying a radix of 16:

```
var n = parseInt("4F",16);
    // result: 79
```

Example

Both `parseInt()` and `parseFloat()` exhibit a very useful behavior: If the string passed as a parameter starts with at least one number followed by, say, letters, the functions do their jobs on the numeric part of the string and ignore the rest. This is why you can use `parseFloat()` on the `navigator.appVersion` string to extract just the reported version number without having to parse the rest of the string. For example, NN6 for Windows reports a `navigator.appVersion` value as

```
5.0 (Windows; en-US)
```

But you can get just the numeric part of the string via `parseFloat()`:

```
var ver = parseFloat(navigator.appVersion);
```

Because the result is a number, you can perform numeric comparisons to see, for instance, whether the version is greater than or equal to 4.

`toString([radix])`

Returns: String.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

Every JavaScript core language object and every DOM `document` object has a `toString()` method associated with it. This method is designed to render the contents of the object in as meaningful a way as possible. Table 34-1 shows the result of applying the `toString()` method on each of the convertible core `language` object types.

Table 34-1: toString() Method Results for Object Types

Object Type	*Result*
String	The same string
Number	String equivalent (but numeric literals cannot be converted)
Boolean	`"true"` or `"false"`
Array	Comma-delimited list of array contents (with no spaces after commas)
Function	Decompiled string version of the function definition

Many DOM objects can be converted to a string. For example, a `location` object returns its URL. But when an object has nothing suitable to return for its content as a string, it usually returns a string in the following format:

```
[object objectType]
```

Example

The `toString()` method is available on all versions of all browsers. By setting the optional `radix` parameter between 2 and 16, you can convert numbers to string equivalents in different number bases. Listing 34-1 calculates and draws a conversion table for decimal, hexadecimal, and binary numbers between 0 and 20. In this case, the source of each value is the value of the index counter variable each time the `for` loop's statements execute.

Listing 34-1: **Using toString() with Radix Values**

```
<html>
   <head>
      <title>Number Conversion Table</title>
   </head>
   <body>
      <b>Using toString() to convert to other number bases:</b>
      <hr />
      <table border="1">
         <tr>
            <th>Decimal</th>
            <th>Hexadecimal</th>
            <th>Binary</th>
            <script type="text/javascript">
            var content = "";
            for (var i = 0; i <= 20; i++) {
               content += "<tr>";
               content += "<td>" + i.toString(10) + "<\/td>";
               content += "<td>" + i.toString(16) + "<\/td>";
               content += "<td>" + i.toString(2) + "<\/td><\/tr>";
            }
            document.write(content);
            </script>
         </tr>
      </table>
   </body>
</html>
```

The `toString()` method of user-defined objects does not convert the object into a meaningful string, but you can create your own method to do just that. For example, if you want to make your custom object's `toString()` method behave like an array's method, define the action of the method and assign that function to a property of the object (as shown in Listing 34-2).

Listing 34-2: **Creating a Custom toString() Method**

```
<html>
   <head>
      <title>Custom toString()</title>
      <script type="text/javascript">
      function customToString() {
         var dataArray = new Array();
```

Continued

Listing 34-2 *(continued)*

```
            var count = 0;
            for (var i in this) {
               dataArray[count++] = this[i];
               if (count > 2) {
                  break;
               }
            }
            return dataArray.join(",");
         }
         var book = {title:"The Aeneid", author:"Virgil", pageCount:543};
         book.toString = customToString;
         </script>
      </head>
      <body>
         <b>A user-defined toString() result:</b>
         <hr />
         <script type="text/javascript">
         document.write(book.toString());
         </script>
      </body>
   </html>
```

When you run Listing 34-2, you can see how the `custom` object's `toString()` handler extracts the values of all elements of the object. You can customize how the data should be labeled and/or formatted.

`unwatch(`*property*`)`
`watch(`*property, handler*`)`

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

To supply the right kind of information to external debuggers, JavaScript in NN4+ implements two global functions that belong to every object — including user-defined objects. The `watch()` function keeps an eye on a desired object and property. If that property is set by assignment, the function invokes another user-defined function that receives information about the property name, its old value, and its new value. The `unwatch()` function turns off the watch functionality for a particular property.

Statements

`//`
`/*...*/`

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Comments are statements that the JavaScript interpreter (or server-side compiler) ignores. However, these statements enable authors to leave notes about how things work in their scripts. While lavish comments are useful to authors during a script's creation and maintenance, the full content of a client-side comment is downloaded with the document. Every

byte of non-operational content of the page takes a bit more time to download. Still, I recommend lots of comments — particularly as you create a script.

JavaScript offers two styles of comments. One style consists of two forward slashes (no spaces between them). JavaScript ignores any characters to the right of those slashes on the same line, even if they appear in the middle of a line. You can stack as many lines of these single-line comments as is necessary to convey your thoughts. I typically place a space between the second slash and the beginning of my comment. The following are examples of valid, one-line comment formats:

```
// this is a comment line usually about what's to come
var a = "Fred";  // a comment about this line
// You may want to capitalize the first word of a comment
// sentence if it runs across multiple lines.
//
// And you can leave a completely blank line, like the one above.
```

For longer comments, it is usually more convenient to enclose the section in the other style of comment. The following comment opens with a forward slash and asterisk (/*) and ends with an asterisk and forward slash (*/). JavaScript ignores all statements in between — including multiple lines. If you want to comment out briefly a large segment of your script for debugging purposes, it is easiest to bracket the segment with these comment symbols. To make these comment blocks easier to find, I generally place these symbols on their own lines as follows:

```
/*
some
  commented-out
    statements
*/
```

If you are developing rather complex documents, you might find using comments a convenient way to help you organize segments of your scripts and make each segment easier to find. For example, you can define a comment block above each function and describe what the function is about, as in the following example.

```
/*-----------------------------------------------
   calculate()
   Performs a mortgage calculation based on
   parameters blah, blah, blah.  Called by blah
   blah blah.
-----------------------------------------------*/
function calculate(form) {
    statements
}
```

const

Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `const` keyword initializes a constant. Unlike a variable, whose data is subject to change while a page loads, a constant's value cannot be modified once it is assigned. It is common practice in many programming languages to define constant identifiers with all uppercase letters, usually with underscore characters to delimit multiple words. This style makes it easier to see a constant's application later in the program.

Example

Listing 34-3 shows how you can use a constant in NN6+/Moz1+. The page conveys temperature data for several cities. (Presumably, this data is updated on the server and fashioned

into an array of data when the user requests the page.) For temperatures below freezing, the temperature is shown in a distinctive text style. Because the freezing temperature is a constant reference point, it is assigned as a constant.

Listing 34-3: **Using the const Keyword**

```
<html>
  <head>
     <title>const(ant)</title>
     <style type="text/css">
     .cold {font-weight:bold; color:blue}
     td {text-align:center}
     </style>
     <script type="text/javascript">
     const FREEZING_F = 32;
     var cities = ["London", "Moscow", "New York", "Tokyo", "Sydney"];
     var tempsF = [33, 12, 20, 40, 75];
     function showData() {
        var tableData = "";
        for (var i = 0; i < cities.length; i++) {
           tableData += "<tr><td>" + cities[i] + "<\/td><td ";
           tableData += (tempsF[i] < FREEZING_F) ? "class='cold'" : "";
           tableData += ">" + tempsF[i] + "<\/td><\/tr>";
        }
        document.getElementById("display").innerHTML = tableData;
     }
     </script>
  </head>
  <body onload="showData()">
     <h1>The const keyword</h1>
     <hr />
     <table id="temps">
        <tr>
           <th>City</th>
           <th>Temperature</th>
        </tr>
        <tbody id="display"></tbody>
     </table>
  </body>
</html>
```

The `const` keyword likely will be adopted in the next version of the ECMA-262 standard and will become part of the JavaScript vernacular in future browsers.

var

Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

Before using any variable, you should declare it (and optionally initialize it with a value) via the `var` statement. If you omit the `var` keyword, the variable is automatically assigned as a global variable within the current document. To keep a variable local to a function, you must declare or initialize the variable with the `var` keyword inside the function's braces.

If you assign no value to a variable, it evaluates to `null`. Because a JavaScript variable is not limited to one variable type during its lifetime, you don't need to initialize a variable to an

empty string or zero unless that initial value helps your scripting. For example, if you initialize a variable as an empty string, you can then use the add-by-value operator (+=) to append string values to that variable in a future statement in the document.

To save statement lines, you can declare and/or initialize multiple variables with a single `var` statement. Separate each *`varName=value`* pair with a comma, as in

```
var name, age, height;  // declare as null
var color = "green", temperature = 85.6; // initialize
```

Variable names (also known as identifiers) must be one contiguous string of characters, and the first character must be a letter. Many punctuation symbols are also banned, but the underscore character is valid and often is used to separate multiple words in a long variable name. All variable names (like most identifiers in JavaScript) are case-sensitive, so you must name a particular variable identically throughout the variable's scope.

WinIE Objects

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

Microsoft prides itself on the integration between Web browser functionality and the Windows operating system. The linkage between browser and OS is most apparent in IE's facilities for accessing ActiveX objects. Microsoft has fashioned several such objects for access to scripters — again, provided the deployment is intended only for Windows versions of Internet Explorer. Some objects also exist as a way to expose some Visual Basic Script (VBScript) functionality to JavaScript. Because these objects are more within the realm of Windows and ActiveX programming, the details and quirks of working with them from WinIE is best left to other venues. But in case you are not familiar with these facilities, the following discussions introduce the basic set of WinIE objects. You can find more details at the Microsoft Developer Network (MSDN) Web site at `http:// msdn.microsoft.com/`.

The objects mentioned here are the `ActiveXObject`, `Dictionary`, `Enumerator`, and `VBArray` objects. Microsoft documents these objects as if they are part of the native JScript language. However, you can be sure that they will remain proprietary certainly to Internet Explorer, if not exclusively for Windows-only versions.

ActiveXObject

`ActiveXObject` is a generic object that allows your script to open and access what Microsoft sometimes calls *automation objects*. An automation object is an executable program that might run on the client or be served from a server. This can include local applications, such as applications from the Microsoft Office suite, executable DLLs (dynamic-link libraries), and so on.

Use the constructor for the `ActiveXObject` to obtain a reference to the object according to the following syntax:

```
var objRef = new ActiveXObject(appName.className[, remoteServerName]);
```

This JScript syntax is the equivalent of the VBScript `CreateObject()` method. You need to know a bit about Windows programming to determine the application name and the classes or types available for that application. For example, to obtain a reference to an Excel worksheet, use this constructor:

```
var mySheet = new ActiveXObject("Excel.Sheet");
```

Once you have a reference to the desired object, you must also know the names of the properties and methods of the object you'll be addressing. You can access much of this information via Microsoft's developer tools, such as Visual Studio .NET or the tools that come with Visual

Basic .NET. These tools enable you to query an object to discover its properties and methods. Unfortunately, an `ActiveXObject`'s properties are not enumerable through a typical JavaScript `for-in` property inspector.

Accessing an `ActiveXObject`, especially one on the client, involves some serious security considerations. The typical security setup for an IE client prevents scripts from accessing client applications, at least not without asking the user if it's okay to do so. While it's foolhardy to state categorically that you cannot perform surreptitious inspection or damage to a client without the user's knowledge (hackers find holes from time to time), it is highly unlikely. In a corporate environment, where some level of access to all clients is desirable, the client may be set up to accept instructions to work with ActiveX objects when they come from trusted sources. The bottom line is that unless you are well versed in Windows programming, don't expect the `ActiveXObject` to become some kind of magic portal that enables you to invade the privacy or security of unsuspecting users.

Dictionary

While the `Dictionary` object is very helpful to VBScript authors, JavaScript already provides the equivalent functionality natively. A `Dictionary` object behaves very much like a JavaScript array that has string index values (similar to a Java hash table), although numeric index values are also acceptable in the `Dictionary`. Indexes are called *keys* in this environment. VBScript arrays do not have this facility natively, so the `Dictionary` object supplements the language for the sake of convenience. Unlike a JavaScript array, however, you must use the various properties and methods of the `Dictionary` object to add, access, or remove items from it.

You create a `Dictionary` object via `ActiveXObject` as follows:

```
var dict = new ActiveXObject("Scripting.Dictionary");
```

You must create a separate `Dictionary` object for each array. Table 34-2 lists the properties and methods of the `Dictionary` object. After you create a blank `Dictionary` object, populate it via the `Add()` method for each entry. For example, the following statements create a `Dictionary` object to store U.S. state capitals:

```
var stateCaps = new ActiveXObject("Scripting.Dictionary");
stateCaps.Add("Illinois", "Springfield");
```

You can then access an individual item via the `Key` property (which, thanks to its VBScript heritage, looks more like a JavaScript method). One convenience of the `Dictionary` object is the `Keys()` method, which returns an array of all the keys in the dictionary — something that a string-indexed JavaScript array could use.

Table 34-2: Dictionary Object Properties and Methods

Property	*Description*
`Count`	Integer number of entries in the dictionary (read-only)
`Item("key")`	Reads or writes a value for an entry whose name is *key*
`Key("key")`	Assigns a new key name to an entry
Method	***Description***
`Add("key", value)`	Adds a value associated with a unique key name
`Exists("key")`	Returns Boolean *true* if *key* exists in dictionary

Method	Description
`Items()`	Returns `VBArray` of values in dictionary
`Keys()`	Returns `VBArray` of keys in dictionary
`Remove("key")`	Removes *key* and its value
`RemoveAll()`	Removes all entries

Enumerator

An `Enumerator` object provides JavaScript with access to collections that otherwise do not allow direct access to their items via index number or name. This object isn't necessary when working with DOM collections, such as `document.all`, because you can use the `item()` method to obtain a reference to any member of the collection. But if you are scripting ActiveX objects, some of these objects' methods or properties may return collections that cannot be accessed through this mechanism or the JavaScript `for-in` property inspection technique. Instead, you must wrap the collection inside an `Enumerator` object.

To wrap a collection in an `Enumerator`, invoke the constructor for the object, passing the collection as the parameter:

```
var myEnum = new Enumerator(someCollection);
```

This enumerator instance must be accessed via one of its four methods to position a "pointer" to a particular item and then extract a copy of that item. In other words, you don't access a member directly (that is, by diving into the collection with an item number to retrieve). Instead, you move the pointer to the desired position and then read the item value. As you can see from the list of methods in Table 34-3, this object is truly intended for looping through the collection. Pointer control is limited to positioning it at the start of the collection and incrementing its position along the collection by one:

```
myEnum.moveFirst();
for (; !myEnum.atEnd(); myEnum.moveNext()) {
   val = myEnum.item();
   // more statements that work on value
}
```

Table 34-3: Enumerator Object Methods

Method	Description
`atEnd()`	Returns `true` if pointer is at end of collection
`item()`	Returns value at current pointer position
`moveFirst()`	Moves pointer to first position in collection
`moveNext()`	Moves pointer to next position in collection

VBArray

The `VBArray` object provides JavaScript access to Visual Basic *safe arrays*. Such an array is read-only and is commonly returned by ActiveX objects. Such arrays can be composed in

VBScript sections of client-side scripts. Visual Basic arrays by their very nature can have multiple dimensions. For example, the following code creates a three-by-two VB array:

```
<script type="text/vbscript">
Dim myArray(2, 1)
myArray(0, 0) = "A"
myArray(0, 1) = "a"
myArray(1, 0) = "B"
myArray(1, 1) = "b"
myArray(2, 1) = "C"
myArray(2, 2) = "c"
</script>
```

Once you have a valid VB array, you can convert it to an object that the JScript interpreter can't choke on:

```
<script type="text/javascript">
var theVBArray = new VBArray(myArray);
</script>
```

Global variables from one script language block can be accessed by another block, even in a different language. But at this point, the array is not in the form of a JavaScript array yet. You can either convert it to such via the `VBArray.toArray()` method or access information about the `VBArray` object through its other methods (described briefly in Table 34-4). Once you convert a `VBArray` to a JavaScript array, you can then iterate through the values just like any JavaScript array.

Table 34-4: VBArray Object Methods

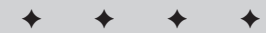
Method	*Description*
`dimensions()`	Returns number of dimensions of the original array
`getItem(`*`dim1`*`[, `*`dim2`*`[,...`*`dimN`*`]])`	Returns value at array location defined by dimension addresses
`ibound(`*`dim`*`)`	Returns lowest index value for a given dimension
`toArray()`	Returns JavaScript array version of `VBArray`
`ubound(`*`dim`*`)`	Returns highest index value for a given dimension

When you use the `toArray()` method and the source array has multiple dimensions, values from dimensions after the first "row" are simply appended to the JavaScript array with no nesting structure.

✦ ✦ ✦

CHAPTER 35

Body Text Objects

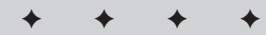


In This Chapter

Objects that display running body text in documents

Using the NN/Mozilla `Range` and IE `TextRange` objects

Scripting search-and-replace actions



A large number of HTML elements fall into a catchall category of elements whose purposes are slightly more targeted than contextual elements covered in Chapter 15. In this group are some very widely used elements, such as the `h1` through `h6` header elements, plus several elements that are not yet widely used because their full support may be lacking in even some of the most modern browsers. In this chapter, you find all sorts of text-related objects, excluding those objects that act as form controls (text boxes and such, which are covered in Chapter 23). For the most part, properties, methods, and event handlers of this chapter's objects are the generic ones covered in Chapter 15. Only those items that are unique to each object are covered in this chapter (as will be the case in all succeeding chapters).

Beyond the HTML element objects covered in this chapter, you also meet the `TextRange` object, first introduced in IE4, and the corresponding `Range` object from the W3C DOM. This object is a very powerful one for scripters because it allows scripts to work very closely with body content — not in terms of, for example, the `innerText` or `nodeValue` properties of elements, but rather in terms of the text as it appears on the page in what users see as paragraphs, lists, and the like. The `TextRange` and `Range` objects essentially give your scripts cursor control over running body text for functions, such as cutting, copying, pasting, and applications that extend from those basic operations — search and replace, for instance. Bear in mind that everything you read in this chapter requires at minimum the dynamic object models of IE4+ and NN6+/W3C; some items require IE5+. Unfortunately, the IE `TextRange` object is not implemented in MacIE5.

blockquote and q Element Objects

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`cite`		

Syntax

Accessing `blockquote` or `q` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

About these objects

The `blockquote` element is a special-purpose text container. Browsers typically start the content on its own line in the body and indent on both the left and right margins approximately 40 pixels. An inline quotation can be encased inside a `q` element, which does not force the quoted material to start on the next line.

From an object point of view, the only property that distinguishes these two objects from any other kind of contextual container is the `cite` property, which comes from the HTML 4.0 `cite` attribute. This attribute simply provides a URL reference for the citation and does not act as an `src` or `href` attribute to load an external document.

Property

`cite`

Value: String. Read/Write
Compatibility: WinIE6, MacIE5+, NN6+, Moz1+, Safari1+

The `cite` property can contain a URL (as a string) that points to the source of the quotation in the `blockquote` or `q` element. Future browsers may provide some automatic user interface link to the source document, but none of the browsers that support the `cite` property do anything special with this information.

br Element Object

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`clear`		

Syntax

Accessing `br` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

About this object

The `br` element forces a carriage return and line feed for rendered content on the page. This element does not provide the same kind of vertical spacing that goes between paragraphs in a series of `p` elements. Only one attribute (`clear`) distinguishes this element from generic HTML elements and objects.

Property

`clear`

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `clear` property defines how any text in an element following the `br` element wraps around a floating element (for example, an image set to float along the right margin). While recent browsers expose this property, the attribute on which it is based is deprecated in the HTML 4.0 specification in an effort to encourage the use of the `clear` stylesheet attribute for a `br` element.

Values for the `clear` property can be one of the following strings: `all`, `left`, or `right`.

Related Items: `clear` stylesheet property.

font Element Object

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`color`		
`face`		
`size`		

Syntax

Accessing `font` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

About this object

In a juxtaposition of standards implementations, the `font` element is exposed as an object only in browsers that also support Cascading Style Sheets as the preferred way to control font faces, colors, and sizes. This doesn't mean that you shouldn't use `font` elements in your page with modern browsers — using this element may be necessary for a single page that

needs to be backward compatible with older browsers. But it does present a quandary for scripters who want to use scripts to modify font characteristics of body text after the page has loaded. A good rule of thumb to follow is to use the `font` element (and script the `font`-HTML element object's properties) when the page must work in all browsers; use stylesheets (and their scriptable properties) on pages that will be running exclusively in IE4+ and NN6+/W3C.

Properties

`color`

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

A `font` object's text color can be controlled via the `color` property. Values can be either hexadecimal triplets (for example, `#FFCCFF`) or the plain-language color names recognized by most browsers. In either case, the values are case-insensitive strings.

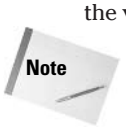
Example

Listing 35-1 contains a page that demonstrates changes to the three `font` element object properties: `color`, `face`, and `size`. Along the way, you can see an economical use of the `setAttribute()` method to do the work for all of the property changes. This page loads successfully in all browsers, but the `select` lists make changes to the text only in IE4+ and NN6+/W3C.

A `p` element contains a nested `font` element that encompasses three words whose appearance is controlled by three `select` lists. Each list controls one of the three `font` object properties, and their `name` attributes are strategically assigned the names of the properties (as you see in a moment). `value` attributes for `option` elements contain strings that are to be assigned to the various properties. Each `select` element invokes the same `setFontAttr()` function, passing a reference to itself so that the function can inspect details of the element.

The first task of the `setFontAttr()` function is to make sure that only browsers capable of treating the `font` element as an object get to the meat of the function. The test for the existence of `document.all` and the `myFONT` element blocks all older browsers from changing the font characteristics.

For suitably equipped browsers, the function next extracts the string from the `value` property of the `select` object that was passed to the function. If a selection is made (meaning other than the first, empty one), the single nested statement uses the `setAttribute()` method to assign the value to the attribute whose name matches the name of the `select` element.



An odd bug in MacIE5 doesn't let the rendered color change when changing the `color` property. But the setting is valid, as proven by selecting any of the other two property choices.

Listing 35-1: Dynamically Changing Font Properties

```
<html>
   <head>
      <title>Font Object Properties</title>
      <script type="text/javascript">
      // one function does all!
```

```
      function setFontAttr(select) {
         var choice = select.options[select.selectedIndex].value;
         if (choice) {
            document.getElementById("myFONT").setAttribute(select.name,
               choice);
         }
      }
      </script>
   </head>
   <body>
      <h1>Font Object Properties</h1>
      <br />
      <p>This may look like a simple sentence, but <font id="myFONT">THESE
         THREE WORDS</font> are contained by a FONT element.</p>
      <form>
         Select a text color: <select name="color"
         onchange="setFontAttr(this)">
            <option></option>
            <option value="red">Red</option>
            <option value="green">Green</option>
            <option value="blue">Blue</option>
            <option value="#FA8072">Some Hex Triplet Value</option>
         </select><br />
         Select a font face: <select name="face" onchange="setFontAttr(this)">
            <option></option>
            <option value="Helvetica">Helvetica</option>
            <option value="Times">Times</option>
            <option value="Comic Sans MS, sans-serif">Comic Sans MS,
            sans-serif</option>
            <option value="Courier, monospace">Courier, monospace</option>
            <option value="Zapf Dingbats, serif">Zapf Dingbats, serif</option>
         </select><br />
         Select a font size: <select name="size" onchange="setFontAttr(this)">
            <option></option>
            <option value="3">3 (Default)</option>
            <option value="+1">Increase Default by 1</option>
            <option value="-1">Decrease Default by 1</option>
            <option value="1">Smallest</option>
            <option value="7">Biggest</option>
         </select>
      </form>
   </body>
</html>
```

Related Items: `color` stylesheet attribute.

face

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

A `font` object's font face is controllable via the `face` property. Just as the `face` attribute (and the corresponding `font-family` stylesheet attribute), you can specify one or more font names in a comma-delimited string. Browsers start with the leftmost font face and look for a match in the client computer's system. The first matching font face that is found in the client system is applied to the text surrounded by the `font` element. You should list the most specific fonts first,

and generally allow the generic font faces (`sans-serif`, `serif`, and `monospace`) to come last; that way you exert at least some control over the look of the font on systems that don't have your pretty fonts. If you know that Windows displays a certain font you like and the Macintosh has something that corresponds to that font but with a different name, you can specify both names in the same property value. The browser skips over font face names not currently installed on the client.

Example

See Listing 35-1 for an example of values that can be used to set the `face` property of a `font` element object. While you will notice visible changes to most choices on the page, the font face selections may not change from one choice to another, since that all depends on the fonts that are installed on your PC.

Related Items: `font-family` style sheet attribute.

`size`

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The size of text contained by a `font` element can be controlled via the `size` property. Unlike the more highly recommended `font-size` stylesheet attribute, the `size` property of the `font` element object (and its corresponding `SIZE` attribute) are restricted to the relative font size scale imposed by early HTML implementations: a numbering scale from 1 to 7.

Values for the `size` property are strings, even though most of the time they are single numeral values. You can also specify a size relative to the default value by including a plus or minus sign before the number. For example, if the default font size (as set by the browser's user preferences) is 3, you can bump up the size of a text segment by encasing it inside a `font` element and then setting its `size` property to `"+2"`.

For more accurate font sizing using units, such as pixels or points, use the `font-size` stylesheet attribute.

Example

See Listing 35-1 for an example of values that can be used to set the `size` property of a `font` element object. Notice that incrementing or decrementing the `size` property is applied only to the size assigned to the `size` attribute of the element (or the default, if none is specified) and not the current setting adjusted by script.

Related Items: `font-size` style sheet attribute.

h1...h6 Element Objects

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`align`		

Syntax

Accessing h1 through h6 element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

About these objects

The so-called "heading" elements (denoted by h1, h2, h3, h4, h5, and h6) provide shortcuts for formatting up to six different levels of headings and subheadings. While you can simulate the appearance of these headings with p elements and stylesheets, the heading elements very often contain important contextual information about the structure of the document. With the IE5+ and NN6+/W3C powers of inspecting the node hierarchy of a document, a script can generate its own table of contents or outline of a very long document by looking for elements whose nodeName properties are in the h*n* family. Therefore, it is a good idea to continue using these elements for contextual purposes, even if you intend to override the default appearance by way of stylesheet templates.

As for the scriptable aspects of these six objects, they are essentially the same as the generic contextual objects with the addition of the align property. Because each h*n* element is a block-level element, you can use stylesheets to set their alignment rather than the corresponding attribute or property. The choice is up to you.

Property

align

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

String values of the align property control whether the heading element is aligned with the left margin (left), center of the page (center), or right margin (right). The corresponding align attribute is deprecated in HTML 4.0 in favor of the text-align stylesheet attribute.

Related Items: text-align stylesheet attribute.

hr Element Object

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
align		
color		
noShade		
width		

Syntax

Accessing `hr` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

About this object

The `hr` element draws a horizontal rule according to size, dimension, and alignment characteristics normally set by the attributes of this element. Stylesheets can also specify many of those settings, the latter route being recommended for pages that will be loaded exclusively in pages that support CSS. In IE4+ and NN6+/W3C DOM browsers, your scripts can modify the appearance of an `hr` element either directly through element object properties or through stylesheet properties. To reference a specific `hr` element by script, you must assign an `id` attribute to the element — a practice that you are probably not accustomed to observing.

Properties

`align`

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

An `hr` object's horizontal alignment can be controlled via the `align` property. String values enable you to set it to align with the left margin (`left`), the center of the page (`center`), or right margin (`right`). By default, the element is centered.

Example

Listing 35-2 contains a page that demonstrates the changes to the five `hr` element object properties: `align`, `color`, `noShade`, `size`, and `width`. Along the way, you can see an economical use of the `setAttribute()` method to do the work for all of the property changes. This page loads successfully in all browsers, but the `select` lists make changes to the text only in IE4+ and NN6+/W3C DOM browsers (because they treat the element as an object).

An `hr` element (whose `id` is `myHR`) is displayed with the browser default settings (100% width, centered, and its "magic" color). Each list controls one of the five `hr` object properties, and their `name` attributes are strategically assigned the names of the properties (as you see in a moment). `value` attributes for `option` elements contain strings that are to be assigned to the various properties. Each `select` element invokes the same `setHRAttr()` function, passing a reference to itself so that the function can inspect details of the element.

The first task of the `setHRAttr()` function is to make sure that only browsers capable of treating the `hr` element as an object get to the meat of the function. As the page loads, the `document.all` property is set for NN6+/W3C using a normalization technique described in Chapter 14.

For suitably equipped browsers, the function next reads the string from the `value` property of the `select` object that is passed to the function. If a selection is made (that is, other than the first, empty one), the single, nested statement uses the `setAttribute()` method to assign the value to the attribute whose name matches the name of the `select` element.

Listing 35-2: **Controlling hr Object Properties**

```
<html>
   <head>
      <title>hr Object Properties</title>
      <script type="text/javascript">
      // one function does all!
      function setHRAttr(select) {
         var choice = select.options[select.selectedIndex].value;
         if (choice) {
            document.getElementById("myHR").setAttribute(select.name, choice);
         }
      }
      </script>
   </head>
   <body>
      <h1>hr Object Properties</h1>
      <br />
      <p>Here is the hr element you will be controlling:</p>
      <hr id="myHR" />
      <form>
         Select an alignment: <select name="align" onchange="setHRAttr(this)">
            <option></option>
            <option value="left">Left</option>
            <option value="center">Center</option>
            <option value="right">Right</option>
         </select><br />
         Select a rule color (IE only): <select name="color"
         onchange="setHRAttr(this)">
            <option></option>
            <option value="red">Red</option>
            <option value="green">Green</option>
            <option value="blue">Blue</option>
            <option value="#FA8072">Some Hex Triplet Value</option>
         </select><br />
         Select a rule shading: <select name="noShade"
         onchange="setHRAttr(this)">
            <option></option>
            <option value="true">No Shading</option>
            <option value="false">Shading</option>
         </select><br />
         Select a rule height: <select name="size" onchange="setHRAttr(this)">
            <option></option>
            <option value="2">2 (Default)</option>
            <option value="4">4 Pixels</option>
            <option value="10">10 Pixels</option>
         </select><br />
         Select a rule width: <select name="width" onchange="setHRAttr(this)">
            <option></option>
            <option value="100%">100% (Default)</option>
            <option value="80%">80%</option>
            <option value="300">300 Pixels</option>
         </select>
      </form>
   </body>
</html>
```

Related Items: `text-align` stylesheet attribute.

color

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

An `hr` object's color can be controlled via the `color` property. Values can be either hexadecimal triplets (for example, `#FFCCFF`) or the plain-language color names recognized by most browsers. In either case, the values are case-insensitive strings. If you change the color from the default, the default shading (3-D effect) of the rule disappears. I have yet to find the magic value that lets you return the color to the browser default after it has been set to another color.

Example

See Listing 35-2 earlier in this chapter for an example of values that can be used to set the `color` property of an `hr` element object.

Related Items: `color` stylesheet attribute.

noShade

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

A default `hr` element is displayed with a kind of three-dimensional effect, called *shading*. You can turn shading off under script control by setting the `noShade` property to `true`. But be aware that in IE4+, the `noShade` property is a one-way journey: You cannot restore shading after it is removed. Moreover, default shading is lost if you assign a different color to the rule.

Example

See Listing 35-2 earlier in this chapter for an example of values that can be used to set the `noShade` property of an `hr` element object. Because of the buggy behavior associated with setting this property, adjusting the property in the example has unexpected (and usually undesirable) consequences.

Related Items: `color` stylesheet attribute.

size

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The size of an `hr` element is its vertical thickness, as controlled via the `size` property. Values are integers, representing the number of pixels occupied by the rule. Safari 1.0 does not change the element's rendered size via this property.

Example

See Listing 35-2 earlier in this chapter for an example of values that can be used to set the `size` property of an `hr` element object.

width

Value: Integer or string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The width of an `hr` element is controlled via the `width` property. By default, the element occupies the entire width of its parent container (usually the Body).

You can specify width as either an absolute number of pixels (as an integer) or as a percentage of the width of the parent container. Percentage values are strings that include a trailing percent character (`%`).

Example

See Listing 35-2 earlier in this chapter for an example of values that can be used to set the `width` property of an `hr` element object.

Related Items: `width` stylesheet attribute.

label Element Object

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`accessKey`		
`form`		
`htmlFor`		

Syntax

Accessing `label` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

About this object

The `label` element lets you assign a contextual relationship between a form control (text field, radio button, select list, and so on) and the otherwise freestanding text that is used to label the control on the page. This element does not control the rendering or physical relationship between the control and the label — the HTML source code order does that. Wrapping a form control label inside a `label` element is important if scripts will be navigating the element hierarchy of a page's content and the relationship between a form control and its label is important to the results of the document parsing.

Properties

accessKey

Value: String. Read/Write
Compatibility: WinIE4+, MacIE5+, NN6+, Moz1+, Safari1+

For most other HTML element objects, the `accessKey` property description is covered in the generic element property descriptions of Chapter 15. The function of the property for the

label object is the same as the IE implementation for all other elements. The single-character string value is the character key to be used in concert with the OS- and browser-specific modifier key (for example, Ctrl in IE for Windows) to bring focus to the form control associated with the label. This value is best set initially via the accesskey attribute for the label element.

Related Items: accessKey property of generic elements.

form

Value: Form object reference. Read-Only
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The form property of a label element object returns a reference to the form object that contains the form control with which the label is associated. This property can be useful in a node parsing script that wants to retrieve the form container from the perspective of the label rather than from the form control. The form object reference returned from the label element object is the same form object reference returned by the form property of any form control object.

Related Items: form property of input element objects.

htmlFor

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The htmlFor property is a string that contains the id of the form control element with which the label is associated. This value is normally set via the htmlfor attribute in the label element's tag. Modifying this property does not alter the position or rendering of the label, but it does change the relationships between label and control.

marquee Element Object

Compatibility: WinIE4+, MacIE4+, NN7+, Moz1+, Safari-

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
behavior	start()	onbounce
bgColor	stop()	onfinish
direction		onstart
height		
hspace		
loop		
scrollAmount		
scrollDelay		
trueSpeed		
vspace		
width		

Syntax

Accessing `marquee` element object properties or methods:

```
(IE4+) [window.]document.all.elemID.property | method([parameters])
(IE5+) [window.]document.getElementById("elemID").property |
method([parameters])
```

About this object

The `marquee` element is a Microsoft proprietary element that displays scrolling text within a rectangle specified by the `width` and `height` attributes of the element. Text that scrolls in the element goes between the element's `start` and `end` tags. The IE4+ object model exposes the element and many properties to the object model for control by script. The element and some of its scriptability is implemented in NN7+/Moz1+.

Properties

behavior

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `behavior` property controls details in the way scrolled text moves within the scrolling space. Values for this property are one of the following three strings: `alternate`, `scroll`, and `slide`. NN7/Moz allows only `alternate`. When set to `alternate`, scrolling alternates between left and right (or up and down, depending on the `direction` property setting). A value of `scroll` means that the text marches completely to and through the space before appearing again. And a value of `slide` causes the text to march into view until the last character is visible. When the `slide` value is applied as a property (instead of as an attribute value in the tag), the scrolling stops when the text reaches an edge of the rectangle. Default behavior for the `marquee` element is the equivalent of `scroll`.

Example

Listing 35-3 contains a page that demonstrates the changes to several `marquee` element object properties: `behavior`, `bgColor`, `direction`, `scrollAmount`, and `scrollDelay`. NN7+/Moz do not react to all property settings. See the description of Listing 35-1 for details on the attribute setting script.

Listing 35-3: Controlling marquee Object Properties

```
<html>
   <head>
      <title>marquee Object Properties</title>
      <script type="text/javascript">
      // one function does all!
      function setMARQUEEAttr(select) {
         var choice = select.options[select.selectedIndex].value;
         if (choice) {
            document.getElementById("myMARQUEE").setAttribute(select.name,
               choice);
         }
      }
```

Continued

Listing 35-3 *(continued)*

```
      </script>
   </head>
   <body>
      <h1>marquee Object Properties</h1>
      <br />
      <hr />
      <marquee id="myMARQUEE" width="400" height="24">This is the marquee
      element object you will be controlling.</marquee>
      <form>
         <input type="button" value="Start Marquee"
         onclick="document.getElementById('myMARQUEE').start()" /> <input
         type="button" value="Stop Marquee"
         onclick="document.getElementById('myMARQUEE').stop()" /><br />
         Select a behavior: <select name="behavior"
         onchange="setMARQUEEAttr(this)">
            <option></option>
            <option value="alternate">Alternate</option>
            <option value="scroll">Scroll</option>
            <option value="slide">Slide</option>
         </select><br />
         Select a background color: <select name="bgColor"
         onchange="setMARQUEEAttr(this)">
            <option></option>
            <option value="red">Red</option>
            <option value="green">Green</option>
            <option value="blue">Blue</option>
            <option value="#FA8072">Some Hex Triplet Value</option>
         </select><br />
         Select a scrolling direction: <select name="direction"
         onchange="setMARQUEEAttr(this)">
            <option></option>
            <option value="left">Left</option>
            <option value="right">Right</option>
            <option value="up">Up</option>
            <option value="down">Down</option>
         </select><br />
         Select a scroll amount: <select name="scrollAmount"
         onchange="setMARQUEEAttr(this)">
            <option></option>
            <option value="4">4</option>
            <option value="6">6 (Default)</option>
            <option value="10">10</option>
         </select><br />
         Select a scroll delay: <select name="scrollDelay"
         onchange="setMARQUEEAttr(this)">
            <option></option>
            <option value="50">Short</option>
            <option value="85">Normal</option>
            <option value="125">Long</option>
         </select>
      </form>
   </body>
</html>
```

Related Items: `direction` property of `marquee` object.

bgColor

Value: Hexadecimal triplet or color name string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The bgColor property determines the color of the background of the marquee element's rectangular space. To set the color of the text, either surround the marquee element with a font element or apply the color stylesheet attribute to the marquee element. Values for all color properties can be either the common HTML hexadecimal triplet value (for example, "#00FF00") or any of the Netscape color names (a list is available at http://developer.netscape.com/docs/manuals/htmlguid/colortab.htm).

Example

See Listing 35-3 earlier in this chapter for an example of how to apply values to the bgColor property.

direction

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN7+, Moz1+, Safari-

The direction property lets you get or set the horizontal or vertical direction in which the scrolling text moves. Four possible string values are left, right, down, up. NN7/Moz observe left and right only. The default value is left.

Example

See Listing 35-3 earlier in this chapter for an example of how to apply values to the direction property.

Related Items: behavior property of marquee object.

height
width

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN7+, Moz1+, Safari-

The height and width properties enable you to get or set the pixel size of the rectangle occupied by the element. NN7/Moz implement width only. You can adjust each property independently of the other, but like most attribute-inspired properties of IE objects, if no height or width attributes are defined in the element's tag, you cannot use these properties to get the size of the element as rendered by default.

hspace
vspace

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The hspace and vspace properties let you get or set the amount of blank margin space surrounding the marquee element. Adjustments to the hspace property affect both the left and right (horizontal) margins of the element; vspace governs both top and bottom (vertical) margins. Margin thicknesses are independent of the height and width of the element.

loop

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `loop` property allows you to discover the number of times the `marquee` element was set to repeat its scrolling according to the `loop` attribute. Although this property is read/write, modifying it by script does not cause the text to loop only that number of times more before stopping. Treat this property as read-only.

scrollAmount
scrollDelay

Value: Integers. Read/Write
Compatibility: WinIE4+, MacIE4+, NN7+, Moz1+, Safari-

The `scrollAmount` and `scrollDelay` properties control the perceived speed and scrolling smoothness of the `marquee` element text. The number of pixels between redrawings of the scrolling text is controlled by the `scrollAmount` property. The smaller the number, the less jerky the scrolling is (the default value is `6`). At the same time, you can control the time in milliseconds between each redrawing of the text with the `scrollDelay` property. The smaller the number, the more frequently redrawing is performed (the default value is 85 or 90, depending on the operating system). Thus, a combination of low `scrollAmount` and `scrollDelay` property values presents the smoothest (albeit slow) perceived scrolling.

Example

See Listing 35-3 earlier in this chapter for an example of how to apply values to the `scrollAmount` and `scrollDelay` properties.

Related Items: `trueSpeed` property of `marquee` object.

trueSpeed

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

IE has a built-in regulator that prevents `scrolldelay` attribute or `scrollDelay` property settings below 60 from causing the `marquee` element text to scroll too quickly. But if you genuinely want to use a speed faster than 60 (meaning, a value lower than 60), then also set the `trueSpeed` property to `true`.

Related Items: `scrollDelay` property of `marquee` object.

Methods

start()
stop()

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN7+, Moz1+, Safari-

Scripts can start or stop (pause) a `marquee` element via the `start()` and `stop()` methods. Neither method takes parameters, and you are free to invoke them as often as you like after the page loads. Be aware that the `start()` method does not trigger the `onstart` event handler for the object.

Example

See Listing 35-3 earlier in this chapter for examples of both the `start()` and `stop()` methods, which are invoked in event handlers of separate controlling buttons on the page. Notice, too, that when you have the behavior set to `slide`, stopping and restarting the `marquee` does not cause the scroll action to start from a blank region.

Event Handlers

onbounce

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `onbounce` event handler fires only when the `marquee` element has its behavior set to `alternate`. In that back-and-forth mode, each time the text reaches a boundary and is about to start its return trip, the `onbounce` event fires. If you truly want to annoy your users, you could have the `onbounce` event handlers play a sound at each bounce (I'm kidding — please don't do this).

Related Items: `behavior` property of `marquee` object.

onfinish

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `onfinish` event handler fires only when the `marquee` element has its loop set to a specific value of `1` or greater. After the final text loop has completed, the `onfinish` event fires.

Related Items: `loop` property of `marquee` object.

onstart

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `onstart` event handler fires as the `marquee` element begins its scrolling, but only as a result of the page loading. The `start()` method does not trigger this event handler.

Related Items: `start()` method of `marquee` object.

Range Object

Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

Properties	*Methods*	*Event Handlers*
`collapsed`	`cloneContents()`	
`commonAncestorContainer`	`cloneRange()`	
`endContainer`	`collapse()`	
`endOffset`	`compareBoundaryPoints()`	
	`compareNode()`	
	`comparePoint()`	
`startContainer`	`createContextualFragment()`	

Range

Properties	*Methods*	*Event Handlers*
startOffet	deleteContents()	
	detach()	
	extractContents()	
	insertNode()	
	intersectsNode()	
	isPointInRange()	
	selectNode()	
	selectNodeContents()	
	setEnd()	
	setEndAfter()	
	setEndBefore()	
	setStart()	
	setStartAfter()	
	setStartBefore()	
	surroundContents()	
	toString()	

Syntax

Creating a `Range` object:

```
var rangeRef = document.createRange();
```

Accessing `Range` object properties or methods:

```
(NN6+/Moz1+)    rengeRef.property | method([parameters])
```

About this object

The `Range` object is the W3C DOM (Level 2) version of what Microsoft had implemented earlier as its `TextRange` object. A number of important differences (not the least of which is an almost entirely different property and method vocabulary) distinguish the behaviors and capabilities of these two similar objects. Although Microsoft participated in the W3C DOM Level 2 working groups, no participant from the company is credited on the DOM specification chapter regarding the `Range` object. Because the W3C version has not been implemented as of IE6, it is unknown if or when IE will eventually implement the W3C version. In the meantime, see the WinIE `TextRange` object section later in this chapter for comparisons between the two objects. Neither the W3C DOM `Range` nor Microsoft `TextRange` objects are implemented in MacIE5.

The purpose of the W3C DOM `Range` object is to provide hooks to a different "slice" of content (most typically a portion of a document's content) that is not necessarily restricted to the node hierarchy (tree) of a document. While a `Range` object can be used to access and modify nodes and elements, it can also transcend node and element boundaries to encompass arbitrary segments of a document's content. The content contained by a range is sometimes referred to as a *selection*, but this does not mean that the text is highlighted on the page, such as a user selection. Instead, the term "selection" here means a segment of the document's content that can be

addressed as a unit, separate from the node tree of the document. As soon as the range is created, a variety of methods let scripts examine, modify, remove, replace, and insert content on the page.

A range object (meaning, an instance of the static `Range` object) has a start point and an end point, which together define the boundaries of the range. The points are defined in terms of an offset count of positions within a container. These counts are usually character positions within text nodes (ignoring any HTML tag or attribute characters), but when both boundaries are at the edges of the same node, the offsets may also be counts of nodes within a container that surrounds both the start and end points. An example helps clarify these concepts.

Consider the following simplified HTML document:

```
<html>
   <body>
   <p>This paragraph has an <em>emphasized</em> segment.</p>
   </body>
</html>
```

You can create a range that encompasses the text inside the `em` element from several points of view, each with its own offset counting context:

1. *From the `em` element's only child node (a text node).* The offset of the start point is zero, which is the location of the insertion point in front of the first character (lowercase "e"); the end point offset is 10, which is the character position (zero-based) following the lowercase "d".

2. *From the `em` element.* The point of view here is that of the child text node inside the `em` element. Only one node exists here, and the offset for the start point is 0, while the offset for the end point is 1.

3. *From the `p` element's child nodes (two text nodes and an element node).* You can set the start point of a range to the very end (counting characters) of the first child text node of the `p` element; you can then set the end point to be in front of the first character of the last child text node of the `p` element. The resulting range encompasses the text within the `em` element.

4. *From the `p` element.* From the point of view of the `p` element, the range can be set with an offset starting with 1 (the second node nested inside the `p` element) and ending with 2 (the start of the third node).

Although these different points of view provide a great deal of flexibility, they also can make it more difficult to imagine how you can use this power. The W3C vocabulary for the `Range` methods, however, helps you figure out what kind of offset measure to use.

A range object's start point could be in one element, and its end point in another. For example, consider the following HTML:

```
<p>And now to introduce our <em>very special</em> guest:</p>
```

If the text shown in boldface indicates the content of a range object, you can see that the range crosses element boundaries in a way that would make HTML element or node object properties difficult to use for replacing that range with some other text. The W3C specification provides guidelines for browser makers on how to handle the results of removing or inserting HTML content that crosses node borders.

An important aspect of the `Range` object is that the size of a range can be zero or more characters. Start and end points always position themselves between characters. When the start point and end point of a range are at the same location, the range acts like a text insertion pointer.

Working with ranges

To create a range object, use the `document.createRange()` method and assign the range object returned by this method to a variable that you can use to control the range:

```
var rng = document.createRange();
```

With an active range stored in a variable, you can use many of the object's methods to adjust the start and end points of the range. If the range is to consist of all of the contents of a node, you have two convenience methods that do so from different points of view: `selectNode()` and `selectNodeContents()`. The sole parameter passed with both methods is a reference to the node whose contents you want to turn into a range. The difference between the two methods is how the offset properties of the range are calculated as a result (see the discussion about these methods later in the chapter for details). Another series of methods (`setStartBefore()`, `setStartAfter()`, `setEndBefore()`, and `setEndAfter()`) let you adjust each end point individually to a position relative to a node boundary. For the most granular adjustment of boundaries, the `setStart()` and `setEnd()` methods let you specify a reference node (where to start counting the offset) and the offset integer value.

If you need to select an insertion point (for example, to insert some content into an existing node), you can position either end point where you want it, and then invoke the `collapse()` method. A parameter determines whether the collapse should occur at the range's start or end point.

A suite of other methods lets your scripts work with the contents of a range directly. You can copy (`cloneContents()`), delete (`deleteContents()`, `extractContents()`), insert a node (`insertNode()`), and even surround a range's contents with a new parent node (`surroundContents()`). Several properties let your scripts examine information about the range, such as the offset values, the containers that hold the offset locations, whether the range is collapsed, and a reference to the next outermost node that contains both the start and end points.

Mozilla adds a proprietary method to the `Range` object (which is actually a method of an object that is built around the `Range` object) called `createContextualFragment()`. This method lets scripts create a valid node (of type `DocumentFragment`) from arbitrary strings of HTML content — a feature that the W3C DOM does not (yet) offer. This method was devised at first as a substitute for what eventually became the NN6+/Moz `innerHTML` property.

Using the `Range` object can be a bit tedious, because it often requires a number of script statements to execute an action. Three basic steps are generally required to work with a `Range` object:

1. Create the text range.
2. Set the start and end points.
3. Act on the range.

As soon as you are comfortable with this object, you will find it provides a lot of flexibility in scripting interaction with body content. For ideas about applying the `Range` object in your scripts, see the examples that accompany the descriptions of individual properties and methods in the following sections.

The Evaluator (Chapter 13) automatically initializes a W3C DOM `Range` object in browsers that support the feature. You can access the object via the `rng` global variable to work with examples in the following sections.

Properties

collapsed

Value: Boolean. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `collapsed` property reports whether a range has its start and end points set to the same position in a document. If the value is `true`, the range's start and end containers are the same and the offsets are also the same. You can use this property to verify that a range is in the form of an insertion pointer just prior to inserting a new node:

```
if (rng.collapsed) {
   rng.insertNode(someNewNodeReference);
}
```

Example

Use The Evaluator's predefined `rng` object to experiment with the `collapsed` property. Reload the page and set the range to encompass a node:

```
rng.selectNode(document.body)
```

Enter `a.collapsed` into the top text box. The expression returns `false` because the end points of the range are not the same.

Related Items: `endContainer`, `endOffset`, `startContainer`, `startOffset` properties; `Range.collapse()` method.

commonAncestorContainer

Value: Node object reference. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `commonAncestorContainer` property returns a reference to the document tree node that both the start and end points have in common. It is not uncommon for a range's start point to be in one node and the end point to be in another. Yet a more encompassing node most likely contains both of those nodes, perhaps even the `document.body` node. The W3C DOM specification also calls the shared ancestor node the *root node* for the range (a term that may make more sense to you).

Example

Use The Evaluator's predefined `rng` object to experiment with the `commonAncestorContainer` property. Reload the page. Now set the start point to the beginning of the contents of the `myEM` element and set the end point to the end of the surrounding `myP` element:

```
rng.setStartBefore(document.getElementById("myEM").firstChild)
rng.setEndAfter(document.getElementById("myP").lastChild)
```

Verify that the text range is set to encompass content from the `myEM` node (the word "all") and end of `myP` nodes (note that Safari 1.0 returns the wrong data here):

```
rng.toString()
```

Verify, too, that the two end point containers are different nodes:

```
rng.startContainer.tagName
rng.endContainer.tagName
```

Finally, see what node contains both of these two end points:

```
rng.commonAncestorContainer.id
```

The result is the myP element, which both the myP and myEM nodes have in common.

Related Items: endContainer, endOffset, startContainer, startOffset properties; all "set" and "select" methods of the Range object.

endContainer
startContainer

Value: Node object reference. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The endContainer and startContainer properties return a reference to the document tree node that contains the range's end point and start point, respectively. Be aware that the object model calculates the container, and the container may not be the reference you used to set the start and end points of a range. For example, if you use the selectNode() method to set the start and end points of a range to encompass a particular node, the containers of the end points are most likely the next outermost nodes. Thus, if you want to expand a range to the start of the node that contains the current range's start point, you can use the value returned by the startContainer property as a parameter to the setStartBefore() method:

```
rng.setStartBefore(rng.startContainer)
```

Example

Use The Evaluator's predefined rng object to experiment with the endContainer and startContainer properties. Reload the page and set the range to encompass the myEM element:

```
rng.selectNode(document.getElementById("myEM")
```

Inspect the containers for both the start and end points of the selection:

```
rng.startContainer.id
rng.endContainer.id
```

The range encompasses the entire myEM element, so the start and end points are outside of the element. Therefore, the container of both start and end points is the myP element that also surrounds the myEM element.

Related Items: commonAncestor, endOffset, startOffset properties; all "set" and "select" methods of the Range object.

endOffset
startOffset

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The endOffset and startOffset properties return an integer count of the number of characters or nodes for the location of the range's end point and start point, respectively. These counts are relative to the node that acts as the container node for the position of the boundary (see the Range.endContainer and Range.startContainer properties earlier in this chapter).

When a boundary is at the edge of a node (or perhaps "between" nodes is a better way to say it), the integer returned is the offset of nodes (zero-based) within the boundary's container. But when the boundary is in the middle of a text node, the integer returned is an index of the character position within the text node. The fact that each boundary has its own measuring system (nodes versus characters, relative to different containers) can get confusing if you're not careful, because conceivably the integer returned for an end point could be smaller than that for the start point. Consider the following nested elements:

```
<p>This paragraph has an <em>emphasized</em> segment.</p>
```

The next script statements set the start of the range to a character within the first text node and the end of the range to the end of the `em` node:

```
var rng = document.createRange();
rng.setStart(document.getElementById("myP").firstChild, 19);
rng.setEndAfter(document.getElementById("myEM"));
```

Using boldface to illustrate the body text that is now part of the range and the pipe (|) character to designate the boundaries as far as the nodes are concerned, here is the result of the preceding script execution:

```
<p id="myP">This paragraph has |an <em id="myEM">emphasized</em>| segment.</p>
```

Because the start of the range is in a text node (the first child of the `p` element), the range's `startOffset` value is 19, which is the zero-based character position of the "a" of the word "an." The end point, however, is at the end of the `em` element. The system recognizes this point as a node boundary, and thus counts the `endOffset` value within the context of the end container: the `p` element. The `endOffset` value is 2 (the `p` element's text node is node index 0; the `em` element is node index 1; and the position of the end point is at the start of the `p` element's final text node, at index 2).

For the `endOffset` and `startOffset` values to be of any practical use to a script, you must also use the `endContainer` and `startContainer` properties to read the context for the offset integer values.

Example

Use The Evaluator's predefined `rng` object to experiment with the `endOffset` and `startOffset` properties, following similar paths you just saw in the description. Reload the page and set the range to encompass the `myEM` element and then move the start point outward to a character within the `myP` element's text node:

```
rng.selectNode(document.getElementById("myEM"))
rng.setStart(document.getElementById("myP").firstChild, 7)
```

Inspect the node types of the containers for both the start and end points of the selection:

```
rng.startContainer.nodeType
rng.endContainer.nodeType
```

The `startContainer` node type is 3 (text node), while the `endContainer` node type is 1 (element). Now inspect the offsets for both the start and end points of the selection:

```
rng.startOffset
rng.endOffset
```

Related Items: `endContainer`, `startContainer` properties; all "set" and "select" methods of the `Range` object.

Methods

cloneContents()
cloneRange()

Returns: DocumentFragment node reference; `Range` object reference.
Compatibility: WinIE-, MacIE-, NN7+, Moz1+, Safari1+

The `cloneContents()` method (available in NN7+) takes a snapshot copy of the contents of a `Range` object and returns a reference to that copy. The copy is stored in the browser's memory, but is not part of the document tree. The `cloneRange()` method (available in NN6+) performs the same action on an entire range and stores the range copy in the browser's memory. A range's contents can consist of portions of multiple nodes and may not be surrounded by an element node; that's why its data is of the type `DocumentFragment` (one of the W3C DOM's node types). Because a `DocumentFragment` node is a valid node, it can be used with other document tree methods where nodes are required as parameters. Therefore, you can clone a text range to insert a copy elsewhere in the document.

In contrast, the `cloneRange()` method deals with range objects. While you are always free to work with the contents of a range object, the `cloneRange()` method returns a reference to a range object, which acts as a kind of wrapper to the contents (just as it does when the range is holding content in the main document). You can use the `cloneRange()` method to obtain a copy of one range to compare the end points of another range (via the `Range.compareBoundaryPoints()` method).

Example

Use The Evaluator's predefined `rng` object in NN7+/Moz/Safari to see the `cloneContents()` method in action. Begin by reloading the page and setting the range to the `myP` paragraph element:

```
rng.selectNode(document.getElementById("myP")
```

Next, clone the original range and preserve the copy in variable `b`:

```
b = rng.cloneContents()
```

Move the original range so that it is an insertion point at the end of the body by first expanding it to encompass the entire body and then collapse it to the end:

```
rng.selectNode(document.body)
rng.collapse(false)
```

Now, insert the copy at the very end of the body:

```
rng.insertNode(b)
```

If you scroll to the bottom of the page, you see a copy of the text in NN7/Moz. Safari 1.0 appears to miscalculate the range's boundary points after `collapse()`, causing a DOM hierarchy error when invoking `insertNode()`. But you can also use `appendChild()` or `insertBefore()` on any element node to put the cloned range into the document tree.

See the description of the `compareBoundaryPoints()` method later in this chapter to see an example of the `cloneRange()` method.

Related Items: `compareBoundaryPoints()`, `extractContents()` methods.

collapse([*startBoolean*])

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

Use the `collapse()` method to shrink a range from its current size down to a single insertion point between characters. Collapsing a range becomes more important than you may think at first, especially in a function that is traversing the body or large chunk of text. For example, in a typical looping word-counting script, you create a text range that encompasses the body fully. To begin counting words, you can first collapse the range to the insertion point at the very beginning of the range. Next, use the `expand()` method to set the range to the first word of text (and increment the counter if the `expand()` method returns `true`). At that point, the text range extends around the first word. You want the range to collapse at the end of the current range so that the search for the next word starts after the current one. Use `collapse()` once more, but this time with a twist of parameters.

The optional parameter of the `collapse()` method is a Boolean value that directs the range to collapse itself either at the start or end of the current range. The default behavior is the equivalent of a value of `true`, which means that unless otherwise directed, a `collapse()` method shifts the text range to the point in front of the current range. This method works great at the start of a word-counting script, because you want the text range to collapse to the start of the text. But for subsequent movements through the range, you want to collapse the range so that it is after the current range. Thus, you include a `false` parameter to the `collapse()` method.

Example

See Listings 35-11 and 15-14 to see the `collapse()` method at work (albeit with the IE `TextRange` object).

Related Items: `Range.setEnd()`, `Range.setStart()` methods.

compareBoundaryPoints(*typeInteger, sourceRangeRef*)

Returns: Integer (`-1`, `0`, or `1`).
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

Generating multiple range objects and assigning them to different variables is not a problem. You can then use the `compareBoundaryPoints()` method to compare the relative positions of start and end points of both ranges. One range is the object you use to invoke the `compareBoundaryPoints()` method, and the other range is the second parameter of the method. The order in which you reference the two ranges influences the results, based on the value assigned to the first parameter.

Values for the first parameter can be one of four constant values that are properties of the static `Range` object: `Range.START_TO_START`, `Range.START_TO_END`, `Range.END_TO_START`, and `Range.END_TO_END`. What these values specify is which point of the current range is compared with which point of the range passed as the second parameter. For example, consider the following body text that has two text ranges defined within it:

```
It was the best of times.
```

The first text range (assigned in our discussion here to variable `rng1`) is shown in boldface, while the second text range (`rng2`) is shown in bold-italic. In other words, `rng2` is nested inside `rng1`. We can compare the position of the start of `rng1` against the position of the start of `rng2` by using the `Range.START_TO_START` value as the first parameter of the `compareBoundaryPoints()` method:

```
var result = rng1.compareBoundaryPoints(Range.START_TO_START, rng2);
```

The value returned from the `compareBoundaryPoints()` method is an integer of one of three values. If the positions of both points under test are the same, then the value returned is `0`. If the start point of the (so-called source) range is before the range on which you invoke the method, the value returned is `-1`; in the opposite positions in the code, the return value is `1`. Therefore, from the example above, because the start of `rng1` is before the start of `rng2`, the method returns `-1`. If you change the statement to invoke the method on `rng2`, as in

```
var result = rng2.compareBoundaryPoints(Range.START_TO_START, rng1);
```

the result is `1`.

In practice, this method is helpful in knowing if two ranges are the same, if one of the boundary points of both ranges is the same, or if one range starts where the other ends.

Example

The page rendered by Listing 35-4 lets you experiment with text range comparisons in NN6+/Moz. The bottom paragraph contains a `span` element that has a `Range` object assigned to its nested text node after the page loads (in the `init()` function). That fixed range becomes a solid reference point for you to use while you select text in the paragraph.

After you make a selection, all four versions of the `compareBoundaryPoints()` method run to compare the start and end points of the fixed range against your selection. One column of the results table shows the raw value returned by the `compareBoundaryPoints()` method, while the third column puts the results into plain language.

To see how this page works, begin by selecting the first word of the fixed text range (carefully drag the selection from the first red character). You can see that the starting positions of both ranges are the same, because the returned value is `0`. Because all of the invocations of the `compareBoundaryPoints()` method are on the fixed text range, all comparisons are from the point of view of that range. Thus, the first row of the table for the `START_TO_END` parameter indicates that the start point of the fixed range comes before the end point of the selection, yielding a return value of `-1`.

Other selections to make include:

- ✦ Text that starts before the fixed range and ends inside the range
- ✦ Text that starts inside the fixed range and ends beyond the range
- ✦ Text that starts and ends precisely at the fixed range boundaries
- ✦ Text that starts and ends before the fixed range
- ✦ Text that starts after the fixed range

Study the returned values and the plain language results and see how they align with the selection you made.

Listing 35-4: **Lab for NN6+/Moz compareBoundaryPoints() Method**

```
<html>
   <head>
      <title>TextRange.compareBoundaryPoints() Method</title>
      <style type="text/css">
      td {text-align:center}
      .propName {font-family:Courier, monospace}
      #fixedRangeElem {color:red; font-weight:bold}
```

```
      </style>
      <script type="text/javascript">
      var fixedRange;

      function setAndShowRangeData() {
         try {
            var selectedRange = window.getSelection();
            selectedRange = selectedRange.getRangeAt(0);
            var result1 = fixedRange.compareBoundaryPoints(Range.START_TO_END,
               selectedRange);
            var result2 = fixedRange.compareBoundaryPoints(
               Range.START_TO_START, selectedRange);
            var result3 = fixedRange.compareBoundaryPoints(Range.END_TO_START,
               selectedRange);
            var result4 = fixedRange.compareBoundaryPoints(Range.END_TO_END,
               selectedRange);

            document.getElementById("B1").innerHTML = result1;
            document.getElementById("compare1").innerHTML =
               getDescription(result1);
            document.getElementById("B2").innerHTML = result2;
            document.getElementById("compare2").innerHTML =
               getDescription(result2);
            document.getElementById("B3").innerHTML = result3;
            document.getElementById("compare3").innerHTML =
               getDescription(result3);
            document.getElementById("B4").innerHTML = result4;
            document.getElementById("compare4").innerHTML =
               getDescription(result4);
         }
         catch(err) {
            alert("Vital Range or Selection object services are not yet
implemented in
               this browser.");
         }
      }

      function getDescription(comparisonValue) {
         switch (comparisonValue) {
         case -1 :
            return "comes before";
            break;
         case 0 :
            return "is the same as";
            break;
         case 1 :
            return "comes after";
            break;
         default :
            return "vs.";
         }
      }

      function init() {
         fixedRange = document.createRange();
         fixedRange.selectNodeContents(document.getElementById(
            "fixedRangeElem").firstChild);
         fixedRange.setEnd(fixedRange.endContainer,
```

Continued

Listing 35-4 *(continued)*

```
            fixedRange.endContainer.nodeValue.length);
      }
      </script>
   </head>
   <body onload="init()">
      <h1>TextRange.compareBoundaryPoints() Method</h1>
      <hr />
      <p>Select text in the paragraph in various places relative to the fixed
         text range (shown in red). See the relations between the fixed and
         selected ranges with respect to their start and end points.</p>
      <table id="results" border="1" cellspacing="2" cellpadding="2">
         <tr>
            <th>Property</th>
            <th>Returned Value</th>
            <th>Fixed Range vs. Selection</th>
         </tr>
         <tr>
            <td class="propName">StartToEnd</td>
            <td class="count" id="B1"> </td>
            <td class="count" id="C1">Start of Fixed <span
            id="compare1">vs.</span> End of Selection</td>
         </tr>
         <tr>
            <td class="propName">StartToStart</td>
            <td class="count" id="B2"> </td>
            <td class="count" id="C2">Start of Fixed <span
            id="compare2">vs.</span> Start of Selection</td>
         </tr>
         <tr>
            <td class="propName">EndToStart</td>
            <td class="count" id="B3"> </td>
            <td class="count" id="C3">End of Fixed <span
            id="compare3">vs.</span> Start of Selection</td>
         </tr>
         <tr>
            <td class="propName">EndToEnd</td>
            <td class="count" id="B4"> </td>
            <td class="count" id="C4">End of Fixed <span
            id="compare4">vs.</span> End of Selection</td>
         </tr>
      </table>
      <hr />
      <p onmouseup="setAndShowRangeData()">Lorem ipsum dolor sit,
         <span id="fixedRangeElem">consectetaur adipisicing elit</span>, sed
         do eiusmod tempor incididunt ut labore et dolore aliqua. Ut enim
         adminim veniam, quis nostrud exercitation ullamco laboris nisi ut
         aliquip ex ea commodo consequat.</p>
   </body>
</html>
```

compareNode(*nodeReference*)

Returns: Integer (0, 1, 2, or 3).
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The compareNode() method returns an integer code that indicates the relative position of the specified node with respect to the range. The node is passed as the only parameter to the method, and the value returned indicates the relative location of the node. The following four constants may be returned from the compareNode() method, and correspond to integer values in the range 0–3: Range.NODE_BEFORE, Range.NODE_AFTER, Range.NODE_BEFORE_AND_AFTER, Range.NODE_INSIDE. The first two values explain themselves, but the third value (Range.NODE_BEFORE_AND_AFTER) indicates that the node begins before the range and ends after the range. The final value (Range.NODE_INSIDE), on the other hand, indicates that the node is contained in its entirety by the range.

Related Items: comparePoint() method.

comparePoint(*nodeReference, offset*)

Returns: Integer (-1, 0, or 1).
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The comparePoint() method returns an integer code that indicates the relative position of the specified node at a certain offset with respect to the range. The node (as an object reference) and its offset (an integer count of an element's nodes or a text node's characters) are passed as parameters to the method, and the value returned indicates the relative location of the node. This location is specified with respect to the point (node and offset), not the range. The location of the node is indicated by the integer values –1, 0, and 1, where –1 indicates that the point comes before the start of the range, 0 indicates that the point is located within the range, and 1 reveals that the point comes after the end of the range.

Related Items: compareNode() method.

createContextualFragment("*text*")

Returns: W3C DOM document fragment node.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The createContextualFragment() method provides a way, within the context of the W3C DOM Level 2 node hierarchy to create a string of HTML text (with or without HTML tags, as needed) for insertion or appendage to existing node trees. During the development of the NN6 browser, this method filled a gap that was eventually filled by Netscape's adoption of the Microsoft proprietary innerHTML property. The method obviates the need for tediously assembling a complex HTML element via a long series of document.createElement() and document.createTextNode() methods for each segment, plus the assembly of the node tree prior to inserting it into the actual visible document. The existence of the innerHTML property of all element objects, however, reduces the need for the createContextualFragment() method, while allowing more code to be shared across browser brands.

The parameter to the createContextualFragment() method is any text, including HTML tags. To invoke the method, however, you need to have an existing range object available. Therefore, the sequence used to generate a document fragment node is

```
var rng = document.createRange();
rng.selectNode(document.body); // any node will do
var fragment = rng.createContextualFragment("<h1>Howdy</h1>");
```

As a document fragment, the node is not part of the document node tree until you use the fragment as a parameter to one of the tree modification methods, such as Node.insertBefore() or Node.appendChild().

Example

Use The Evaluator's predefined `rng` object to replace an existing document tree node with the fragment. Begin by creating the fragment from The Evaluator's built-in range:

```
b = rng.createContextualFragment("<span style='font-size:22pt'>a bunch of
</span>")
```

This fragment consists of a `span` element node with a text node nested inside. At this point, you can inspect the properties of the document fragment by entering `b` into the bottom text box.

To replace the `myEM` element on the page with this new fragment, use the `replaceChild()` method on the enclosing `myP` element:

```
document.getElementById("myP").replaceChild(b, document.getElementById("myEM"))
```

The fragment now becomes a legitimate child node of the `myP` element and can be referenced like any node in the document tree. For example, if you enter the following statement into the top text box of The Evaluator, you can retrieve a copy of the text node inside the new `span` element:

```
document.getElementById("myP").childNodes[1].firstChild.nodeValue
```

Related Items: Node object (Chapter 15).

deleteContents()

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `deleteContents()` method removes all contents of the current range from the document tree. After deletion, the range collapses to an insertion point where any surrounding content (if any) cinches up to its neighbors.

Some alignment of a range's boundaries forces the browser to make decisions about how element boundaries inside the range are treated after the deletion. An easy deletion is one for which the range boundaries are symmetrical. For example, consider the following HTML with a range highlighted in bold:

```
<p>One paragraph with an <em>emphasis</em> inside.</p>
```

After you delete the contents of this range, the text node inside the `em` element disappears, but the `em` element remains in the document tree (with no child nodes). Similarly, if the range is defined as being the entire second child node of the `p` element, as follows

```
<p>One paragraph with an <em>emphasis</em> inside.</p>
```

then deleting the range contents removes both the text node and the `em` element node, leaving the `p` element with a single, unbroken text node as a child (although in the previous case, an extra space would be between the words "an" and "inside" because the `em` element does not encompass a space on either side).

When range boundaries are not symmetrical, the browser does its best to maintain document tree integrity after the deletion. Consider the following HTML and range:

```
<p>One paragraph with an <em>emphasis</em> inside.</p>
```

After deleting this range's contents, the document tree for this segment looks like the following:

```
<p>One paragraph <em>phasis</em> inside.</p>
```

The range collapses to an insertion point just before the `<em>` tag. But notice that the `em` element persisted to take care of the text still under its control. Many other combinations of range

boundaries and nodes are possible, so be sure that you check out the results of a contents deletion for asymmetrical boundaries before applying the deletion.

Example

Use The Evaluator's predefined `rng` object to experiment with deleting contents of both a text node and a complete element node. Begin by adjusting the text range to the text node inside the `myEM` element (enter the third statement, which wraps below, as one continuous expression):

```
rng.setStart(document.getElementById("myEM").firstChild, 0)
rng.setEnd(document.getElementById("myEM").lastChild,
    document.getElementById("myEM").lastChild.length)
```

Verify the makeup of the range by entering `a` into the bottom text box and inspect its properties. Both containers are text nodes (they happen to be the same text node), and offsets are measured by character positions.

Now, delete the contents of the range:

```
rng.deleteContents()
```

The italicized word "all" is gone from the tree, but the `myEM` element is still there. To prove it, put some new text inside the element:

```
document.getElementById("myEM").innerHTML = "a band of "
```

The italic style of the `em` element applies to the text, as it should.

Next, adjust the range boundaries to include the `myEM` element tags, as well:

```
rng.selectNode(document.getElementById("myEM"))
```

Inspect the `Range` object's properties again by entering `rng` into the bottom text box. The container nodes are the `p` element that surrounds the `em` element; the offset values are measured in nodes. Delete the range's contents:

```
rng.deleteContents()
```

Not only is the italicized text gone, but the `myEM` element is gone, too. The `myP` element now has two child nodes: the two text nodes that used to flank the `em` element. The following entries into the top text box of The Evaluator verify this fact:

```
document.getElementById("myP").childNodes.length
document.getElementById("myP").childNodes[0].nodeValue
```

To combine the two sibling text nodes into one, invoke the `normalize()` method of the container:

```
document.getElementById("myP").normalize()
```

Check the number of child nodes again to verify the results.

Related Items: `Range.extractContents()` method.

`detach()`

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `detach()` method instructs the browser to release the current range object from the object model. In the process, the range object is nulled out to the extent that an attempt to access the object results in a script error. You can still assign a new range to the same variable if you like. You are not required to detach a range when you're finished with it, and the browser

resources employed by a range are not that large. But it is good practice to "clean up after yourself," especially when a script repetitively creates and manages a series of new ranges.

Related Items: `document.createRange()` `method.extractContents()`

`extractContents()`

Returns: `DocumentFragment` node reference.
Compatibility: WinIE-, MacIE-, NN7+, Moz1+, Safari1+

The `extractContents()` method deletes the contents of the range and returns a reference to the document fragment node that is held in the browser memory, but which is no longer part of the document tree. A range's contents can consist of portions of multiple nodes and may not be surrounded by an element node; that's why its data is of the type `DocumentFragment` (one of the W3C DOM's node types). Because a `DocumentFragment` node is a valid node, it can be used with other document tree methods where nodes are required as parameters. Therefore, you can extract a text range from one part of a document to insert elsewhere in the document.

Example

Use The Evaluator's predefined `rng` object in NN7+/Moz/Safari to see how the `extractContents()` method works. Begin by setting the built-in range object to contain the text of the `myP` paragraph element.

```
rng.selectNode(document.getElementById("myP"))
```

Next, extract the original range's content and preserve the copy in variable `b`:

```
b = a.extractContents()
```

Now, insert the extracted fragment at the very end of the body:

```
document.body.appendChild(b)
```

If you scroll to the bottom of the page, you see a copy of the text.

Related Items: `cloneContents()`, `deleteContents()` methods.

`insertNode(`*`nodeReference`*`)`

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN7+, Moz1+, Safari1+

The `insertNode()` method inserts a node at the start point of the current range. The node being inserted may be an element or text fragment node, and its source can be any valid node creation mechanism, such the `document.createTextNode()` method or any node extraction method.

Example

Listing 35-5 demonstrates the `insertNode()` method plus some additional items from the `selection` object. The example even includes a rudimentary undo buffer for scripted changes to a text range. In the page generated by this listing, users can select any text in a paragraph and have the script automatically convert the text to all uppercase characters. The task of replacing a selection with other text requires several steps, starting with the selection, which is retrieved via the `window.getSelection()` method. After making sure the selection contains some text (that is, the selection isn't collapsed), the selection is preserved as a range object so that the starting text can be stored in a global variable (as a property of the `undoBuffer` global variable object). After that, the selection is deleted from the document tree, leaving the selection as a

collapsed insertion point. A copy of that selection in the form of a range object is preserved in the `undoBuffer` object so that the undo script knows where to reinsert the original text. A new text node is created with an uppercase version of the original text, and, finally, the `insertNode()` method is invoked to stick the converted text into the collapsed range.

Undoing this operation works in reverse. Original locations and strings are copied from the `undoBuffer` object. After creating the range with the old start and end points (which represent a collapsed insertion point), the resurrected text (converted to a text node) is inserted into the collapsed range. For good housekeeping, the `undoBuffer` object is restored to its unused form.

Listing 35-5: **Inserting a Node into a Range**

```
<html>
   <head>
      <title>NN Selection Object Replacement</title>
      <script type="text/javascript">
      var undoBuffer = {rng:{}, txt:""};

      function convertSelection() {
         var sel, grossRng, netRng, newText;
         try {
            sel = window.getSelection();
            if (!sel.isCollapsed) {
               grossRng = sel.getRangeAt(0);
               undoBuffer.txt = grossRng.toString();
               undoBuffer.rng.startContainer = grossRng.startContainer;
               undoBuffer.rng.startOffset = grossRng.startOffset;
               undoBuffer.rng.endContainer = grossRng.endContainer;
               undoBuffer.rng.endOffset = grossRng.endOffset;
               sel.deleteFromDocument();
               netRng = sel.getRangeAt(0);
               newText = document.createTextNode(undoBuffer.txt.toUpperCase());
               netRng.insertNode(newText);
               netRng.commonAncestorContainer.parentNode.normalize();
            }
         }
         catch(err) {
            alert("Vital Range or Selection object services are not yet
implemented " +
               "in this browser.");
         }
      }

      function undoConversion() {
         var rng, oldText;
         if (undoBuffer.rng) {
            rng = document.createRange();
            rng.setStart(undoBuffer.rng.startContainer,
undoBuffer.rng.startOffset);
            rng.setEnd(undoBuffer.rng.endContainer, undoBuffer.rng.endOffset);
            rng.extractContents();
            oldText = document.createTextNode(undoBuffer.txt);
            rng.insertNode(oldText);
            undoBuffer.rng = {};
```

Continued

Listing 35-5 *(continued)*

```
            undoBuffer.txt = "";
         }
      }
      </script>
   </head>
   <body>
      <h1 id="H1_1">NN6+/Moz Selection Object Replacement</h1>
      <hr />
      <p id="P_1" onmouseup="convertSelection()">This paragraph contains text
         that you can select. Selections are deleted and replaced by all
         uppercase versions of the selected text.</p>
      <button onclick="undoConversion()">Undo Last</button> <button
      onclick="location.reload(true)">Start Over</button>
   </body>
</html>
```

intersectsNode(*nodeReference*)

Returns: Boolean.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `intersectsNode()` method returns a Boolean value that indicates whether (`true`) or not (`false`) any part of the range overlaps the node whose reference is passed as the method's parameter.

Related Items: `compareNode()` method.

isPointInRange(*nodeReference, offset*)

Returns: Boolean.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `isPointInRange()` method returns a Boolean value that indicates whether (`true`) or not (`false`) the specified node (a node object reference) and offset (integer count of an element's nodes or a text node's characters) are located entirely within the range.

selectNode(*nodeReference*)
selectNodeContents(*nodeReference*)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

The `selectNode()` and `selectNodeContents()` methods are convenience methods for setting both end points of a range to surround a node or a node's contents. The kind of node you supply as the parameter to either method (text node or element node) has a bearing on the range's container node types and units of measure for each (see the container- and offset-related properties of the `Range` object earlier in this chapter).

Starting with the `selectNode()` method, if you specify an element node as the one to select, the start and end container node of the new range is the next outermost element node; offset values count nodes within that parent element. If you specify a text node to be selected, the

container node for both ends is the parent element of that text node; offset values count the nodes within that parent.

With the `selectNodeContents()` method, the start and end container nodes are the very same element specified as the parameter; offset values count the nodes within that element. If you specify a text node's contents to be selected, the text node is the start and end parent, but the range is collapsed at the beginning of the text.

By and large, you specify element nodes as the parameter to either method, allowing you to set the range to either encompass the element (via `selectNode()`) or just the contents of the element (via `selectNodeContents()`).

Example

Use The Evaluator's predefined `rng` object to see the behavior of both the `selectNode()` and `selectNodeContents()` methods work. Begin by setting the range object's boundaries to include the `myP` element node:

```
rng.selectNode(document.getElementById("myP"))
```

Enter `a` into the bottom text box to view the properties of the range. Notice that because the range has selected the entire paragraph node, the container of the range's start and end points is the `body` element of the page (the parent element of the `myP` element).

Now change the range so that it encompasses only the contents of the `myP` element:

```
rng.selectNodeContents(document.getElementById("myP"))
```

Click the List Properties button to view the current properties of the range. The container of the range's boundary points is the `p` element that holds the element's contents.

Related Items: `setEnd()`, `setEndAfter()`, `setEndBefore()`, `setStart()`, `setStartAfter()`, `setStartBefore()` methods.

setEnd(*nodeReference, offset*)
setStart(*nodeReference, offset*)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

You can adjust the start and end points of a text range independently via the `setStart()` and `setEnd()` methods. While not as convenient as the `selectNode()` or `selectNodeContents()` methods, these two methods give you the ultimate in granularity over precise positioning of a range boundary.

The first parameter to both methods is a reference to a node. This reference can be an element or text node, but your choice here also influences the kind of measure applied to the integer offset value supplied as the second parameter. When the first parameter is an element node, the offset counts are in increments of child nodes inside the specified element node. But if the first parameter is a text node, the offset counts are in increments of characters within the text node.

When you adjust the start and end points of a range with these methods, you have no restrictions to the symmetry of your boundaries. One boundary can be defined relative to a text node, while the other relative to an element node — or vice versa.

To set the end point of a range to the last node or character within a text node (depending on the unit of measure for the *`offset`* parameter), you can use the `length` property of the units being measured. For example, to set the end point to the end of the last node within an element

(perhaps there are multiple nested elements and text nodes within that outer element), you can use the first parameter reference to help you get there:

```
rng.setEnd(document.getElementById("myP"),
   document.getElementById("myP").childNodes.length);
```

These kinds of expressions get lengthy, so you may want to make a shortcut to the reference to simplify the values of the parameters, as shown in this version that sets the end point to after the last character of the last text node of a `p` element:

```
var nodeRef = document.getElementById("myP").lastChild;
rng.setEnd(nodeRef, nodeRef.nodeValue.length);
```

In both previous examples with the `length` properties, the values of those properties are always pointing to the node or character position after the final object because the index values for those objects' counts are zero-based. Also bear in mind that if you want to set a range end point at the edge of a node, you have four other methods to choose from (`setEndAfter()`, `setEndBefore()`, `setStartAfter()`, and `setStartBefore()`). The `setEnd()` and `setStart()` methods are best used when an end point needs to be set at a location other than at a node boundary.

Example

Use The Evaluator's predefined `rng` object to experiment with both the `setStart()` and `setEnd()` methods. For the first range, set the start and end points to encompass the second node (the `myEM` element) inside the `myP` element:

```
rng.setStart(document.getElementById("myP"), 1)
rng.setEnd(document.getElementById("myP"), 2)
```

The text encompassed by the range consists of the word "all" plus the trailing space that is contained by the `myEM` element. Prove this by entering the following statement into the top text box (Safari 1.0 returns an incorrect value):

```
rng.toString()
```

If you then click the Results box to the right of the word "all," you see that the results contain the trailing space. Yet, if you examine the properties of the range (enter `a` into the bottom text box), you see that the range is defined as actually starting before the `myEM` element and ending after it.

Next, adjust the start point of the range to a character position inside the first text node of the `myP` element:

```
rng.setStart(document.getElementById("myP").firstChild, 11)
```

Click the List Properties button to see that the `startContainer` property of the range is the text node, and that the `startOffset` measures the character position. All end boundary properties, however, have not changed. Enter `rng.toString()` in the top box again to see that the range now encompasses text from two of the nodes inside the `myP` element.

You can continue to experiment by setting the start and end points to other element and text nodes on the page. After each adjustment, verify the properties of the `a` range object and the text it encompasses (via `rng.toString()`).

Related Items: `selectNode()`, `selectNodeContents()`, `setEndAfter()`, `setEndBefore()`, `setStartAfter()`, `setStartBefore()` methods.

setEndAfter(*nodeReference*)
setEndBefore(*nodeReference*)
setStartAfter(*nodeReference*)
setStartBefore(*nodeReference*)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

You can adjust the start and end points of a text range relative to existing node boundaries via your choice of these four methods. The "before" and "after" designations are used to specify which side of the existing node boundary the range should have for its boundary. For example, using `setStartBefore()` and `setEndAfter()` with the same element node as a parameter is the equivalent of the `selectNode()` method on that element. You may also specify a text node as the parameter to any of these methods. But because these methods work with node boundaries, the offset values are always defined in terms of node counts, rather than character counts. At the same time, however, the boundaries do not need to be symmetrical, so that one boundary can be inside one node and the other boundary inside another node.

Example

Use The Evaluator's predefined `rng` object to experiment with all four methods. For the first range, set the start and end points to encompass the `myEM` element inside the `myP` element:

```
rng.setStartBefore(document.getElementById("myEM"))
rng.setEndAfter(document.getElementById("myEM"))
```

The text encompassed by the range consists of the word "all" plus the trailing space that is contained by the `myEM` element. Prove this by entering the following statement into the top text box (Safari 1.0 returns an incorrect value):

```
rng.toString()
```

Next, adjust the start point of the range to the beginning of the first text node of the `myP` element:

```
rng.setStartBefore(document.getElementById("myP").firstChild)
```

Enter `rng` into the bottom text box to see that the `startContainer` property of the range is the `p` element node, while the `endContainer` property points to the `em` element.

You can continue to experiment by setting the start and end points to before and after other element and text nodes on the page. After each adjustment, verify the properties of the `a` range object and the text it encompasses (via `rng.toString()`).

Related Items: `selectNode()`, `selectNodeContents()`, `setEnd ()`, `setStart()` methods.

surroundContents(*nodeReference*)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN7+, Moz1+, Safari1+

The `surroundContents()` method surrounds the current range with a new parent element. Pass the new parent element as a parameter to the method. No document tree nodes or elements are removed or replaced in the process, but the current range becomes a child node of the new node; if the range coincides with an existing node, then the relationship between that node and its original parent becomes that of grandchild and grandparent. An application of this method may be to surround user-selected text with a `span` element whose class renders the content with a special font style or other display characteristic based on a stylesheet selector for that class name.

When the element node being applied as the new parent has child nodes itself, those nodes are discarded before the element is applied to its new location. Therefore, for the most predictable results, using content-free element nodes as the parameter to the `surroundContents()` method is best.

Example

Listing 35-6 demonstrates how the `surroundContents()` method wraps a range inside a new element. As the page loads, a global variable (`newSpan`) stores a `span` element that is used as a prototype for elements to be used as new surrounding parent nodes. When you select text in either of the two paragraphs, the selection is converted to a range. The `surroundContents()` method then wraps the range with the `newSpan` element. Because that `span` element has a class name of `hilite`, the element and its contents pick up the stylesheet properties as defined for that class selector.

NN7 and Mozilla (at least through version 1.4) exhibit a bug in this method when used with a range derived from a `selection` object or when the range is not aligned with element boundaries. Until this bug is fixed, Listing 35-6 will follow its exception execution track.

Listing 35-6: **Using the Range.surroundContents() Method**

```
<html>
   <head>
      <title>Range.surroundContents() Method</title>
      <style type="text/css">
      .hilite {background-color:yellow; color:red; font-weight:bold}
      </style>
      <script type="text/javascript">
      var newSpan = document.createElement("span");
      newSpan.className = "hilite";

      function highlightSelection() {
         var sel, rng;
         try {
            sel = window.getSelection();
            if (!sel.isCollapsed) {
               rng = sel.getRangeAt(0);
               rng.surroundContents(newSpan.cloneNode(false));
            }
         }
         catch(err) {
            alert("Vital Range or Selection object services are not yet
implemented in
               this browser.");
         }
      }
      </script>
   </head>
   <body>
      <h1>Range.surroundContents() Method</h1>
      <hr />
      <p onmouseup="highlightSelection()">These paragraphs contain text that
         you can select. Selections are surrounded by span elements that share
         a stylesheet class selector for special font and display
         characteristics.</p>
```

```
      <p onmouseup="highlightSelection()">Lorem ipsum dolor sit amet,
         consectetaur adipisicing elit, sed do eiusmod tempor incididunt ut
         labore et dolore magna aliqua. Ut enim adminim veniam, quis nostrud
         exercitation ullamco laboris nisi ut aliquip ex ea commodo
         consequat.</p>
   </body>
</html>
```

Related Items: `Range.insertNode()` method.

`toString()`

Returns: String.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

Use the `toString()` method to retrieve a copy of the body text that is contained by the current text range. The text returned from this method is ignorant of any HTML tags or node boundaries that exist in the document tree. You also use this method to get the text of a user selection, after it has been converted to a text range. Safari 1.0 commonly returns text beyond the range, even though the range's property values reflect correct characteristics.

Example

Use The Evaluator's predefined `rng` object to see the results of the `toString()` method. Enter the following sequence of statements into the top text box:

```
rng.selectNode(document.getElementById("myP"))
rng.toString()
```

If you type only `rng` into the top text box, you see the text contents of the range, but don't be fooled. Internal workings of The Evaluator attempt to evaluate any expression entered into that text field. Assigning a range object to a text box forces an internal application of the `toString()` method (just as the `Date` object does when you create a new object instance in The Evaluator).

Related Items: `Selection.getRangeAt()`, `Range.extractContents()` methods.

selection Object

Compatibility: WinIE4+, MacIE-, NN6+, Moz1+, Safari-

Properties	*Methods*	*Event Handlers*
anchorNode	addRange()	
anchorOffset	clear()	
focusNode	collapse()	
focusOffset	collapseToEnd()	
isCollapsed	collapseToStart()	
rangeCount	containsNode()	
type	createRange()	

Continued

selection

Properties	*Methods*	*Event Handlers*
typeDetail	deleteFromDocument()	
	empty()	
	extend()	
	getRangeAt()	
	removeAllRanges()	
	removeRange()	
	selectAllChildren()	
	toString()	

Syntax

Accessing `selection` object properties or methods:

```
(IE4+)     [window.]document.selection.property | method()
(NN6+/Moz) window.selection.property | method()
```

About this object

The `selection` object provides scripted access to any body text or text in a form text control that is selected either by the user or by script. A `selection` object of one character or more is always highlighted on the page, and only one `selection` object can be active at any given instant.

Take advantage of the `selection` object when your page invites a user to select text for some operation that utilizes the selected text. The best event to use for working with a selection is the `onmouseup` event handler. This event fires on every release of the mouse, and your script can investigate the `document.selection` object to see if any text has been selected (using the selection's `type` property).

If you intend to perform some action on a selection, you may not be able to trigger that action by way of a button or link. In some browser versions and operating systems, clicking one of these elements automatically deselects the body selection.

One important difference between the IE and NN selections is that the NN6+/Moz `selection` object works only on body text, and not on selections inside text-oriented form controls. An NN6/Moz `selection` object has relationships with the document's node tree in that the object defines itself by the nodes (and offsets within those nodes) that encase the start and end points of a selection. When a user drags a selection, the node in which the selection starts is called the *anchor* node; the node holding the text at the point of the selection release is called the *focus* node; for double- or triple-clicked selections, the direction between anchor and focus nodes is in the direction of the language script (for example, left-to-right in Latin-based script families). In many ways, an NN6+/Moz `selection` object behaves just as the W3C DOM `Range` object, complete with methods to collapse and extend the selection. Unlike a range, however, the text encompassed by a `selection` object is highlighted on the page. If your scripts need to work with the nodes inside a selection, the `getRangeAt()` method of the `selection` object returns a range object whose boundary points coincide with the selection's boundary points.

Properties

anchorNode
focusNode

Value: Node reference. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

These two properties return a reference to the node where the user started (`anchorNode`) and ended (`focusNode`) the selection. If the selection is modified via the `addRange()` method, these properties point to the node boundaries of the most recently added range.

Related Items: `anchorOffset` and `focusOffset` properties.

anchorOffset
focusOffset

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

These two properties return an integer count of the number of characters or nodes from the beginning of the selection to the anchor node (`anchorOffset`) and focus node (`focusOffset`). The count represents characters for text nodes, and nodes for element nodes. If the selection is modified via the `addRange()` method, these properties point to the node offsets of the most recently added range.

Related Items: `anchorNode` and `focusNode` properties.

isCollapsed

Value: Boolean. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `isCollapsed` property returns a Boolean value indicating whether or not the anchor and focus boundaries of the selection are the same. If they are the same (`true`), it means the selection has zero length between two characters (or before the first or after the last character of the document).

Related Items: `anchorNode` and `focusNode` properties.

rangeCount

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `rangeCount` property returns an integer count of the range objects contained within the selection. A manual user selection always results in a single range being selected, but the `addRange()` method can result in multiple ranges being contained by the selection.

Related Items: `getRangeAt()` method.

type

Value: String. Read-Only
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The `type` property returns `Text` whenever a selection exists on the page. Otherwise the property returns `None`. A script can use this information to determine if a selection is made on the page:

```
if (document.selection.type == "Text") {
   // process selection
   ...
}
```

Microsoft indicates that this property can sometimes return `Control`, but that terminology is associated with an edit mode outside the scope of this book.

Example

Listing 35-7 contains a page that demonstrates several features of the IE `selection` object. When you make a selection with the Deselect radio button selected, you see the value of the `selection.type` property (in the status bar) before and after the selection is deselected. After the selection goes away, the `type` property returns `None`.

Listing 35-7: Using the document.selection Object

```
<html>
   <head>
      <title>selection Object</title>
      <script type="text/javascript">
      function processSelection() {
         if (document.choices.process[0].checked) {
            status = "Selection is type: " + document.selection.type;
            setTimeout("emptySelection()", 2000);
         } else if (document.choices.process[1].checked) {
            var rng = document.selection.createRange();
            document.selection.clear();
         }
      }
      function emptySelection() {
         document.selection.empty();
         status = "Selection is type: " + document.selection.type;
      }
      </script>
   </head>
   <body>
      <h1>IE selection Object</h1>
      <hr />
      <form name="choices">
         <input type="radio" name="process" checked="checked" />De-select after
         two seconds<br />
         <input type="radio" name="process" />Delete selected text.
      </form>
```

```
      <p onmouseup="processSelection()">Lorem ipsum dolor sit amet,
         consectetaur adipisicing elit, sed do eiusmod tempor incididunt ut
         labore et dolore magna aliqua. Ut enim adminim veniam, quis nostrud
         exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
         Duis aute irure dolor in reprehenderit involuptate velit esse cillum
         dolore eu fugiat nulla pariatur.</p>
   </body>
</html>
```

Related Items: `TextRange.select()` method.

`typeDetail`

Value: See text. Read-Only
Compatibility: WinIE5.5+, MacIE-, NN-, Moz-, Safari-

The `typeDetail` property serves as a placeholder for applications that use the IE browser component, in which case the property can serve as a means of providing additional selection type information.

Related Items: `type` property.

Methods

`addRange(rangeRef)`

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `addRange()` method is used to highlight a selection on a page based upon a `Range` object. You can add multiple ranges to a selection by making repeated calls to the `addRange()` method.

Related Items: `removeRange()` method.

`clear()`

Returns: Nothing.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

Use the `clear()` method to delete the current selection from the document. To the user, the `clear()` method has the same effect as setting the `TextRange.text` property to an empty string. The difference is that you can use the `clear()` method without having to generate a text range for the selection. After you delete a selection, the `selection.type` property returns `None`.

Example

See Listing 35-7 earlier in this chapter to see the `selection.clear()` method at work.

Related Items: `selection.empty()` method.

collapse(*nodeRef, offset*)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `collapse()` method collapses the current selection to a location specified by the two parameters, which consist of a node reference and an offset. The `nodeRef` parameter is a text or element node in the document tree to which the collapsed selection is to be moved, while the `offset` parameter is an integer count of characters or nodes within the target node where the collapsed selection should be moved.

Related Items: `collapseToEnd()` and `collapseToStart()` methods.

collapseToEnd()
collapseToStart()

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

These methods collapse the current selection to a location at the start (`collapseToStart()`) or end (`collapseToEnd()`) of the selection. After the collapse, any previously highlighted selection returns to normal display and the selection contains only one range.

Related Items: `collapse()` method.

containsNode(*nodeRef, entirelyFlag*)

Returns: Boolean.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `containsNode()` method returns a Boolean value indicating whether or not the specified node is contained in the selection. The `nodeRef` parameter is the node you are checking for selection containment, while the `entirelyFlag` parameter specifies whether or not the node must be contained in its entirety (passing `null` for the `entirelyFlag` parameter is usually sufficient).

createRange()

Returns: `TextRange` object.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

To generate a text range for a user selection in IE, invoke the `createRange()` method of the `selection` object. I'm not sure why the method for the `selection` object is called `createRange()` while text ranges for other valid objects are created with a `createTextRange()` method. The result of both methods is a full-fledged `TextRange` object.

Example

See Listings 15-36 and 15-45 to see the `selection.createRange()` method turn user selections into text ranges.

Related Items: `TextRange` object.

deleteFromDocument()

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `deleteFromDocument()` method deletes the current selection from the document tree.

Related Items: `removeRange()` method.

empty()

Returns: Nothing.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The `empty()` method deselects the current IE selection. After deselection, the `selection.type` property returns `None`. The action of the `empty()` method is the same as the `UnSelect` command invoked via the `execCommand()` method for a document. If the selection was made from a `TextRange` object (via the `TextRange.select()` method), the `empty()` method affects only the visible selection and not the text range.

Example

See Listing 35-7 earlier in this chapter to view the `selection.empty()` method at work.

Related Items: `selection.clear()` method.

extend(*nodeRef, offset*)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `extend()` method extends the boundary of the selection to the specified node (`nodeRef`) and offset (`offset`) within that node. The start of the boundary (anchor node) remains unaffected by this method; only the end of the boundary (focus node) is altered.

getRangeAt(*rangeIndex*)

Returns: `Range` object.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `getRangeAt()` method obtains the range at the specified zero-based index (`rangeIndex`) within the selection. You can use the `rangeCount` property to determine how many ranges are contained within the selection.

Related Items: `rangeCount` property.

removeAllRanges()

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `removeAllRanges()` method empties the selection by removing all of the ranges from it. Upon calling this method, the selection collapses and the `rangeCount` property goes to zero; the document tree remains unaffected.

Related Items: `removeRange()` method.

removeRange(*rangeRef*)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `removeRange()` method removes the specified range (`rangeRef`) from the selection, but not from the document tree.

Related Items: `removeAllRanges()` method.

selectAllChildren(*elementNodeRef*)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `selectAllChildren()` method forces the selection to encompass the specified node (`elementNodeRef`) and all of its children. Calling this method on an element node results in the anchor and focus nodes being set to that node.

toString()

Returns: String.
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `toString()` method returns a string representation of the selection, which is the body content from the selection minus tags and attributes.

Text and TextNode Objects

Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

Properties	*Methods*	*Event Handlers*
attributes†	appendChild()†	
childNodes†	appendData()	
data	cloneNode()†	
firstChild†	deleteData()	
lastChild†	hasChildNodes()†	
length†	insertBefore()†	
localName†	insertData()	
namespaceURI†	normalize()†	
nextSibling†	removeChild()†	
nodeName†	replaceChild()†	
nodeType†	replaceData()	
nodeValue†	splitText()	
ownerDocument†	substringData()	
parentNode†		
prefix†		
previousSibling†		

†See Chapter 15.

Syntax

Accessing `Text` and `TextNode` object properties or methods:

```
(IE5+/W3C) [window.]document.getElementById("id").textNodeRef.property | method()
```

About this object

Discussing the `Text` object of the W3C DOM (as implemented in NN6+/Moz) in the same breath as the IE5+ `TextNode` object is a little tricky. Conceptually, they are the same kind of object in that they are the document tree objects — text nodes — that contain an HTML element's text (see Chapter 14 for details on the role of the text node in the `document` object hierarchy). Generating a new text node by script is achieved the same way in both object models: `document.createTextNode()`. What makes the discussion of the two objects tricky is that while the W3C DOM version comes from a strictly object-oriented specification (in which a text node is an instance of a `CharacterData` object, which, in turn is an instance of the generic `Node` object), the IE object model is not quite as complete. For example, while the W3C DOM `Text` object inherits all of the properties and methods of the `CharacterData` and `Node` definitions, the IE `TextNode` object exposes only those properties and method that Microsoft deems appropriate.

No discrepancy in terminology gets in the way as to what to call these objects because their object names never become part of the script. Instead script statements always refer to text nodes by other means, such as through a child node-related property of an element object or as a variable that receives the result of the `document.createTextNode()` method.

Although both objects share a number of properties and one method, the W3C DOM `Text` object contains a few methods that have "data" in their names. These properties and methods are inherited from the `CharacterData` object in the DOM specification. They are discussed as a group in the section about object methods in this chapter. In all cases, check the browser version support for each property and method described here.

Properties

data

Value: String. Read/Write
Compatibility: WinIE5+, MacIE5+, NN6+, Moz1+, Safari1+

The `data` property contains the string comprising the text node. Its value is identical to the `nodeValue` property of a text node. See the description of the `nodeValue` property in Chapter 15.

Example

In the example for the `nodeValue` property used in a text replacement script (in Chapter 1 of this book), you can substitute the `data` property for `nodeValue` to accomplish the same result.

Related Items: `nodeValue` property of all element objects (Chapter 15).

Methods

```
appendData("text")
deleteData(offset, count)
insertData(offset, "text")
replaceData(offset, count, "text")
substringData(offset, count)
```

Returns: See text.
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

These five methods of the W3C DOM `Text` object provide scripted manipulation of the text inside a text node. Methods that modify the node's data automatically change the values of both the `data` and `nodeValue` properties.

The purposes of these methods are obvious for the most part. Any method that requires an *`offset`* parameter uses this integer value to indicate where in the existing text node the deletion, insertion, or replacement starts. Offsets are zero-based, meaning that to indicate the action should take place starting with the first character, specify a zero for the parameter. A *`count`* parameter is another integer, but one that indicates how many characters are to be included. For example, consider a text node that contains the following data:

```
abcdefgh
```

This node could be a node of an element on the page or a node that has been created and assigned to a variable but not yet inserted into the page. To delete the first three characters of that text node, the statement is

```
textNodeReference.deleteData(0,3)
```

This leaves the text node content as

```
defgh
```

As for the `replaceData()` method, the length of the text being put in place of the original chunk of text need not match the *`count`* parameter. The *`count`* parameter, in concert with the *`offset`* parameter, defines what text is to be removed and replaced by the new text.

The `substringData()` method is similar to the JavaScript core language `String.substr()` method in that both require parameters indicating the offset within the string to start reading and for how many characters. You get the same result with the `substringData()` method of a text node as you do from a `nodeValue.substr()` method when both are invoked from a valid text node object.

Of all five methods discussed here, only `substringData()` returns a value: a string.

Example

The page created by Listing 35-8 is a working laboratory that you can use to experiment with the five data-related methods in NN6+/Moz/Safari. The text node that invokes the methods is a simple sentence in a `p` element. Each method has its own clickable button, followed by two or three text boxes into which you enter values for method parameters. Don't be put off by the length of the listing. Each method's operation is confined to its own function and is fairly simple.

Each of the data-related methods throws exceptions of different kinds. To help handle these errors gracefully, the method calls are wrapped inside a `try/catch` construction. All caught exceptions are routed to the `handleError()` function where details of the error are inspected and friendly alert messages are displayed to the user. See Chapter 39 on the CD-ROM for details on the `try/catch` approach to error handling in W3C DOM-capable browsers.

Listing 35-8: **Text object Data Method Laboratory**

```
<html>
   <head>
      <title>Data Methods of a W3C Text Object</title>
      <script type="text/javascript">
      function doAppend(form) {
         var node = document.getElementById("myP").firstChild;
         var newString = form.appendStr.value;
         try {
            node.appendData(newString);
         }
         catch(err) {
            handleError(err);
         }
      }

      function doDelete(form) {
         var node = document.getElementById("myP").firstChild;
         var offset = form.deleteOffset.value;
         var count = form.deleteCount.value;
         try {
            node.deleteData(offset, count);
         }
         catch(err) {
            handleError(err);
         }
      }

      function doInsert(form) {
         var node = document.getElementById("myP").firstChild;
         var offset = form.insertOffset.value;
         var newString = form.insertStr.value;
         try {
            node.insertData(offset, newString);
         }
         catch(err) {
            handleError(err);
         }
      }

      function doReplace(form) {
         var node = document.getElementById("myP").firstChild;
         var offset = form.replaceOffset.value;
         var count = form.replaceCount.value;
         var newString = form.replaceStr.value;
```

Continued

Listing 35-8 *(continued)*

```
         try {
            node.replaceData(offset, count, newString);
         }         catch(err) {
            handleError(err);
         }
      }

      function showSubstring(form) {
         var node = document.getElementById("myP").firstChild;
         var offset = form.substrOffset.value;
         var count = form.substrCount.value;
         try {
            alert(node.substringData(offset, count));
         }
         catch(err) {
            handleError(err);
         }
      }

      // error handler for these methods
      function handleError(err) {
         switch (err.name) {
         case "NS_ERROR_DOM_INDEX_SIZE_ERR":
            alert("The offset number is outside the allowable range.");
            break;
         case "NS_ERROR_DOM_NOT_NUMBER_ERR":
            alert("Make sure each numeric entry is a valid number.");
            break;
         default:
            alert("Double-check your text box entries.");
         }
      }
      </script>
   </head>
   <body>
      <h1>Data Methods of a W3C Text Object</h1>
      <hr />
      <p id="myP" style="font-weight:bold; text-align:center">
So I called myself Pip, and became to be called Pip.</p>
      <form name="choices">
         <p><input type="button" onclick="doAppend(this.form)"
            value="appendData()" /> String:<input type="text" name="appendStr"
            size="30" /></p>
         <p><input type="button" onclick="doDelete(this.form)"
            value="deleteData()" /> Offset:<input type="text"
            name="deleteOffset" size="3" /> Count:<input type="text"
            name="deleteCount" size="3" /></p>
         <p><input type="button" onclick="doInsert(this.form)"
            value="insertData()" /> Offset:<input type="text"
            name="insertOffset" size="3" /> String:<input type="text"
            name="insertStr" size="30" /></p>
         <p><input type="button" onclick="doReplace(this.form)"
            value="replaceData()" /> Offset:<input type="text"
            name="replaceOffset" size="3" /> Count:<input type="text"
            name="replaceCount" size="3" /> String:<input type="text"
            name="replaceStr" size="30" /></p>
```

```
            <p><input type="button" onclick="showSubstring(this.form)"
               value="substringData()" /> Offset:<input type="text"
               name="substrOffset" size="3" /> Count:<input type="text"
               name="substrCount" size="3" /></p>
         </form>
      </body>
   </html>
```

Related Items: `appendChild()`, `removeChild()`, `replaceChild()` methods of element objects (Chapter 15).

`splitText(`*`offset`*`)`

Returns: `Text` or `TextNode` object.
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The `splitText()` method performs multiple actions with one blow. The *`offset`* parameter is an integer indicating the zero-based index position within the text node at which the node is to divide into two nodes. After you invoke the method on the current text node, the current node consists of the text from the beginning of the node up to the offset position. The method returns a reference to the text node whose data starts with the character after the dividing point and extends to the end of the original node. Users won't notice any change in the rendered text: This method influences only the text node structure of the document. Using this method means, for example, that an HTML element that starts with only one text node will have two after the `splitText()` method is invoked. The opposite action (combining contiguous text node objects into a single node) is performed by the NN6+ `normalize()` method (see Chapter 15).

Example

Use The Evaluator (Chapter 13) to see the `splitText()` method in action. Begin by verifying that the `myEM` element has but one child node, and that its `nodeValue` is the string "all":

```
document.getElementById("myEM").childNodes.length
document.getElementById("myEM").firstChild.nodeValue
```

Next, split the text node into two pieces after the first character:

```
document.getElementById("myEM").firstChild.splitText(1)
```

Two text nodes are now inside the element:

```
document.getElementById("myEM").childNodes.length
```

Each text node contains its respective portion of the original text:

```
document.getElementById("myEM").firstChild.nodeValue
document.getElementById("myEM").lastChild.nodeValue
```

If you are using NN6+/Moz/Safari, now bring the text nodes back together:

```
document.getElementById("myEM").normalize()
document.getElementById("myEM").childNodes.length
```

At no time during these statement executions does the rendered text change.

Related Items: `normalize()` method (Chapter 15).

TextRange Object

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

Properties	*Methods*	*Event Handlers*
boundingHeight	collapse()	
boundingLeft	compareEndPoints()	
boundingTop	duplicate()	
boundingWidth	execCommand()	
htmlText	expand()	
offsetLeft†	findText()	
offsetTop†	getBookmark()	
text	getBoundingClientRect()†	
	getClientRects()†	
	inRange()	
	isEqual()	
	move()	
	moveEnd()	
	moveStart()	
	moveToBookmark()	
	moveToElementText()	
	moveToPoint()	
	parentElement()	
	pasteHTML()	
	queryCommandEnabled()	
	queryCommandIndeterm()	
	queryCommandState()	
	queryCommandSupported()	
	queryCommandValue()	
	scrollIntoView()†	
	select()	
	setEndPoint()	

†See Chapter 15.

Syntax

Creating a TextRange object:

```
var rangeRef = document.body.createTextRange();
var rangeRef = buttonControlRef.createTextRange();
var rangeRef = textControlRef.createTextRange();
var rangeRef = document.selection.createRange();
```

Accessing TextRange object properties or methods:

```
(IE4+)        rangeRef.property | method([parameters])
```

About this object

Unlike most of the objects covered in Part III of the book, the IE4+ TextRange object is not tied to a specific HTML element. The TextRange object is, instead, an abstract object that represents text content anywhere on the page (including text content of a text-oriented form control) between a start point and an end point (collectively, the *boundaries* of the range). The user may not necessarily know that a TextRange object exists, because no requirement exists to force a TextRange object to physically select text on the page (although the TextRange object can be used to assist scripts in automating the selection of text; or a script may turn a user selection into a TextRange object for further processing).

The purpose of the TextRange object is to give scripts the power to examine, modify, remove, replace, and insert content on the page. Start and end points of an IE TextRange object are defined exclusively in terms of character positions within the element that is used to create the range (usually the body element, but also button- and text-related form control elements). Character positions of body text do not take into account source code characters that may define HTML elements. This factor is what distinguishes a TextRange's behavior from, for instance, the various properties and methods of HTML elements that let you modify or copy elements and their text (for example, innerText and outerText properties). A TextRange object's start point can be in one element, and its end point in another. For example, consider the following HTML:

```
<p>And now to introduce our <em>very special</em> guest:</p>
```

If the text shown in boldface indicates the content of a TextRange object, you can see that the range crosses element boundaries in a way that makes HTML element object properties difficult to use for replacing that range with some other text. Challenges still remain in this example, however. Simply replacing the text of the range with some other text forces your script (or the browser) to reconcile the issue of what to do about the nested em element, because the TextRange object handles only its text. (Your word processing program must address the same kind of issue when you select a phrase that starts in italic but ends in normal font, and then you paste text into that selection.)

An important aspect of the TextRange object is that the size of the range can be zero or more characters. Start and end points always position themselves between characters. When the start point and end point of a range are at the same location, the range acts as a text insertion pointer. In fact, when the TextRange object represents text inside a text-related form control, the select() method of the TextRange object can be used to display the text insertion pointer where your script desires. Therefore, through the TextRange object you can script your forms to always display the text insertion pointer at the end of existing text in a text box or textarea when the control receives focus.

Working with text ranges

To create a TextRange object, use the createTextRange() method with the document.body object or any button- or text-related form control object. If you want to convert a block of selected text to a text range, use the special createRange() method of the document.selection object. Regardless of how you create it, the range encompasses the entire text of the object used to generate the range. In other words, the start point is at the very beginning of the text and the end point is at the very end. Note that when you create a TextRange object from the body element, text that is inside text-related form controls is not part of the text of the TextRange (just as text field content isn't selected if you select manually the entire text of the page).

After you create a `TextRange` object (assigned to a variable), the typical next steps involve some of the many methods associated with the object that help narrow the size of the range. Some methods (`move()`, `moveEnd()`, `moveStart()`, and `sentEndPoint()`) offer manual control over the intra-character position for the start and end points. Parameters of some of these methods understand concepts, such as words and sentences, so not every action entails tedious character counts. Another method, `moveToElementText()`, automatically adjusts the range to encompass a named element. The oft-used `collapse()` method brings the start and end points together at the beginning or end of the current range — helpful when a script must iterate through a range for tasks, such as word counting or search and replace. The `expand()` method can extend a collapsed range to encompass the whole word, whole sentence, or entire range surrounding the insertion point. Perhaps the most powerful method is `findText()`, which allows scripts to perform single or global search-and-replace operations on body text.

After the range encompasses the desired text, several other methods let scripts act on the selection. The types of operations include scrolling the page to make the text represented by the range visible to the user (`scrollIntoView()`) and selecting the text (`select()`) to provide visual feedback to the user that something is happening (or to set the insertion pointer at a location in a text form control). An entire library of additional commands are accessed through the `execCommand()` method for operations, such as copying text to the clipboard and a host of formatting commands that can be used in place of stylesheet property changes. To swap text from the range with new text accumulated by your script, you can modify the `text` property of the range.

Using the `TextRange` object can be a bit tedious, because it often requires a number of script statements to execute an action. Three basic steps are generally required to work with a `TextRange` object:

1. Create the text range.
2. Set the start and end points.
3. Act on the range.

As soon as you are comfortable with this object, you will find it provides a lot of flexibility in scripting interaction with body content. For ideas about applying the `TextRange` object in your scripts, see the examples that accompany the following descriptions of individual properties and methods.

About browser compatibility

The `TextRange` object is available only for 32-bit versions of IE4 and later. There is no support for the `TextRange` object in MacIE.

The W3C DOM and NN6+ implement a slightly different concept of text ranges in what they call the `Range` object. In many respects, the fundamental way of working with a `Range` object is the same as for a `TextRange` object: create, adjust start and end points, and act on the range. But the W3C version (like the W3C DOM itself) is more conscious of the node hierarchy of a document. Properties and methods of the W3C `Range` object reflect this node-centric point of view, so that most of the terminology for the `Range` object differs from that of the IE `TextRange` object. As of this writing, it is unknown if or when IE will implement the W3C `Range` object.

At the same time, the W3C `Range` object lacks a couple of methods that are quite useful with the IE `TextRange` object — notably `findText()` and `select()`. On the other hand, the `Range` object, as implemented in NN6+/Moz, works on all OS platforms and is also implemented in Apple Safari for Mac OS X.

The bottom-line question, then, is whether you can make range-related scripts work in both browsers. While the basic sequence of operations is the same for both objects, the scripting vocabulary is quite different. Table 35-1 presents a summary of the property and method behaviors that the two objects have in common and their respective vocabulary terms (sometimes the value of a property in one object is accessed via a method in the other object). Notice that the ways of moving individual end points are not listed in the table because the corresponding methods for each object (for example, `moveStart()` in `TextRange` versus `setStart()` in `Range`) use very different spatial paradigms.

Table 35-1: TextRange versus Range Common Denominators

TextRange Object	*Range Object*
`text`	`toString()`
`collapse()`	`collapse()`
`compareEndPoints()`	`compareEndPoints()`
`duplicate()`	`clone()`
`moveToElementText()`	`selectContents()`
`parentElement()`	`commonParent`

To blend text range actions for both object models into a single scripted page, you have to include script execution branches for each category of object model or create your own API to invoke library functions that perform the branching. On the IE side of things, too, you have to script around actions that can cause script errors when run on MacOS and other non-Windows versions of the browser.

Properties

`boundingHeight`
`boundingLeft`
`boundingTop`
`boundingWidth`

Value: Integer. Read-Only

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

Every text range has physical dimension and location on the page, even if you cannot see the range reflected graphically with highlighting. Even a text insertion pointer (meaning a collapsed text range) has a rectangle whose height equals the line height of the body text in which the insertion point resides; its width, however, is zero.

The pixel dimensions of the rectangle of a text range can be retrieved via the `boundingHeight` and `boundingWidth` properties of the `TextRange` object. When a text range extends across multiple lines, the dimensions of the rectangle are equal to the smallest single rectangle that can contain the text (a concept identical to the bounding rectangle of inline body text, as described in the `TextRectangle` object later in this chapter). Therefore, even a range consisting of one character at the end of one line and another character at the beginning of the next, force the bounding rectangle to be as wide as the paragraph element.

TextRange.boundingHeight

A text range rectangle has a physical location on the page. The top-left position of the rectangle (with respect to the browser window edge) is reported by the `boundingTop` and `boundingLeft` properties. In practice, text ranges that are generated from selections can report very odd `boundingTop` values in IE4 when the page scrolls. Use the `offsetTop` and `offsetLeft` properties for more reliable results.

Example

Listing 35-9 provides a simple playground to explore the four bounding properties (and two offset properties) of a `TextRange` object. As you select text in the big paragraph, the values of all six properties are displayed in the table. Values are also updated if you resize the window via an `onresize` event handler.

Notice, for example, if you simply click in the paragraph without dragging a selection, the `boundingWidth` property shows up as zero. This action is the equivalent of a `TextRange` acting as an insertion point.

Listing 35-9: **Exploring the Bounding TextRange Properties**

```
<html>
   <head>
      <title>TextRange Object Dimension Properties</title>
      <style type="text/css">
      td {text-align:center}
      .propName {font-family: Courier, monospace}
      </style>
      <script type="text/javascript">
      function setAndShowRangeData() {
         var range = document.selection.createRange();
         B1.innerText = range.boundingHeight;
         B2.innerText = range.boundingWidth;
         B3.innerText = range.boundingTop;
         B4.innerText = range.boundingLeft;
         B5.innerText = range.offsetTop;
         B6.innerText = range.offsetLeft;
      }
      </script>
   </head>
   <body onresize="setAndShowRangeData()">
      <h1>TextRange Object Dimension Properties</h1>
      <hr />
      <p>Select text in the paragraph below and observe the "bounding" property
         values for the TextRange object created for that selection.</p>
      <table id="results" border="1" cellspacing="2" cellpadding="2">
         <tr>
            <th>Property</th>
            <th>Pixel Value</th>
         </tr>
         <tr>
            <td class="propName">boundingHeight</td>
            <td class="count" id="B1"> </td>
         </tr>
         <tr>
            <td class="propName">boundingWidth</td>
            <td class="count" id="B2"> </td>
         </tr>
```

```
            <tr>
               <td class="propName">boundingTop</td>
               <td class="count" id="B3"> </td>
            </tr>
            <tr>
               <td class="propName">boundingLeft</td>
               <td class="count" id="B4"> </td>
            </tr>
            <tr>
               <td class="propName">offsetTop</td>
               <td class="count" id="B5"> </td>
            </tr>
            <tr>
               <td class="propName">offsetLeft</td>
               <td class="count" id="B6"> </td>
            </tr>
         </table>
         <hr />
         <p onmouseup="setAndShowRangeData()">Lorem ipsum dolor sit amet,
            consectetaur adipisicing elit, sed do eiusmod tempor incididunt ut
            labore et dolore magna aliqua. Ut enim adminim veniam, quis nostrud
            exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
            Duis aute irure dolor in reprehenderit involuptate velit esse cillum
            dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat
            non proident, sunt in culpa qui officia deseruntmollit anim id est
            laborum Et harumd und lookum like Greek to me, dereud facilis est er
            expedit.</p>
      </body>
   </html>
```

Related Items: `offsetLeft`, `offsetTop` properties of element objects (Chapter 15).

`htmlText`

Value: String. Read-Only

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The `htmlText` property returns the HTML of the text contained by a text range. If a range's start and end points are at the very edges of an element's text, the HTML tag for that element becomes part of the `htmlText` property value. Also, if the range starts in one element and ends partway in another, the tags that influence the text of the end portion become part of the `htmlText`. This property is read-only, so you cannot use it to insert or replace HTML in the text range (see the `pasteHTML()` method and various insert commands associated with the `execCommand()` method in the following section).

Example

Use The Evaluator (Chapter 13) to investigate values returned by the `htmlText` property. Use the top text box to enter the following statements and see the values in the Results box.

Begin by creating a `TextRange` object for the entire body and store the range in local variable `a`:

```
a = document.body.createTextRange()
```

Next, use the `findText()` method to set the start and end points of the text range around the word "all," which is an `em` element inside the `myP` paragraph:

```
a.findText("all")
```

The method returns `true` (see the `findText()` method) if the text is found and the text range adjusts to surround it. To prove that the text of the text range is what you think it is, examine the `text` property of the range:

```
a.text
```

Because the text range encompasses all of the text of the element, the `htmlText` property contains the tags for the element as well:

```
a.htmlText
```

If you want to experiment by finding other chunks of text and looking at both the `text` and `htmlText` properties, first restore the text range to encompass the entire body with the following statement:

```
a.expand("textEdit")
```

You can read about the `expand()` method later in this chapter. In other tests, use `findText()` to set the range to "for all" and just "for al." Then, see how the `htmlText` property exposes the `em` element's tags.

Related Items: `text` property.

`text`

Value: String. Read/Write
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

Use the `text` property to view or change the string of visible characters that comprise a text range. The browser makes some decisions for you if the range you are about to change has nested elements inside. By and large, the nested element (and any formatting that may be associated with it) is deleted, and the new text becomes part of the text of the container that houses the start point of the text range. By the same token, if the range starts in the middle of one element and ends in the parent element's text, the tag that governs the start point now wraps all of the new text.

Example

See Listing 35-11 later in this chapter for the `findText()` method to see the `text` property used to perform the replace action of a search-and-replace function.

Related Items: `htmlText` property.

Methods

`collapse([startBoolean])`

Returns: Nothing.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

Use the `collapse()` method to shrink a text range from its current size down to a single insertion point between characters. This method becomes more important than you may think at first, especially in a function that is traversing the body or large chunk of text. For example, in a typical looping word-counting script, you create a text range that encompasses the full body (or text in a `textarea`). When the range is created, its start point is at the very beginning of the text, and its end point is at the very end. To begin counting words, you can first collapse the range to the insertion point at the very beginning of the range. Next, use the `expand()` method to set the range to the first word of text (and increment the counter if the

expand() method returns true). At that point, the text range extends around the first word. What you want is for the range to collapse at the end of the current range so that the search for the next word starts after the current one. Use collapse() once more, but this time with a twist of parameters.

The optional parameter of the collapse() method is a Boolean value that directs the range to collapse itself either at the start or end of the current range. The default behavior is the equivalent of a value of true, which means that unless otherwise directed, a collapse() method shifts the text range to the point in front of the current range. That works great as an early step in the word-counting example, because you want the text range to collapse to the start of the text before doing any counting. But for subsequent movements through the range, you want to collapse the range so that it is after the current range. Thus, you include a false parameter to the collapse() method.

Example

See Listings 35-11 and 15-14 to see the collapse() method at work.

Related Items: Range.collapse(), TextRange.expand() methods.

compareEndPoints("*type*", *rangeRef*)

Returns: Integer (-1, 0, or 1).
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

Generating multiple TextRange objects and assigning them to different variables is no problem. You can then use the compareEndPoints() method to compare the relative positions of start and end points of two ranges. One range is the object that you use to invoke the compareEndPoints() method, and the other range is the second parameter of the method. The order doesn't matter, because the first parameter of the method determines which points in each range you will be comparing.

Values for the first parameter can be one of four explicit strings: StartToEnd, StartToStart, EndToStart, and EndToEnd. What these values specify is which point of the current range is compared with which point of the range passed as the second parameter. For example, consider the following body text that has two text ranges defined within it:

```
It was the best of times.
```

The first text range (assigned in our discussion here to variable rng1) is shown in boldface, while the second text range (rng2) is shown in bold-italic. In other words, rng2 is nested inside rng1. We can compare the position of the start of rng1 against the position of the start of rng2 by using the StartToStart parameter of the compareEndPoints() method:

```
var result = rng1.compareEndPoints("StartToStart", rng2);
```

The value returned from the compareEndPoints() method is an integer of one of three values. If the positions of both points under test are the same, the value returned is 0. If the first point is before the second, the value returned is -1; if the first point is after the second, the value is 1. Therefore, from the example above, because the start of rng1 is before the start of rng2, the method returns -1. If you changed the statement to invoke the method on rng2, as in

```
var result = rng2.compareEndPoints("StartToStart", rng1);
```

the result would be 1.

In practice, this method is helpful in knowing if two ranges are the same, if one of the boundary points of both ranges is the same, or if one range starts where the other ends.

Example

The page rendered by Listing 35-10 lets you experiment with text range comparisons. The bottom paragraph contains a `span` element that has a `TextRange` object assigned to its text after the page loads (in the `init()` function). That fixed range becomes a solid reference point for you to use while you select text in the paragraph. After you make a selection, all four versions of the `compareEndPoints()` method run to compare the start and end points of the fixed range against your selection. One column of the results table shows the raw value returned by the `compareEndPoints()` method, while the third column puts the results into plain language.

To see how this page works, begin by selecting the first word of the fixed text range (double-click the word). You can see that the starting positions of both ranges are the same, because the returned value is `0`. Because all of the invocations of the `compareEndPoints()` method are on the fixed text range, all comparisons are from the point of view of that range. Thus, the first row of the table for the `StartToEnd` parameter indicates that the start point of the fixed range comes before the end point of the selection, yielding a return value of `-1`.

Other selections to make include:

- ✦ Text that starts before the fixed range and ends inside the range
- ✦ Text that starts inside the fixed range and ends beyond the range
- ✦ Text that starts and ends precisely at the fixed range boundaries
- ✦ Text that starts and ends before the fixed range
- ✦ Text that starts after the fixed range

Study the returned values and the plain language results and see how they align with the selection you make.

Listing 35-10: **Lab for compareEndPoints() Method**

```
<html>
   <head>
      <title>TextRange.compareEndPoints() Method</title>
      <style type="text/css">
      td {text-align:center}
      .propName {font-family:Courier, monospace}
      #fixedRangeElem {color:red; font-weight:bold}
      </style>
      <script type="text/javascript">
      var fixedRange;

      function setAndShowRangeData() {
         var selectedRange = document.selection.createRange();
         var result1 = fixedRange.compareEndPoints("StartToEnd",
            selectedRange);
         var result2 = fixedRange.compareEndPoints("StartToStart",
            selectedRange);
         var result3 = fixedRange.compareEndPoints("EndToStart",
            selectedRange);
         var result4 = fixedRange.compareEndPoints("EndToEnd", selectedRange);

         B1.innerText = result1;
         compare1.innerText = getDescription(result1);
         B2.innerText = result2;
```

```
      compare2.innerText = getDescription(result2);
      B3.innerText = result3;
      compare3.innerText = getDescription(result3);
      B4.innerText = result4;
      compare4.innerText = getDescription(result4);
   }

   function getDescription(comparisonValue) {
      switch (comparisonValue) {
      case -1 :
         return "comes before";
         break;
      case 0 :
         return "is the same as";
         break;
      case 1 :
         return "comes after";
         break;
      default :
         return "vs.";
      }
   }

   function init() {
      fixedRange = document.body.createTextRange();
      fixedRange.moveToElementText(fixedRangeElem);
   }
   </script>
</head>
<body onload="init()">
   <h1>TextRange.compareEndPoints() Method</h1>
   <hr />
   <p>Select text in the paragraph in various places relative to the fixed
      text range (shown in red). See the relations between the fixed and
      selected ranges with respect to their start and end points.</p>
   <table id="results" border="1" cellspacing="2" cellpadding="2">
      <tr>
         <th>Property</th>
         <th>Returned Value</th>
         <th>Fixed Range vs. Selection</th>
      </tr>
      <tr>
         <td class="propName">StartToEnd</td>
         <td class="count" id="B1"> </td>
         <td class="count" id="C1">Start of Fixed <span
         id="compare1">vs.</span> End of Selection</td>
      </tr>
      <tr>
         <td class="propName">StartToStart</td>
         <td class="count" id="B2"> </td>
         <td class="count" id="C2">Start of Fixed <span
         id="compare2">vs.</span> Start of Selection</td>
      </tr>
      <tr>
         <td class="propName">EndToStart</td>
         <td class="count" id="B3"> </td>
         <td class="count" id="C3">End of Fixed <span
         id="compare3">vs.</span> Start of Selection</td>
```

Continued

Listing 35-10 *(continued)*

```
         </tr>
         <tr>
            <td class="propName">EndToEnd</td>
            <td class="count" id="B4"> </td>
            <td class="count" id="C4">End of Fixed <span
            id="compare4">vs.</span> End of Selection</td>
         </tr>
      </table>
      <hr />
      <p onmouseup="setAndShowRangeData()">Lorem ipsum dolor sit,
         <span id="fixedRangeElem">consectetaur adipisicing elit</span>,
         sed do eiusmod tempor incididunt ut labore et dolore aliqua.
         Ut enim adminim veniam, quis nostrud exercitation ullamco laboris
         nisi ut aliquip ex ea commodo consequat.</p>
   </body>
</html>
```

Related Items: `Range.compareEndPoints()` method.

`duplicate()`

Returns: `TextRange` object.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The `duplicate()` method returns a `TextRange` object that is a snapshot copy of the current `TextRange` object. In a way, a non-intuitive relationship exists between the two objects. If you alter the `text` property of the copy without moving the start or end points of the original, the original takes on the new text. But if you move the start or end points of the original, the `text` and `htmlText` of the original obviously change, while the copy retains its properties from the time of the duplication. Therefore, this method can be used to clone text from one part of the document to other parts.

Example

Use The Evaluator (Chapter 13) to see how the `duplicate()` method works. Begin by creating a new `TextRange` object that contains the text of the `myP` paragraph element:

```
a = document.body.createTextRange()
a.moveToElementText(myP)
```

Next, clone the original range and preserve the copy in variable `b`:

```
b = a.duplicate()
```

The method returns no value, so don't be alarmed by the "undefined" that appears in the Results box. Move the original range so that it is an insertion point at the end of the body by first expanding it to encompass the entire body, and then collapse it to the end:

```
a.expand("textedit")
a.collapse(false)
```

Now, insert the copy at the very end of the body:

```
a.text = b.text
```

If you scroll to the bottom of the page, you'll see a copy of the text.

Related Items: `Range.clone()`, `TextRange.isEqual()` methods.

execCommand("*commandName*"[, *UIFlag*[, *value*]])

Returns: Boolean.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

IE4+ for Win32 operating systems lets scripts access a very large number of commands that act on insertion points, abstract text ranges, and selections that are made with the help of the `TextRange` object. Access to these commands is through the `execCommand()` method, which works with `TextRange` objects and the `document` object (see the `document.execCommand()` method discussion in Chapter 18 and list of document- and selection-related commands in Table 18-3).

The first, required parameter is the name of the command that you want to execute. Only a handful of these commands offer unique capabilities that aren't better accomplished within the IE object model and stylesheet mechanism. Of particular importance is the command that lets you copy a text range into the Clipboard. Most of the rest of the commands modify styles or insert HTML tags at the position of a collapsed text range. These actions are better handled by other means, but they are included in Table 35-2 for the sake of completeness only (see Table 18-3 for additional commands).

Table 35-2: TextRange.execCommand() Commands

Command	*Parameter*	*Description*
`Bold`	None	Encloses the text range in a `<b>` tag pair
`Copy`	None	Copies the text range into the Clipboard
`Cut`	None	Copies the text range into the Clipboard and deletes it from the document or text control
`Delete`	None	Deletes the text range
`InsertButton`	ID String	Inserts a `<button>` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertFieldset`	ID String	Inserts a `<fieldset>` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertHoritontalRule`	ID String	Inserts an `<hr>` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertIFrame`	ID String	Inserts an `<iframe>` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputButton`	ID String	Inserts an `<input type="button">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertIntpuCheckbox`	ID String	Inserts an `<input type="checkbox">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputFileUpload`	ID String	Inserts an `<input type="FileUpload">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputHidden`	ID String	Inserts an `<input type="hidden">` tag at the insertion point, assigning the parameter value to the `id` attribute

Continued

Table 35-2 *(continued)*

Command	*Parameter*	*Description*
`InsertInputImage`	ID String	Inserts an `<input type="image">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputPassword`	ID String	Inserts an `<input type="password">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputRadio`	ID String	Inserts an `<input type="radio">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputReset`	ID String	Inserts an `<input type="reset">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertInputSubmit`	ID String	Inserts an `<input type="submit">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertIntputText`	ID String	Inserts an `<input type="text">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertMarquee`	ID String	Inserts a `<marquee>` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertOrderedList`	ID String	Inserts an `<ol>` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertParagraph`	ID String	Inserts a `<p>` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertSelectDropdown`	ID String	Inserts a `<select type="select-one">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertSelectListbox`	ID String	Inserts a `<select type="select-multiple">` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertTextArea`	ID String	Inserts an empty `<textarea>` tag at the insertion point, assigning the parameter value to the `id` attribute
`InsertUnorderedList`	ID String	Inserts a `<ul>` tag at the insertion point, assigning the parameter value to the `id` attribute
`Italic`	None	Encloses the text range in an `<i>` tag pair
`OverWrite`	Boolean	Sets the text input control mode to overwrite (`true`) or insert (`false`)
`Paste`	None	Pastes the current Clipboard contents into the insertion point or selection
`PlayImage`	None	Begins playing dynamic images if they are assigned to the `dynsrc` attribute of the `img` element
`Refresh`	None	Reloads the current page
`StopImage`	None	Stops playing dynamic images if they are assigned to the `dynsrc` attribute of the `img` element
`Underline`	None	Encloses the text range in a `<u>` tag pair

An optional second parameter is a Boolean flag to instruct the command to display any user interface artifacts that may be associated with the command. The default is `false`. For the third parameter, some commands require an attribute value for the command to work. For example, to insert a new paragraph at an insertion point, you pass the identifier to be assigned to the `id` attribute of the `p` element. The syntax is

```
myRange.execCommand("InsertParagraph", true, "myNewP");
```

The `execCommand()` method returns Boolean `true` if the command is successful; `false` if not successful. Some commands can return values (for example, finding out the font name of a selection), but these values are accessed through the `queryCommandValue()` method.

Although the commands in Table 35-2 work on text ranges, even the commands that work on selections (Table 18-3) can frequently benefit from some preprocessing with a text range. Consider, for example, a function whose job it is to find every instance of a particular word in a document and set its background color to a yellow highlight. Such a function utilizes the powers of the `findText()` method of a text range to locate each instance. Then the `select()` method selects the text in preparation for applying the `BackColor` command. Here is a sample:

```
function hiliteIt(txt) {
   var rng = document.body.createTextRange();
   for (var i = 0; rng.findText(txt); i++) {
      rng.select();
      rng.execCommand("BackColor", "false", "yellow");
      rng.execCommand("Unselect");
      // prepare for next search
      rng.collapse(false);
   }
}
```

This example is a rare case that makes the `execCommand()` method way of modifying HTML content more efficient than trying to wrap some existing text inside a new tag. The downside is that you don't have control over the methodology used to assign a background color to a span of text (in this case, IE wraps the text in a `<font>` tag with a `style` attribute set to `background-color:yellow` — probably not the way you'd do it on your own).

Example

Use The Evaluator (Chapter 13) to see how to copy a text range's text into the client computer's Clipboard. Begin by setting the text range to the `myP` element:

```
a = document.body.createTextRange()
a.moveToElementText(myP)
```

Now use `execCommand()` to copy the range into the Clipboard:

```
a.execCommand("Copy")
```

To prove that the text is in the Clipboard, click the bottom text field and choose Paste from the Edit menu (or type Ctrl+V).

Related Items: Several query command methods of the `TextRange` object.

expand("*unit*")

Returns: Boolean.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The single `expand()` method can open any range — collapsed or not — to the next largest character, word, or sentence or to the entire original range (for example, encompassing the text of the `body` element if the range was generated by `document.body.createTextRange()`). The parameter is a string designating which unit to expand outward to: `character`, `word`, `sentence`, or `textedit`. If the operation is successful, the method returns `true`; otherwise it returns `false`.

When operating from an insertion point, the `expand()` method looks for the word or sentence that encloses the point. The routine is not very smart about sentences, however. If you have some text prior to a sentence that you want to expand to, but that text does not end in a period, the `expand()` routine expands outward until it can find either a period or the beginning of the range. Listing 15-14 demonstrates a workaround for this phenomenon. When expanding from an insertion point to a character, the method expands forward to the next character in language order. If the insertion point is at the end of the range, it cannot expand to the next characters, and the `expand()` method returns `false`.

It is not uncommon in an extensive script that needs to move the start and end points all over the initial range to perform several `collapse()` and `expand()` method operations from time to time. Expanding to the full range is a way to start some range manipulation with a clean slate, as if you just created the range.

Example

You can find examples of the `expand()` method in Listing 15-14.

Related Items: `TextRange.collapse()` method.

`findText("searchString"[, searchScope, flags])`

Returns: Boolean.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

One of the most useful methods of the `TextRange` object is `findText()`, whose default behavior is to look through a text range starting at the range's start point up to the end of the range in search of a case-insensitive match for a search string. If an instance is found in the range, the start and end points of the range are cinched up to the found text and the method returns `true`; otherwise it returns `false`, and the start and end points do not move. Only the rendered text is searched and not any of the tags or attributes.

Optional parameters let you exert some additional control over the search process. You can restrict the distance from a collapsed range to be used for searching. The *`searchScope`* parameter is an integer value indicating the number of characters from the start point. The larger the number, the more text of the range is included in the search. Negative values force the search to operate backward from the current start point. If you want to search backward to the beginning of the range, but you don't know how far away the start of the range is, you can specify an arbitrarily huge number that would encompass the text.

The optional *`flags`* parameter lets you set whether the search is to be case-sensitive and/or to match whole words only. The parameter is a single integer value that uses bit-wise math to calculate the single value that accommodates one or both settings. The value for matching whole words is `2`; the value for matching case is `4`. If you want only one of those behaviors, then supply just the desired number. But for both behaviors, use the bit-wise XOR operator (the `^` operator) on the values to reach a value of `6`.

The most common applications of the `findText()` method include a search-and-replace action and format changes to every instance of a string within the range. This iterative process requires some extra management of the process. Because searching always starts with the range's current start point, advancing the start point to the end of the text found in the range is necessary. This advancing allows a successive application of `findText()` to look through the rest of the range for another match. And because `findText()` ignores the arbitrary end point of the current range and continues to the end of the initial range, you can use the `collapse(false)` method to force the starting point to the end of the range that contains the first match.

A repetitive search can be accomplished by a `while` or `for` repeat loop. The Boolean returned value of the `findText()` method can act as the condition for continuing the loop. If the number of times the search succeeds is not essential to your script, a `while` loop works nicely:

```
while (rng.findText(searchString)) {
   ...
   rng.collapse(false);
}
```

Or you can use a `for` loop counter to maintain a count of successes, such as a counter of how many times a string appears in the body:

```
for (var i = 0; rng.findText(searchString); i++) {
   ...
   rng.collapse(false);
}
```

Some of the operations you want to perform on a range (such as many of the commands invoked by the `execCommand()` method) require that a selection exists for the command to work. Be prepared to use the `select()` method on the range after the `findText()` method locates a matching range of text.

Example

Listing 35-11 implements two varieties of text search-and-replace operation, while showing you how to include extra parameters for case-sensitive and whole-word searches. Both approaches begin by creating a `TextRange` for the entire body, but they immediately shift the starting point to the beginning of the `div` element that contains the text to search.

One search-and-replace function prompts the user to accept or decline replacement for each instance of a found string. The `select()` and `scrollIntoView()` methods are invoked to help the user see what is about to be replaced. Notice that even when the user declines to accept the replacement, the text range is collapsed to the end of the found range so that the next search can begin after the previously found text. Without the `collapse()` method, the search can get caught in an infinite loop as it keeps finding the same text over and over (with no replacement made). Because no counting is required, this search-and-replace operation is implemented inside a `while` repeat loop.

The other search-and-replace function goes ahead and replaces every match and then displays the number of replacements made. After the loop exits (because there are no more matches), the loop counter is used to display the number of replacements made.

Listing 35-11: **Two Search-and-Replace Approaches (with Undo)**

```
<html>
   <head>
      <title>TextRange.findText() Method</title>
      <script type="text/javascript">
      // global range var for use with Undo
      var rng;

      // return findText() third parameter arguments
      function getArgs(form) {
         var isCaseSensitive = (form.caseSensitive.checked) ? 4 : 0;
         var isWholeWord = (form.wholeWord.checked) ? 2 : 0;
         return isCaseSensitive ^ isWholeWord;
```

Continued

Listing 35-11 *(continued)*

```
}

// prompted search and replace
function sAndR(form) {
   var srchString = form.searchString.value;
   var replString = form.replaceString.value;
   if (srchString) {
      var args = getArgs(form);
      rng = document.body.createTextRange();
      rng.moveToElementText(rights);
      clearUndoBuffer();
      while (rng.findText(srchString, 10000, args)) {
         rng.select();
         rng.scrollIntoView();
         if (confirm("Replace?")) {
            rng.text = replString;
            pushUndoNew(rng, srchString, replString);
         }
         rng.collapse(false);
      }
   }
}

// unprompted search and replace with counter
function sAndRCount(form) {
   var srchString = form.searchString.value;
   var replString = form.replaceString.value;
   var i;
   if (srchString) {
      var args = getArgs(form);
      rng = document.body.createTextRange();
      rng.moveToElementText(rights);
      for (i = 0; rng.findText(srchString, 10000, args); i++) {
         rng.text = replString;
         pushUndoNew(rng, srchString, replString);
         rng.collapse(false);
      }
      if (i > 1) {
         clearUndoBuffer();
      }
   }
   document.getElementById("counter").innerText = i;
}

// BEGIN UNDO BUFFER CODE
// buffer global variables
var newRanges = new Array();
var origSearchString;

// store original search string and bookmarks of each replaced range
function pushUndoNew(rng, srchString, replString) {
   origSearchString = srchString;
   rng.moveStart("character", -replString.length);
   newRanges[newRanges.length] = rng.getBookmark();
}
```

```
      // empty array and search string global
      function clearUndoBuffer() {
         document.getElementById("counter").innerText = "0";
         origSearchString = "";
         newRanges.length = 0;
      }

      // perform the undo
      function undoReplace() {
         if (newRanges.length && origSearchString) {
            for (var i = 0; i < newRanges.length; i++) {
               rng.moveToBookmark(newRanges[i]);
               rng.text = origSearchString;
            }
            document.getElementById("counter").innerText = i;
            clearUndoBuffer();
         }
      }
      </script>
   </head>
   <body>
      <h1>TextRange.findText() Method</h1>
      <hr />
      <form>
         <p>Enter a string to search for in the following text: <input
            type="text" name="searchString" size="20" value="Law" />  
            <input type="checkbox" name="caseSensitive" />Case-sensitive  
            <input type="checkbox" name="wholeWord" />Whole words only</p>
         <p>Enter a string with which to replace found text: <input type="text"
            name="replaceString" size="20" value="legislation" /></p>
         <p><input type="button" value="Search and Replace (with prompt)"
            onclick="sAndR(this.form)" /></p>
         <p><input type="button" value="Search, Replace, and Count (no prompt)"
            onclick="sAndRCount(this.form)" /> <span id="counter">0</span>
            items found and replaced.</p>
         <p><input type="button" value="Undo Search and Replace"
            onclick="undoReplace()" /></p>
      </form>
      <div id="rights">
         <h2><a id="article1" name="article1">ARTICLE I</a></h2>
         <p>Congress shall make no law respecting an establishment of religion,
            or prohibiting the free exercise thereof; or abridging the freedom
            of speech, or of the press; or the right of the people peaceably to
            assemble, and to petition the government for a redress of
            grievances.</p>
         [The rest of the text is snipped for printing here, but it is on the
         CD-ROM version.]
      </div>
   </body>
</html>
```

Having a search-and-replace function available in a document is only one-half of the battle The other half is offering the facilities to undo the changes. To that end, Listing 35-11 includes an undo buffer that accurately undoes only the changes made in the initial replacement actions.

The undo buffer stores its data in two global variables. The first, `origSearchString`, is simply the string used to perform the original search. This variable is the string that has to be put back in the places where it had been replaced. The second global variable is an array that stores `TextRange` bookmarks (see `getBookmark()` later in this chapter). These references are string values that don't mean much to humans, but the browser can use them to re-create a range with its desired start and end point. Values for both the global search string and bookmark specifications are stored in calls to the `pushUndoNew()` method each time text is replaced.

A perhaps unexpected action of setting the `text` property of a text range is that the start and end points collapse to the end of the new text. Because the stored bookmark must include the replaced text as part of its specification, the start point of the current range must be adjusted back to the beginning of the replacement text before the bookmark can be saved. Thus, the `pushUndoNew()` function receives the replacement text string so that the `moveStart()` method can be adjusted by the number of characters matching the length of the replacement string.

After all of the bookmarks are stored in the array, the undo action can do its job in a rather simple `for` loop inside the `undoReplace()` function. After verifying that the undo buffer has data stored in it, the function loops through the array of bookmarks and replaces the bookmarked text with the old string. The benefit of using the bookmarks rather than using the replacement function again is that only those ranges originally affected by the search-and-replace operation are touched in the undo operation. For example, in this document if you replace a case-sensitive "states" with "States" two replacements are performed. At that point, however, the document has four instances of "States," two of which existed before. Redoing the replacement function by inverting the search-and-replace strings would convert all four back to the lowercase version — not the desired effect.

Related Items: `TextRange.select()` method.

`getBookmark()`

Returns: Bookmark string.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

In the context of a `TextRange` object, a bookmark is not to be confused with the kinds of bookmarks you add to a browser list of favorite Web sites. Instead, a bookmark is a string that represents a definition of a text range, including its location in a document, its text, and so on. Viewing the string is futile, because it contains string versions of binary data, so the string means nothing in plain language. But a bookmark allows your scripts to save the current state of a text range so that it may be restored at a later time. The `getBookmark()` method returns the string representation of a snapshot of the current text range. Some other script statement can adjust the `TextRange` object to the exact specifications of the snapshot with the `moveToBookmark()` method (described later in this chapter).

Example

Listing 35-11 earlier in this chapter shows how the `getBookmark()` method is used to preserve specifications for text ranges so that they can be called upon again to be used to undo changes made to the text range. The `getBookmark()` method is used to save the snapshots, while the `moveToBookmark()` method is used during the undo process.

Related Items: `TextRange.moveToBookmark()` method.

inRange(*otherRangeRef*)

Returns: Boolean.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

You can compare the physical stretches of text contained by two different text ranges via the `inRange()` method. Typically, you invoke the method on the larger of two ranges and pass a reference to the smaller range as the sole parameter to the method. If the range passed as a parameter is either contained by or equal to the text range that invokes the method, the method returns `true`; otherwise the method returns `false`.

Example

Use The Evaluator (Chapter 13) to see the `inRange()` method in action. The following statements generate two distinct text ranges, one for the `myP` paragraph element and the other for the `myEM` element nested within:

```
a = document.body.createTextRange()
a.moveToElementText(myP)
b = document.body.createTextRange()
b.moveToElementText(myEM)
```

Because the `myP` text range is larger than the other, invoke the `inRange()` method on it, fully expecting the return value of `true`:

```
a.inRange(b)
```

But if you switch the references, you see that the larger text range is not "in" the smaller one:

```
b.inRange(a)
```

Related Items: `TextRange.isEqual()` method.

isEqual(*otherRangeRef*)

Returns: Boolean.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

If your script has references to two independently adjusted `TextRange` objects, you can use the `isEqual()` method to test whether the two objects are identical. This method tests for a very literal equality, requiring that the text of the two ranges be character-for-character and position-for-position equal in the context of the original ranges (for example, body or text control content). To see if one range is contained by another, use the `inRange()` method instead.

Example

Use The Evaluator (Chapter 13) to try the `isEqual()` method. Begin by creating two separate `TextRange` objects, one for the `myP` element and one for `myEM`:

```
a = document.body.createTextRange()
a.moveToElement(myP)
b = document.body.createTextRange()
b.moveToElement(myEM)
```

Because these two ranges encompass different sets of text, they are not equal, as the results show from the following statement:

```
a.isEqual(b)
```

But if you now adjust the first range boundaries to surround the `myEM` element, both ranges are the same values:

```
a.moveToElement(myEM)
a.isEqual(b)
```

Related Items: `TextRange.inRange()` method.

move("*unit*"[, *count*])

Returns: Integer.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The `move()` method performs two operations. First, the method collapses the current text range to become an insertion point at the location of the previous end point. Next, it moves that insertion point to a position forward or backward any number of character, word, or sentence units. The first parameter is a string specifying the desired unit (`character`, `word`, `sentence`, or `textedit`). A value of `textedit` moves the pointer to the beginning or end of the entire initial text range. If you omit the second parameter, the default value is `1`. Otherwise you can specify an integer indicating the number of units the collapsed range should be moved ahead (positive integer) or backward (negative). The method returns an integer revealing the exact number of units the pointer is able to move — if you specify more units than are available, the returned value lets you know how far it can go.

Bear in mind that the range is still collapsed after the `move()` method executes. Expanding the range around desired text is the job of other methods.

You can also use the `move()` method in concert with the `select()` method to position the flashing text insertion pointer within a text box or textarea. Thus, you can script a text field, upon receiving focus or the page loading, to have the text pointer waiting for the user at the end of existing text. A generic function for such an action is shown in the following:

```
function setCursorToEnd(elem) {
   if (elem) {
      if (elem.type && (elem.type == "text" || elem.type == "textarea")) {
         var rng = elem.createTextRange();
         rng.move("textedit");
         rng.select();
      }
   }
}
```

You can then invoke this method from a text field's `onfocus` event handler:

```
<input type="text" ... onfocus="setCursorToEnd(this)" />
```

The function previously shown includes a couple of layers of error checking, such as making sure that the function is invoked with a valid object as a parameter and that the object has a `type` property whose value is one capable of having a text range made for its content.

Example

Use The Evaluator (Chapter 13) to experiment with the `move()` method. To see how the method returns just the number of units it moves the pointer, begin by creating a text range and set it to enclose the `myP` element:

```
a = document.body.createTextRange()
a.moveToElementText(myP)
```

Now enter the following statement to collapse and move the range backward by 20 words:

```
a.move("word", -20)
```

Continue to click the Evaluate button and watch the returned value in the Results box. The value shows `20` while it can still move backward by 20 words. But eventually the last movement will be some other value closer to zero. And after the range is at the beginning of the `body` element, the range can move no more in that direction, so the result is zero.

Related Items: `TextRange.moveEnd()`, `TextRange.moveStart()` methods.

moveEnd("*unit*"[, *count*])
moveStart("*unit*"[, *count*])

Returns: Integer.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The `moveEnd()` and `moveStart()` methods are similar to the `move()` method, but they each act only on the end and starting points of the current range, respectively. In other words, the range does not collapse before the point is moved. These methods allow you to expand or shrink a range by a specific number of units by moving only one of the range's boundaries.

The first parameter is a string specifying the desired unit (`character`, `word`, `sentence`, or `textedit`). A value of `textedit` moves the pointer to the beginning or end of the entire initial text range. Therefore, if you want the end point of the current range to zip to the end of the body (or text form control), use `moveEnd("textedit")`. If you omit the second parameter, the default value is `1`. Otherwise you can specify an integer indicating the number of units the collapsed range is to move ahead (positive integer) or backward (negative). Moving either point beyond the location of the other forces the range to collapse and move to the location specified by the method. The method returns an integer revealing the exact number of units the pointer is able to move — if you specify more units than are available, the returned value lets you know how far it can go.

Example

Use The Evaluator (Chapter 13) to experiment with the `moveEnd()` and `moveStart()` methods. Begin by creating a text range and set it to enclose the `myEM` element:

```
a = document.body.createTextRange()
a.moveToElementText(myEM)
```

To help you see how movements of the pointers affect the text enclosed by the range, type `a` into the bottom text box and view all the properties of the text range. Note especially the `htmlText` and `text` properties.

Now enter the following statement to move the end of the range forward by one word:

```
a.moveEnd("word")
```

Click the List Properties button to see that the text of the range now includes the word following the `em` element. Try each of the following statements in the top text box and examine both the integer results and (by clicking the List Properties button) the properties of the range after each statement:

```
a.moveStart("word", -1)
a.moveEnd("sentence")
```

Notice that for a sentence, a default unit of 1 expands to the end of the current sentence. And if you move the start point backward by one sentence, you'll see that the lack of a period-ending sentence prior to the `myP` element causes strange results.

Finally, force the start point backward in increments of 20 words and watch the results as the starting point nears and reaches the start of the `body`:

```
a.moveStart("word", -20)
```

Eventually the last movement will be some other value closer to zero. And as soon as the range is at the beginning of the `body` element, the range can move no more in that direction, so the result is zero.

Related Items: `TextRange.move()` method.

`moveToBookmark("bookmarkString")`

Returns: Boolean.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

If a snapshot of a text range specification has been preserved in a variable (with the help of the `getBookmark()` method), the `moveToBookmark()` method uses that bookmark string as its parameter to set the text range to exactly the way it appeared when the bookmark was originally obtained. If the method is successful, it returns a value of `true`, and the text range is set to the same string of text as originally preserved via `getBookmark()`. It is possible that the state of the content of the text range has been altered to such an extent that resurrecting the original text range is not feasible. In that case, the method returns `false`.

Example

Listing 35-11 earlier in this chapter shows how to use the `moveToBookmark()` method to restore a text range so that changes that created the state saved by the bookmark can be undone. The `getBookmark()` method is used to save the snapshots, while the `moveToBookmark()` method is used during the undo process.

Related Items: `TextRange.getBookmark()` method.

`moveToElementText(elemObjRef)`

Returns: Nothing.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The fastest way to cinch up a text range to the boundaries of an HTML element on the page is to use the `moveToElementText()` method. Any valid reference to the HTML element object is accepted as the sole parameter — just don't try to use a string version of the object ID unless it is wrapped in the `document.getElementById()` method (IE5+). When the boundaries are moved to the element, the range's `htmlText` property contains the tags for the element.

Example

A majority of examples for other `TextRange` object methods in this chapter use the `moveToElementText()` method. Listings 35-10 and 35-11 earlier in this chapter show the method within an application context.

Related Items: `TextRange.parentElement()` method.

`moveToPoint(x, y)`

Returns: Nothing.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The moveToPoint() method shrinks the current text range object to an insertion point and then moves it to a position in the current browser window or frame. You control the precise position via the *x* (horizontal) and *y* (vertical) pixel coordinates specified as parameters. The position is relative to the visible window, and not the document, which may have been scrolled to a different position. Invoking the moveToPoint() method is the scripted equivalent of the user clicking that spot in the window. Use the expand() method to flesh out the collapsed text range to encompass the surrounding character, word, or sentence.

Using the moveToPoint() method on a text range defined for a text form control may cause a browser crash. The method appears safe with the document.body text ranges, even if the x,y position falls within the rectangle of a text control. Such a position, however, does not drop the text range into the form control or its content.

Example

Use The Evaluator (Chapter 13) to see the moveToPoint() method in action. Begin by creating a text range for the entire body element:

```
a = document.body.createTextRange()
```

Now, invoke the moveToPoint() method to a location 100,100, which turns out to be in the rectangle space of the Results textarea:

```
a.moveToPoint(100,100)
```

If you type a into the bottom text box and view the properties, both the htmlText and text properties are empty because the insertion point represents no visible text content. But if you gradually move, for example, the start point backward one character at a time, you will see the htmlText and text properties begin to fill in with the body text that comes before the textarea element, namely the "Results:" label and the <br> tag between it and the textarea element. Enter the following statement into the top text box and click the Evaluate button several times:

```
a.moveStart("character", -1)
```

Enter a into the bottom text box after each evaluation to list the properties of the range.

Related Items: TextRange.move(), TextRange.moveStart(), TextRange.moveEnd() methods.

parentElement()

Returns: Element object reference.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The parentElement() method returns a reference to the next outermost HTML element container that holds the text range boundaries. If the text range boundaries are at the boundaries of a single element, the parentElement() method returns that element's reference. But if the boundaries straddle elements, the object returned by the method is the element that contains the text of the least-nested text portion. In contrast to the expand() and various move-related methods, which understand text constructs, such as words and sentences, the parentElement() method is concerned solely with element objects. Therefore, if a text range is collapsed to an insertion point in body text, you can expand it to encompass the HTML element by using the parentElement() method as a parameter to moveToElementText():

```
rng.moveToElementText(rng.parentElement());
```

Example

Use The Evaluator (Chapter 13) to experiment with the `parentElement()` method. Begin by setting the text range to the `myEM` element:

```
a = document.body.createTextRange()
a.moveToElementText(myEM)
```

To inspect the object returned by the `parentElement()` method, enter the following statement in the lower text box:

```
a.parentElement()
```

If you scroll down to the `outerHTML` property, you see that the parent of the text range is the `myEM` element, tag and all.

Next, extend the end point of the text range by one word:

```
a.moveEnd("word")
```

Because part of the text range now contains text of the `myP` object, the `outerHTML` property of `a.parentElement()` shows the entire `myP` element and tags.

Related Items: `TextRange.expand()`, `TextRange.move()`, `TextRange.moveEnd()`, `TextRange.moveStart()` methods.

`pasteHTML("HTMLText")`

Returns: Nothing.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

While the `execCommand()` method offers several commands that insert HTML elements into a text range, it is probably more convenient to simply paste fully formed HTML into the current text range (assuming you need to be working with a text range instead of even more simply setting new values to an element object's `outerHTML` property). Provide the HTML to be inserted as a string parameter to the `pasteHTML()` method.

Use the `pasteHTML()` method with some forethought. Some HTML that you may attempt to paste into a text range may force the method to wrap additional tags around the content you provide to ensure the validity of the resulting HTML. For example, if you were to replace a text range consisting of a portion of text of a `p` element with, for instance, an `li` element, the `pasteHTML()` method has no choice but to divide the `p` element into two pieces, because a `p` element is not a valid container for a solo `li` element. This division can greatly disrupt your document object hierarchy, because the divided `p` element assumes the same ID for both pieces. Existing references to that `p` element will break, because the reference now returns an array of two like-named objects.

Example

Use The Evaluator (Chapter 13) to experiment with the `pasteHTML()` method. The goal of the following sequence is to change the `<em>` tag to a `<span>` tag whose `style` attribute sets the color of the original text that was in the `em` element.

Begin by creating the text range and setting the boundaries to the `myEM` element:

```
a = document.body.createTextRange()
a.moveToElementText(myEM)
```

While you can pass the HTML string directly as a parameter to `pasteHTML()`, storing the HTML string in its own temporary variable may be more convenient (and more easily testable), such as:

```
b = "<span style='color:red'>" + a.text + "</span>"
```

Notice that we concatenate the text of the current text range, because it has not yet been modified. Now we can paste the new HTML string into the current text range:

```
a.pasteHTML(b)
```

At this point the `em` element is gone from the object model, and the `span` element is in its place. Prove it to yourself by looking at the HTML for the `myP` element:

```
myP.innerHTML
```

As noted earlier, the `pasteHTML()` method is not the only way to insert or replace HTML in a document. This method makes excellent sense when the user selects some text in the document to be replaced, because you can use the `document.selection.createRange()` method to get the text range for the selection. But if you're not using text ranges for other related operations, consider the other generic object properties and methods available to you.

Related Items: `outerHTML` property; `insertAdjacenHTML()` method.

queryCommandEnabled("*commandName*")
queryCommandIndeterm("*commandName*")
queryCommandState("*commandName*")
queryCommandSupported("*commandName*")
queryCommandText("*commandName*")
queryCommandValue("*commandName*")

Returns: See `document.queryCommandEnabled()` in Chapter 18.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

See descriptions under `document.queryCommandEnabled()` in Chapter 18.

select()

Returns: Nothing.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The `select()` method selects the text inside the boundaries of the current text range. For some operations, such as prompted search and replace, it is helpful to show the user the text of the current range to highlight what text is about to be replaced. In some other operations, especially several commands invoked by `execCommand()`, the operation works only on a text selection in the document. Thus, you can use the `TextRange` object facilities to set the boundaries, followed by the `select()` method to prepare the text for whatever command you like. Text selected by the `select()` method becomes a `selection` object (covered earlier in this chapter).

Example

See Listing 35-11 earlier in this chapter for an example of the `select()` method in use.

Related Items: `selection` object.

TextRange.setEndPoint()

setEndPoint("*type*", *otherRangeRef*)

Returns: Nothing.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

In contrast to the `moveEnd()` method, which adjusts the end point of the current range with respect to characters, words, sentences, and the complete range, the `setEndPoint()` method sets a boundary of the current range (not necessarily the ending boundary) relative to a boundary of another text range whose reference is passed as the second parameter. The first parameter is one of four types that control which boundary of the current range is to be adjusted and which boundary of the other range is the reference point. Table 35-3 shows the four possible values and their meanings.

Table 35-3: setEndPoint() Method Types

Type	*Description*
`StartToEnd`	Moves the start point of the current range to the end of the other range
`StartToStart`	Moves the start point of the current range to the start of the other range
`EndToStart`	Moves the end point of the current range to the start of the other range
`EndToEnd`	Moves the end point of the current range to the end of the other range

Note that the method moves only one boundary of the current range at a time. If you want to make two ranges equal to each other, you have to invoke the method twice, once with `StartToStart` and once with `EndToEnd`. At that instant, the `isEqual()` method applied to those two ranges returns `true`.

Setting a boundary point with the `setEndPoint()` method can have unexpected results when the revised text range straddles multiple elements. Don't be surprised to find that the new HTML text for the revised range does not include tags from the outer element container.

Example

Use The Evaluator (Chapter 13) to experiment with the `setEndPoint()` method. Begin by creating two independent text ranges, one for the `myP` element and one for `myEM`:

```
a = document.body.createTextRange()
a.moveToElementText(myP)
b = document.body.createTextRange()
b.moveToElementText(myEM)
```

Before moving any end points, compare the HTML for each of those ranges:

```
a.htmlText
b.htmlText
```

Now, move the start point of the `a` text range to the end point of the `b` text range:

```
a.setEndPoint("StartToEnd", b)
```

If you now view the HTML for the a range,

```
a.htmlText
```

you see that the <p> tag of the original a text range is nowhere to be found. This demonstration is a good lesson to use the setEndPoint() method primarily if you are concerned only with visible body text being inside ranges, rather than an element with its tags.

Related Items: TextRange.moveEnd(), TextRange.moveStart(), TextRange.moveToElementText() methods.

TextRectangle Object

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

Properties	*Methods*	*Event Handlers*
bottom		
left		
right		
top		

Syntax

Accessing TextRectangle object properties:

```
(IE5+) [window.]document.getElementById("elemID").
          getBoundingClientRect().property
(IE5+) [window.]document.getElementById("elemID").
          getClientRects()[i].property
```

About this object

The IE5+ TextRectangle object exposes to scripts a concept that is described in the HTML 4.0 specification, whereby an element's rendered text occupies a rectangular space on the page just large enough to contain the text. For a single word, the rectangle is as tall as the line height for the font used to render the word and no wider than the space occupied by the text. But for a sequence of words that wraps to multiple lines, the rectangle is as tall as the line height times the number of lines and as wide as the distance between the leftmost and rightmost character edges, even if it means that the rectangle encloses some other text that is not part of the element.

If you extract the TextRectangle object for an element by way of, for example, the getBoundingClientRect() method, be aware that the object is but a snapshot of the rectangle when the method was invoked. Resizing the page may very well alter dimensions of the actual rectangle enclosing the element's text, but the TextRectangle object copy that you made previously does not change its values to reflect the element's physical changes. After a window resize or modification of body text, any dependent TextRectangle objects should be recopied from the element.

Properties

bottom

left
right
top

Values: Integers. Read-Only
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The screen pixel coordinates of its four edges define every `TextRectangle` object. These coordinates are relative to the window or frame displaying the page. Therefore, if you intend to align a positioned element with an inline element's `TextRectangle`, your position assignments must take into account the scrolling of the body.

To my eye, the left edge of a `TextRectangle` does not always fully encompass the left-most pixels of the rendered text. You may have to fudge a few pixels in the measure when trying to align a real element with the `TextRectangle` of another element.

Example

Listing 35-12 lets you click one of four nested elements to see how the `TextRectangle` is treated. When you click one of the elements, that element's `TextRectangle` dimension properties are used to set the size of a positioned element that highlights the space of the rectangle. Be careful not to confuse the visible rectangle object that you see on the page with the abstract `TextRectangle` object that is associated with each of the clicked elements.

An important part of the listing is the way the action of sizing and showing the positioned element is broken out as a separate function (`setHiliter()`) from the one that is the `onclick` event handler function (`handleClick()`). This is done so that the `onresize` event handler can trigger a script that gets the current rectangle for the last element clicked, and the positioned element can be sized and moved to maintain the highlight of the same text. As an experiment, try removing the `onresize` event handler from the `<body>` tag and watch what happens to the highlighted rectangle after you resize the browser window: the rectangle that represents the `TextRectangle` remains unchanged and loses track of the abstract `TextRectangle` associated with the actual element object.

Listing 35-12: **Using the TextRectangle Object Properties**

```
<html>
   <head>
      <title>TextRectangle Object</title>
      <script type="text/javascript">
      // preserve reference to last clicked elem so resize can re-use it
      var lastElem;
      // TextRectangle left tends to be out of registration by a couple of
      // pixels
      var rectLeftCorrection = 2;

      // process mouse click
```

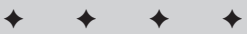
```
      function handleClick() {
         var elem = event.srcElement;
         if (elem.className && elem.className == "sample") {
            // set hiliter element only on a subset of elements
            lastElem = elem;
            setHiliter();
         } else {
            // otherwise, hide the hiliter
            hideHiliter();
         }
      }

      function setHiliter() {
         if (lastElem) {
            var textRect = lastElem.getBoundingClientRect();
            hiliter.style.pixelTop = textRect.top + document.body.scrollTop;
            hiliter.style.pixelLeft = textRect.left +
               document.body.scrollLeft - rectLeftCorrection;
            hiliter.style.pixelHeight = textRect.bottom - textRect.top;
            hiliter.style.pixelWidth = textRect.right - textRect.left;
            hiliter.style.visibility = "visible";
         }
      }

      function hideHiliter() {
         hiliter.style.visibility = "hidden";
         lastElem = null;
      }
      </script>
   </head>
   <body onclick="handleClick()" onresize="setHiliter()">
      <h1>TextRectangle Object</h1>
      <hr />
      <p>Click on any of the four colored elements in the paragraph below and
         watch the highlight rectangle adjust itself to the element's
         TextRectangle object.</p>
      <p class="sample">Lorem ipsum dolor sit amet, <span class="sample"
         style="color:red">consectetaur adipisicing elit</span>, sed do eiusmod
         tempor <span class="sample" style="color:green">incididunt ut labore
         et dolore <span class="sample" style="color:blue">magna aliqua</span>.
         Ut enim adminim veniam, quis nostrud exercitation ullamco</span>
         laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor
         in reprehenderit involuptate velit esse cillum dolore eu fugiat nulla
         pariatur.</p>
      <div id="hiliter"
      style="position:absolute; background-color:salmon; z-index:-1;
      visibility:hidden">
      </div>
   </body>
</html>
```

Related Items: `getBoundingClientRect()`, `getClientRects()` methods of element objects (Chapter 15).

✦ ✦ ✦

PART V

Appendixes



In This Part

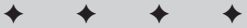
Appendix A
JavaScript and Browser Object Quick Reference

Appendix B
JavaScript Reserved Words

Appendix C
Answers to Tutorial Exercises

Appendix D
JavaScript and DOM Internet Resources

Appendix E
What's on the CD-ROM



APPENDIX

JavaScript and Browser Object Quick Reference

Core JavaScript/JScript/ECMAScript Quick Reference

© 2004 Danny Goodman (www.dannyg.com). All Rights Reserved.

JSB5

Array N3,J2 — 30

constructor N4, IE4
length
prototype

concat(*array2*)N4, IE4
join(*"char"*)IE4
push()N4, W5.5
pop()N4, W5.5
reverse()IE4
shift()N4, W5.5
slice(i,[j]) N4, IE4
sort(*compareFunc*)IE4
splice(i,j[, *items*])N4, W5.5
toLocaleString() N6, W5.5
toString()IE4
unshift()N4, W5.5

Date — 29

constructor N4, IE4
prototype N3, IE4

getFullYear()N4, J2
getYear()
getMonth()
getDate()
getDay()
getHours()
getMinutes()
getSeconds()
getTime()
getMilliseconds() N4, J2
getUTCFullYear()N4, J2
getUTCMonth()N4, J2
getUTCDate()N4, J2
getUTCDay()N4, J2
getUTCHours()N4, J2
getUTCMinutes()N4, J2
getUTCSeconds()N4, J2
getUTCMilliseconds() N4, J2
setYear(*val*)
setFullYear(*val*)N4, J2
setMonth(*val*)
setDate(*val*)
setDay(*val*)
setHours(*val*)
setMinutes(*val*)
setSeconds(*val*)
setMilliseconds(*val*)N4, J2
setTime(*val*)
setUTCFullYear(*val*)N4, J2
setUTCMonth(*val*)N4, J2
setUTCDate(*val*)N4, J2
setUTCDay(*val*)N4, J2
setUTCHours(*val*)N4, J2
setUTCMinutes(*val*)N4, J2
setUTCSeconds(*val*)N4, J2
setUTCMilliseconds(*val*)N4, J2
getTimezoneOffset()
toDateString() W5.5
toGMTString()
toLocaleDateString() W5.5, N6
toLocaleString()
toLocaleTimeString() W5.5, N6
toString()
toTimeString()IE5.5
toUTCString()N4, J2
Date.parse(*"dateString"*)
Date.UTC(*date values*)

String — 27

constructor N4, IE4
length
prototype N4, J2

anchor(*"anchorName"*)
big()
blink()
bold()
charAt(*index*)
charCodeAt([*i*])N4, IE4
concat(*string2*)N4, IE4
fixed()
fontcolor(*#rrggbb*)
fontsize(1 to7)
fromCharCode(*n1...*)(1),N4, IE4
indexOf(*"str"* [,*i*])
italics()
lastIndexOf(*" str"* [,*i*])
link(*URL*)
localeCompare() N6, W5.5
match(*regexp*)N4, IE4
replace(*regexp*,*str*)N4, IE4
search(*regexp*)N4, IE4
slice(*i,j*)N4, IE4
small()
split(*char*)N3, (2), IE4
strike()
sub()
substr(*start*,*length*)N4, IE4
substring(*intA* , *intB*)
sup()
toLocaleLowerCase() N6, W5.5
toLocaleUpperCase() N6, W5.5
toLowerCase()
toString()N4, IE4
toUpperCase()
valueOf()N4, IE4

(1)Method of the static String object.
(2)Added behavior in N4 includes: ability to accept a regexp parameter; second parameter (limit integer) to limit the number of splits to be included; a space string parameter signifying any white space character.

Regular Expressions N4,IE4 — 42

global
ignoreCase M5,W5.5
input (1), IE5.5
lastIndex
multiline (1), W5.5
lastMatch (1), W5.5
lastParen (1), W5.5
leftContext (1),W5.5
prototype
rightContext (1)
source
$1...$9

compile(*regexp*)
exec(*" string"*)(2)
test(*"string"*)
str.match(*regexp*)
str.replace(*regexp*,*str*)
str.search(*regexp*)
str.split(*regexp*[,*limit*])

(1) Property of the static RegExp object.
(2) Returns an array with properties: index, input, [0], [1]...[n].

Boolean N3,J2 — 28

constructor N4, IE4
prototype

toString()N4, IE4
valueOf()N4, IE4

Math (1) — 28

E
LN2
LN10
LOG2E
LOG10E
PI
SQRT1_2
SQRT2

abs(*val*)
acos(*val*)
asin(*val*)
atan(*val*)
atan2(*val1* , *val2*)
ceil(*val*)
cos(*val*)
exp(*val*)
floor(*val*)
log(*val*)
max(*val1* , *val2*)
min(*val1* , *val2*)
pow(*val1* , *power*)
random()
round(*val*)
sin(*val*)
sqrt(*val*)
tan(*val*)

(1)All properties & methods are of the static Math object.

Number N3,J2 — 28

constructor N4, IE4
MAX_VALUE IE4
MIN_VALUE IE4
NaNIE4
NEGATIVE_INFINITY IE4
POSITIVE_INFINITY IE4
prototype

toExponential(*n*)N6, W5.5
toFixed(*n*)N6, IE5.5
toLocaleString() N6, W5.5,M5
toString([*radix*])N4, IE4
toPrecision(*n*)N6, W5.5
valueOf()N4, IE4

Function N3,J2 — 33

arguments
arity N4
caller
constructor N4, IE4
length
prototype

apply(*this*, *argsArray*)N6, W5.5
call(*this* [,*arg1*[,... *argN*]])N6, W5.5
toString()
valueOf()

Globals — 34

Functions

decodeURI(*" encodedURI"*)N6, W5.5
decodeURIComponent(*" encComp"*)N6, W5.5
encodeURI(*" URIString"*)N6, W5.5
encodeURIComponent(*" compString"*)N6, W5.5
escape(*" string"* [,1])
eval(*" string"*)
isFinite(*number*)N4, IE4
isNaN(*expression*)
Number(*string*)N4, IE4
parseFloat(*"string"*)
parseInt(*"string"* [,*radix*])
toString([*radix*])
unescape(*" string"*)
unwatch(*prop*)N4
watch(*prop*, *handler*)N4

Statements

// /*...*/
const N6
var

Control Statements — 31

```
if (condition ) {
    statementsIfTrue
}

if (condition ) {
    statementsIfTrue
} else {
    statementsIfFalse
}

result = condition ? expr1 : expr2

for ([init expr]; [condition ]; [update expr]) {
    statements
}

for (var in object) {          | with (object) {
    statements                 |     statements
}                              | }

while ( condition ) {          | do {
    statements                 |     statements
}                              | } while (condition )N4, IE4

switch ( expression ) {
    case labelN :
        statements
        [break]
    ...
    [default :
        statements ]
}N4, IE4

label ; N4, IE4
continue [ label]N4, IE4
break [ label]N4, IE4

try {
    statements to test
}
[catch ( errorInfo) {
    statements if exception occurs in try block
}]
[finally {
    statements to run, exceptionor not
}]N6, W5

throw valueN6, W5
```

Error N6,W5 — 31

prototype
constructor
description W5
fileName N6
lineNumber N6
message N6, IE5.5
name N6, IE5.5
number W5

toString()

© 2004 Danny Goodman (www.dannyg.com). All Rights Reserved.

JSB5

Operators 32

Comparison

==	Equals
=== N4, IE4	Strictly equals
!=	Does not equal
!== N4, IE4	Strictly does not equal
>	Is greater than
>=	Is greater than or equal to
<	Is less than
<=	Is less than or equal to

Arithmetic

+	Plus (and string concat.)
-	Minus
*	Multiply
/	Divide
%	Modulo
++	Increment
--	Decrement
-*val*	Negation

Assignment

=	Equals
+=	Add by value
-=	Subtract by value
*=	Multiply by value
/=	Divide by value
%=	Modulo by value
<<=	Left shift by value
>>=	Right shift by value
>>>=	Zero fill by value
&=	Bitwise AND by value
|=	Bitwise OR by value
^=	Bitwise XOR by value

Boolean

&&	AND
||	OR
!	NOT

Bitwise

&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Left shift
>>	Right shift
>>>	Zero fill right shift

Miscellaneous

,	Series delimiter
delete N4, IE4	Property destroyer
in N6, W5.5	Item in object
instanceof N6,W5	Instance of
new	Object creator
this	Object self-reference
typeof N3, IE3	Value type
void N3, IE3	Return no value

JavaScript and Browser Objects Quick Reference

Appendix A

JavaScript Bible, 5th Edition
by Danny Goodman

How to Use This Quick Reference

This guide contains quick reference info for the core JavaScript language, the original document object model (2 pp), the IE4+DHTML object model (4 pp), and the W3C DOM object model(4 pp) as implemented in IE5+, NN6, Mozilla, and Safari browsers. All-uppercase object names refer to HTML elements (IE4+ and W3C DOMs). The three columns in each box list the object's properties,methods, and event handlers in that order. Read the compatibility guides for each section.

Numbers in the upper right corners of object squares are chapter numbers in which the object is covered in detail.

Basic browser and operating system notation codes are as follows:

N — Netscape Navigator (including Mozilla when N6 or higher)
IE — MS Internet Explorer
W — Windows MSIE M — Macintosh MSIE

See the accompanying file for printing and collating instructions in booklet or broadside configurations.

style IE4,N6 26

Text & Fonts

color IE4, N6
font IE4, N6
fontFamily IE4, N6
fontSize IE4, N6
fontSizeAdjust M5, N6
fontStretch M5, N6
fontStyle IE4, N6
fontVariant IE4, N6
fontWeight IE4, N6
letterSpacing IE4, N6
lineBreak IE5
lineHeight IE4, N6
quotes M5, N6
rubyAlign IE5
rubyOverhang IE5
rubyPosition IE5
textAlign IE4, N6
textAlignLast IE5.5
textAutospace W5
textDecoration IE4, N6
textDecorationBlink IE-Only
textDecorationLineThrough IE-Only
textDecorationNone IE-Only
textDecorationOverline IE-Only
textDecorationUnderline IE-Only
textIndent IE4, N6
textJustify IE5
textJustifyTrim IE5
textKashidaSpace IE5.5
textOverflow W6
textShadow M5, N6
textTransform IE4, N6
textUnderlinePosition IE5.5
unicodeBidi IE5, N6
whiteSpace IE4, N6
wordBreak W5
wordSpacing M4, N6, W6
wordWrap W5.5
writingMode W5.5

Positioning

bottom IE5, N6
height IE4, N6
left IE4, N6
pixelBottom IE/W-Only
pixelHeight IE-Only
pixelLeft IE-Only
pixelRight IE/W-Only
pixelTop IE-Only
pixelWidth IE-Only
posBottom IE/W-Only
posHeight IE-Only
posLeft IE-Only
posRight IE/W-Only
posTop IE-Only
posWidth IE-Only
position IE4, N6
right IE5, N6
top IE4, N6
width IE4, N6
zIndex IE4, N6

Borders & Edges

border IE4, N6
borderBottom IE4, N6
borderLeft IE4, N6
borderRight IE4, N6
borderTop IE4, N6
borderBottomColor IE4, N6
borderLeftColor IE4, N6
borderRightColor IE4, N6
borderTopColor IE4, N6
borderBottomStyle IE4, N6
borderLeftStyle IE4, N6
borderRightStyle IE4, N6
borderTopStyle IE4, N6
borderBottomWidth IE4, N6
borderLeftWidth IE4, N6
borderRightWidth IE4, N6
borderTopWidth IE4, N6
borderColor IE4, N6
borderStyle IE4, N6
borderWidth IE4, N6
margin IE4, N6
marginBottom IE4, N6
marginLeft IE4, N6
marginRight IE4, N6
marginTop IE4, N6
outline M5, N6
outlineColor M5, N6
outlineStyle M5, N6
outlineWidth M5, N6
padding IE4, N6
paddingBottom IE4, N6
paddingLeft IE4, N6
paddingRight IE4, N6
paddingTop IE4, N6

Tables

borderCollapse M5, N6
borderSpacing M5, N6
captionSide M5, N6
emptyCells M5, N6
tableLayout IE5, N6

Lists

listStyle IE4, N6
listStyleImage IE4, N6
listStylePosition IE4, N6
listStyleType IE4, N6

Background

background IE4, N6
backgroundAttachment IE4, N6
backgroundColor IE4, N6
backgroundImage IE4, N6
gackgroundPosition IE4, N6
backgroundPositionX IE-Only
backgroundPositionY IE-Only
backgroundRepeat IE4, N6

Inline Display & Layout

clear IE4, N6
clip IE4, N6
clipBottom W5
clipLeft W5
clipRight W5
clipTop W5
content M5, N6
counterIncrement M5, N6
counterReset M5, N6
cssFloat M5, N6
cursor IE4, N6
direction IE5, N6
display IE4, N6
filter W4
layoutGrid W5
layoutGridChar W5
layoutGridLine W5
layoutGridMode W5
layoutGridType W5
markerOffset M5, N6
marks M5, N6
maxHeight M5, N6
maxWidth M5, N6
minHeight M5, N6, W6
minWidth M5, N6
MozOpacity N6
overflow IE4, N6
overflowX W5
overflowY W5
styleFloat IE-Only
verticalAlign IE4, N6
visibility IE4, N6
width IE4, N6
zoom W5.5

Printing

orphans M5, N6
widows M5, N6
page M5, N6
pageBreakAfter IE4, N6
pageBreakBefore IE4, N6
pageBreakInside M5, N6
size N6

Miscellaneous

accelerator W5
behavior W5
cssText IE4, N6
imeMode W5

Scrollbars

scrollbar3dLightColor W5.5
scrollbarArrowColor W5.5
scrollbarBaseColor W5.5
scrollbarDarkShadowColor W5.5
scrollbarFaceColor W5.5
scrollbarHighlightColor W5.5
scrollbarShadowColor W5.5
scrollbarTrackColor W5.5

Original DOM (NN2+, IE3+) Quick Reference — Page 1 of 2

© 2004 Danny Goodman (www.dannyg.com). All Rights Reserved.

JSB5

window 16

appCore N6
clientInformation IE4
clipboardData W5
closed N3, IE4
components[] N6
controllers[] N6
crypto N6
defaultStatus
dialogArguments IE4
dialogHeight W4
dialogLeft IE4
dialogTop IE4
dialogWidth W4
directories N4,(S)
document
event IE4
external W4
frameElement W5.5
frames[]
history
innerHeight N4
innerWidth N4
length N6, IE4
loading N4-Only
location
locationbar N4,(S)
menubar N4,(S)
name
navigator N6, IE4
offscreenBuffering IE4
opener IE3, N3
outerHeight N4
outerWidth N4
pageXOffset N4
pageYOffset N4
parent
personalbar N4,(S)
pkcs11 N6
prompter N6
returnValue W4,M5
screen N6, IE4
screenLeft W5
screenTop W5
screenX N6
screenY N6
scrollbars N4,(S)
scrollX N6
scrollY N6
self
sidebar N4,(S)
status
statusbar N4,(S)
toolbar N4,(S)
top
window

alert(*"msg"*)
attachEvent(*" evt",func*) IE5
back() N4
blur() N3, IE4
captureEvents(type) N4-Only
clearInterval(ID) N4, IE4
clearTimeout(ID)
close()
confirm(*"msg"*)
createPopup() W5.5
detachEvent(*" evt",func*) IE5
disableExternalCapture() N4-Only
enableExternalCapture() N4-Only
execScript(*" exprs"[,lang]*) IE4
find(["*str*"][,*case*, *bkwd*]) N4
fireEvent(*"evt"[,evtObj]*) W5.5
focus() N3, IE4
forward() N4
handleEvent(*event*) N4-Only
home() N4
moveBy(*Δx,Δy*) N4, IE4
moveTo(*x,y*) N4, IE4
navigate() IE3
open(*URL,"name","specs"*) (1),(S)
print() N4, IE5
prompt(*"msg","reply"*)
releaseEvents(*type*) N4-Only
resizeBy(*Δ x,Δy*) N4-Only, IE4
resizeTo(*width,height*) N4-Only, IE4
routeEvent(*event*) N4-Only
scroll(*x,y*) N3, IE4
scrollBy(*Δ x,Δy*) N4, IE4
scrollTo(*x,y*) N4, IE4
setActive() W5.5
setInterval(*func*, *msec* [,*args*]) N4,IE4,(2)
setTimeout(*func*, *msec* [,*args*]) (2)
showHelp() W4
showModalDialog() IE4
showModelessDialog() W5
sizeToContent() N6
stop() N4

onabort N6
onafterprint W5
onbeforeprint W5
onbeforeunload IE4
onblur N3, IE4
onchange N6
onclick N6
onclose N6
ondragdrop N4-Only
onerror N3, IE4
onfocus N3, IE4
onhelp IE4
onkeydown N6
onkeypress N6
onkeyup N6
onload
onmousedown N6
onmousemove N6
onmouseout N6
onmouseover N6
onmouseup N6
onmove N4-Only
onreset N6
onresize N4, IE4
onscroll IE4
onselect N6
onsubmit N6
onunload

(1) New window specs for all browsers: height, width, toolbar, location, directories, status, menubar, scrollbars, resizable, copyhistory. Add'l specs for N4+: alwaysLowered(S), alwaysRaised(S), dependent, hotkeys, innerHeight, innerWidth, outerHeight, outerWidth, screenX, screenY, titlebar(S), z-lock(S). Add'l specs for IE4+: channelmode, fullscreen, left, top. Add'l spec for IE5+: title.
N6 + includes left, top.
(2) Optional args parameter added to N4; 3rd parameter in IE4 is for scripting language.

document 18

alinkColor
anchors[]
applets[] N3, IE4
bgColor
cookie
domain N3, IE4
embeds[] N3, IE4
fgColor
forms[]
height N4
images[] N3, IE4
lastModified
layers[] N4-Only
linkColor
links[]
location (1)
referrer
title
URL N3, IE4
vlinkColor
width N4

captureEvents(*type*) N4-Only (None)
clear()
close()
getSelection() N4,(2)
handleEvent(*event*) N4-Only
open(*"mimetype" [,replace]*) (3)
releaseEvents(*type*) N4-Only
routeEvent(*event*) N4-Only
write(*"string"*)
writeln(*"string"*)

(1) Replaced by the URL property in Navigator 3.
(2) M4 has a document.selection property to retrieve the currently selected text.
(3) mimetype parameter new in M4, but only "text/html" type supported.

layer N4-Only 39

above
background
below
bgColor
clip.bottom
clip.height
clip.left
clip.right
clip.top
clip.width
document
left
name
pageX
pageY
parentLayer
siblingAbove
siblingBelow
src
top
visibility
zIndex

load(*"filename", y*)
moveAbove(*layerObj*)
moveBelow(*layerObj*)
moveBy(*Δ x, Δy*)
moveTo(*x, y*)
moveToAbsolute(*x*, *y*)
resizeBy(*Δ x, Δy*)
resizeTo(*width*, *height*)

onblur
onfocus
onload
onmouseout
onmouseover
onmouseup

location 17

hash
host
hostname
href
pathname
port
protocol
search

assign(*" URL"*) (None)
reload([*unconditional*]) N3,IE4
replace(*"URL"*) N3,IE4

link () 19

hash (None)
host
hostname
href
pathname
port
protocol
search
target
text N4-Only
x N4-Only
y N4-Only

onclick
ondblclick N4,(1),IE4
onmousedown N4, IE4
onmouseout N4, IE4
onmouseover
onmouseup N4, IE4

(1) Not implemented in NN4/Mac.

anchor () 19

name N4, IE4 (None) (None)
text N4-Only
x N4-Only
y N4-Only

image N3,(1),IE4 20

border (None)
complete
height
hspace
lowsrc
name
src
vspace
width
x N4-Only
y N4-Only

onabort
onerror
onload

(1) Implemented in IE3.01/Macintosh but not Windows until IE4.

area N3,IE4 20

hash (None)
host
hostname
href
pathname
port
protocol
search
target

onclick N4
onmouseout
onmouseover

history 17

current (S),(1)
length
next (S),(1)
previous (S),(1)

back()
forward()
go(*int* | *"URL"*)

(None)

(1) Available in NN4+/Mozilla with signed scripts.

© 2004 Danny Goodman (www.dannyg.com). All Rights Reserved.

JSB5

form 21

action	handleEvent(*evt*)N4	onresetN3, IE4
elements[]	reset()N3, IE4	onsubmit
encoding	submit()	
length		
method		
name		
target		

button, reset, submit 22

form	click()	onclick
name	handleEvent(*evt*)N3	onmousedown N4, IE4
type N3, IE4		onmouseup N4, IE4
value		

checkbox 22

checked	click()	onclick
defaultChecked	handleEvent(*evt*)N3	onmousedown N4, IE4
form		onmouseup N4, IE4
name		
type N3, IE4		
value		

radio 22

checked	click()	onclick
defaultChecked	handleEvent(*evt*)N3	onmousedown N4, IE4
form		onmouseup N4, IE4
length		
name		
type N3, IE4		
value		

text, textarea, password, hidden 23

defaultValue (1)	blur()	onblur
form	focus()	onchange
name	handleEvent(*evt*)N4	onfocus
type N3, IE4	select()	onkeydown N4, IE4
value (2)		onkeypress N4, IE4
		onkeyup N4, IE4
		onselect

(1) Not available for textarea object.
(2) Password value property returns empty string in NN2.

file input control N3,(1),IE4 24

form	blur()	onblur
name	focus()	onfocus
type	handleEvent(*evt*)N4	onselect
value	select()	

(1) Aso called the fileUpload object.

select 24

length	blur()N3, IE4	onblur
name	focus()N3, IE4	onchange
options[i]	handleEvent(*evt*)N4	onfocus
options[i].defaultSelected		
options[i].index		
options[i].selected		
options[i].text		
options[i].value		
selectedIndex		
typeN3, IE4		

navigator 38

appCodeName	javaEnabled() N3, M4
appMinorVersion IE4	preference(*name*[, *val*])N4,(S)
appName	taintEnabled() N3, IE4
appVersion	
browserLanguage IE4	
cookieEnabled N6, IE4	
cpuClass IE4	
language N4	
mimeTypes[] N3, M4	
onLine IE4	
oscpu N6	
platform N4, IE4	
plugins[] N3, M4	
product N6	
productSub N6	
securityPolicy N4	
systemLanguage IE4	
userAgent	
userLanguage IE4	
userProfile IE4	
vendor N6	
vendorSub N6	

mimeTypeN3,M4 38

description	(None)
enabledPlugin	
type	
suffixes	

pluginN3,M4 38

name	refresh()
filename	
description	
length	

screenN4,IE4 38

availHeight	(None)
availLeft (1)	
availTop (1)	
availWidth	
bufferDepth (2)	
colorDepth	
fontSmoothingEnabled (2)	
height	
pixelDepth	
updateInterval (2)	
width	

(1)N4+ only.
(2)WinIE4+ only.

Original DOM Compatibility Guide

Baseline browsers for this section are NN2, WinIE3/J1, and MacIE3.01. An item with no notation is compatible back to that level, and is found in all new scriptable browsers since then. Notations indicate the browser and version in which the property, method, or event handler was introduced.

Except for the window and navigator objects (whose listings cover **all** browsers and versions), new features listed in this section cover only through NN4 (and IE4 when they matched a new NN4 feature). Notations are as follows:

- N3 — New in NN3
- N4 — New in NN4
- J2 — New in IE3, JScript.dll ver.2
- IE4 — New in IE4, all OSes
- (S) — Requires signed scripts (NN)

Additional window and navigator object notations are:

- W4 — IE4+/Windows only
- W5 — IE5+/Windows only
- IE5 — IE5+ all OSes
- W5.5 — IE5.5+/Windows only
- W6 — IE6+/Windows only
- M4 — IE4+/Mac only
- M5 — IE5+/Mac only
- N6 — New in NN6 & Mozilla

Except for items marked N4-Only (which are not carried over into NN6), an item listed for an early browser is also available in the IE4+ and/or W3C DOM object models. "(None)" means that no methods or events exist for the current object through IE3 and NN4.

IE4+ DHTML DOM Quick Reference — Page 1 of 4

© 2004 Danny Goodman (www.dannyg.com). All Rights Reserved.

JSB5

All HTML Elements 15

accessKey
all[]
behaviorUrns[] W5
canHaveChildren W5
canHaveHTML W5.5
children
className
clientHeight
clientLeft
clientTop
clientWidth
contentEditable W5.5
currentStyle IE5
dataFld W4,M5
dataFormatAs W4,M5
dataSrc W4,M5
disabled
document
filters[] W4
hideFocus W5.5
id
innerHTML
innerText
isContentEditable W5.5
isDisabled W5.5
isMultiLine W5.5
isTextEdit W4
lang
language
length (1)
offsetHeight
offsetLeft
offsetParent
offsetTop
offsetWidth
outerHTML
outerText
parentElement
parentTextEdit
readyState
recordNumber W4
runtimeStyle IE5
scopeName W5
scrollHeight
scrollLeft
scrollTop
scrollWidth
sourceIndex
style
tabIndex
tagName
tagUrn W5
title
uniqueID W5

addBehavior(" *URL*") W5
applyElement(*elem*[,*type*]) W5
attachEvent(" *evt*",*func*) W5
blur()
clearAttributes() W5
click()
componentFromPoint(*x*,*y*) W5
contains(*elem*)
detachEvent(" *evt*",*func*) W5
fireEvent(" *evt*"[,*evt*]) W5.5
focus()
getAdjacentText() W5
getAttribute(" *attr*"[,*case*])
getBoundingClientRect() W5
getClientRects() W5
getExpression() W5
insertAdjacentElement() W5
insertAdjacentHTML()
insertAdjacentText()
item(*index*[,*subindex*]) (1)
mergeAttributes() W5
releaseCapture() W5
removeAttribute(" *attr*"[,*case*])
removeBehavior(*ID*) W5
removeExpression(" *prop*") W5
removeNode(*children*) W5
replaceAdjacentText(" *loc*","*txt*") W5
replaceNode(*new*) W5
scrollIntoView(*top*)
setActive() W5.5
setAttribute(" *attr*",*val*[,*case*])
setCapture(*container*) W5
setExpression(" *prop*","*expr*","*lang*") W5
swapNode(*node*) W5
tags("*tag*") (1)
urns("*URN*") W5 (1)

onactivate W5.5
onbeforecopy W5
onbeforecut W5
onbeforedeactivate W5.5
onbeforeeditfocus W5
onbeforepaste W5
onblur
onclick
oncontextmenu W5
oncontrolselect W5.5
oncopy W5
oncut W5
ondblclick
ondeactivate W5.5
ondrag W5
ondragend W5
ondragenter W5
ondragleave W5
ondragover W5
ondragstart W5
ondrop W5
onfilterchange W4
onfocus
onfocusin W6(2)
onfocusout W6(2)
onhelp W4,M5
onkeydown
onkeypress
onkeyup
onlosecapture W5
onmousedown
onmouseenter W5.5
onmouseleave W5.5
onmousemove
onmouseout
onmouseover
onmouseup
onmousewheel W6
onmove W5.5(2)
onmoveend W5.5(2)
onmovestart W5.5(2)
onpaste W5
onpropertychange W5
onreadystatechange
onresize
onresizeend W5.5
onresizestart W5.5
onselectstart

(1) Property or method of all object collections.
(2) IE content editing mode only.

popup W5.5 16

document
isOpen

hide()
show()

(None)

FRAMESET 16

border
borderColor
cols
frameBorder
frameSpacing
rows

FRAME 16

allowTransparency W6
borderColor
contentWindow W5.5
frameBorder
height, width
marginHeight W6,M5
marginWidth W6,M5
name
noResize
scrolling
src

IFRAME 16

align
allowTransparency W6
contentWindow W5.5
height, width
hspace, vspace
longdesc W6,M5
marginHeight
marginWidth
name
scrolling
src

BASE 36

href
target

BASEFONT 36

color
face
size

META 36

charset
content
httpEquiv
name
url

TITLE 36

text

document 18

activeElement
alinkColor
all
anchors[]
applets[]
bgColor
body
charset
compatMode W6
cookie
defaultCharset
designMode W5
doctype M5
domain
embeds[]
expando
fgColor
fileCreatedDate
fileModifiedDate
fileSize
forms[]
frames[]
images[]
lastModified
linkColor
links[]
location
media W5.5
mimeType W5
nameProp W6
namespaces[] W5.5
parentWindow
plugins[]
protocol
readyState
referrer
scripts[]
security W5.5
selection
styleSheets[]
title
uniqueID
URL
URLUnencoded W5.5
vlinkColor

attachEvent(" *evt*",*func*) IE5
clear()
clearAttributes() IE5
close()
createElement(" *tag*")
createEventObject([*evt*]) W5.5
createStyleSheet([" *URL*"[,*index*]])
detachEvent(" *evt*",*func*) IE5
elementFromPoint(*x*,*y*)
execCommand(" *cmd*"[,*UI*][,*arg*])
focus()
mergeAttributes(*obj*) W5
open([" *mimeType* "[,*replace*])
queryCommandEnabled(" *cmd*")
queryCommandIndterm(" *cmd*")
queryCommandState(" *cmd*")
queryCommandSupported(" *cmd*")
queryCommandText(" *cmd*")
queryCommandValue(" *cmd*")
recalc(*allFlag*) W5
releaseCapture() IE5
setActive() W5.5
write("*str*")
writeln("*str*")

onactivate W5.5
onbeforecut W5
onbeforedeactivate W5.5
onbeforeeditfocus W5
onbeforepaste W5
onclick
oncontextmenu W5
oncontrolselect W5.5
oncut
ondblclick
ondeactivate W5.5
ondrag W5
ondragend W5
ondragenter W5
ondragleave W5
ondragover W5
ondragstart W5
ondrop W5
onhelp W4,M5
onkeydown
onkeypress
onkeyup
onmousedown
onmousemove
onmouseout
onmouseover
onmouseup
onpaste
onpropertychange W5
onreadystatechange
onresizeend W5.5
onresizestart W5.5
onselectionchange W5.5
onstop IE5

BODY 18

alink
background
bgColor
bgProperties
bottomMargin
leftMargin, rightMargin
link
noWrap
scroll
scrollLeft, scrollTop
text
topMargin
vLink

createControlRange() W5
createTextRange()
doScroll([" *scrollAction* "]) W5

onAfterPrint W5
onBeforePrint W5
onScroll

© 2004 Danny Goodman (www.dannyg.com). All Rights Reserved.

JSB5

FORM 21

action	reset()	onreset
autocomplete W5	submit()	onsubmit
elements[]		
encoding		
enctype		
length		
method		
name		
target		

LABEL 35

htmlFor	(None)	(None)

BUTTON 22
INPUT (button, reset, submit, radio, checkbox)

(See original object model button, reset, submit, radio, and checkbox object listings.)

INPUT (image) 22

complete	(None)	(None)
form		
name		
src		
type		

INPUT (text, password, hidden) 23

(See original object model text, password, and hidden object listings.)

maxLength	(None)	onafterupdate W4
readOnly		onbeforeupdate W4
size		onerrorupdate W4

TEXTAREA 23

(See original object model textarea object listing.)

cols	createTextRange()	onafterupdate W4
readOnly		onbeforeupdate W4
rows		onerrorupdate W4
wrap		

SELECT 24

(See original object model select object listing.)

length	item(*index*[,*subindex*]) IE5	(None)
multiple	namedItem("*id*") W6	
size	options[i].add(*elem*[,*index*])	
value	options[i].remove()	

OPTION 24

defaultSelected	(None)	(None)
form		
selected		
text		
value		

A (anchor/link) 19

hash		
host		
hostname		
href		
Methods		
mimeType		
name		
nameProp		
pathname		
port		
protocol		
protocolLong		
rel		
rev		
search		
target		
urn		

IE4+ DHTML DOM Compatibility Guide

The baseline browser for this section is MSIE 4. An item with no notation is compatible back to that level for all OS versions. Notations as follows:

- W4 — IE4+/Windows only
- W5 — IE5+/Windows only
- IE5 — IE5+ all OSes
- W5.5 — IE5.5+/ Windows only
- W6 — IE6+/Windows only
- M5 — IE5+/Mac

All HTML element objects share items from "All HTML" box on Page 1. "(None)" means no special methods or events for the current object.

OL 37

compact		
start		
type		

UL 37

compact		
type		

LI 37

type		
value		

DL, DT, DD, DIR, MENU

compact		

IMG 20

align	(None)	onabort
alt		onerror
border		onload
complete		
dynsrc		
fileCreatedDate W4,M5		
fileModifiedDate W4,M5		
fileSize W4,M5		
fileUpdatedDate W5.5,M5		
height		
href		
hspace		
isMap		
loop		
lowsrc		
mimeType W6		
name		
nameProp W5		
protocol		
src		
start		
useMap		
vspace		
width		

AREA 20

alt	(None)	(None)
coords		
hash		
host		
hostname		
href		
noHref		
pathname		
port		
protocol		
search		
shape		
target		

MAP 20

areas[]	(None)	onscroll
name		

MARQUEE 35

behavior	start()	onbounce
bgColor	stop()	onfinish
direction		onstart
height		
hspace		
loop		
scrollAmount		
scrollDelay		
trueSpeed		
vspace		
width		

IE4+ DHTML DOM Quick Reference — Page 3 of 4

© 2004 Danny Goodman (www.dannyg.com). All Rights Reserved.

JSB5

TABLE 37

align	createCaption()	onscroll
background	createTFoot()	
bgColor	createTHead()	
border	deleteCaption()	
borderColor	deleteRow(*i*)	
borderColorDark	deleteTFoot()	
borderColorLight	deleteTHead()	
caption	firstPage() IE5	
cellPadding	insertRow(*i*)	
cells[] W5	lastPage() IE5	
cellSpacing	moveRow(*srcIndex*,*destIndex*) W5	
cols[]	nextPage() W4,M5	
datePageSize	previousPage() W4,M5	
frame	refresh()	
height		
rows[]		
rules		
tBodies[]		
tFoot		
tHead		
width		

TBODY, TFOOT, THEAD 37

align	deleteRow(*i*)	(None)
bgColor	insertRow(*i*)	
rows	moveRow(*srcIndex*,*destIndex*)	
vAlign		

TR 37

align	deleteCell(*i*)	(None)
bgColor	insertCell(*i*)	
borderColor		
borderColorDark		
borderColorLight		
cells[]		
height		
rowIndex		
sectionRowIndex		
vAlign		

TD, TH 37

align
background
bgColor
borderColor
borderColorDark
borderColorLight
cellIndex
colSpan
height
noWrap
rowSpan
vAlign
width

COL, COLGROUP 37

align
span
vAlign
width

CAPTION 37

align
vAlign

H1...H6 35

align

HR 35

align
color
noShade
size
width

BR 35

clear

FONT 35

color
face
size

IE4+ DHTML DOM Compatibility Guide

The baseline browser for this section is MSIE 4. An item with no notation is compatible back to that level for all OS versions. Notations as follows:

W4 — IE4+/Windows only
W5 — IE5+/Windows only
IE5 — IE5+ all OSes
W5.5 — IE5.5+/Windows only
M5 — IE5+/Mac

All HTML element objects share items from "All HTML" box on Page 1. "(None)" means no special methods or events for the current object.

This Quick Reference does not contain listings for Microsoft's separate XML DOM.

TextRange W4 35

boundingHeight	collapse([*start*])	(None)
boundingLeft	compareEndPoints(" *type*",*range*)	
boundingTop	duplicate()	
boundingWidth	execCommand(" *cmd*"[,*UI*[,*val*]])	
htmlText	expand("*unit*")	
offsetLeft	findText("*str*"[,*scope*,*flags*])	
offsetTop	getBookmark()	
text	getBoundingClientRect()	
	getClientRects()	
	inRange(*range*)	
	isEqual(*range*)	
	move("*unit*"[,*count*])	
	moveEnd("*unit*"[,*count*])	
	moveStart("*unit*"[,*count*])	
	moveToBookmark(" *bookmark*")	
	moveToElementText(*elem*)	
	moveToPoint(*x*,*y*)	
	parentElement()	
	pasteHTML("*HTMLText*")	
	queryCommandEnabled(" *cmd*")	
	queryCommandIndeterm(" *cmd*")	
	queryCommandState(" *cmd*")	
	queryCommandSupported(" *cmd*")	
	queryCommandText(" *cmd*")	
	queryCommandValue(" *cmd*")	
	scrollIntoView()	
	select()	
	setEndPoint(" *type*",*range*)	

TextNode IE5 35

data	appendData(*text*) M5,W6	(None)
length	deleteData(*start*, *count*) M5,W6	
nextSibling	insertData(*start*, "*text*") M5,W6	
nodeName	replaceData(*start*, *count*, "*text*") M5,W6	
nodeType	splitText(*offset*) M5,W6	
nodeValue	substringData(*start*, *count*) M5,W6	
parentNode		
previousSibling		

selection W4 35

type	clear()	(None)
typeDetail W5.5	createRange()	
	empty()	

TextRectangle W5 35

bottom	(None)	(None)
left		
right		
top		

© 2004 Danny Goodman (www.dannyg.com). All Rights Reserved.

JSB5

STYLE 26

media	(None)	(None)
type		

styleSheet 26

cssText IE5	addImport(*"URL"*[,*index*])	(None)
disabled	addRule(*"selector"*,*"styleSpec "*[,*index*])	
href	removeRule(*index*)	
id		
imports[]		
media		
owningElement		
pages[] W5.5		
parentStyleSheet		
readOnly		
rules[]		
title		
type		

currentStyle IE5; runtimeStyle W5.5 26

(See **style** object)

rule 26

readOnly	(None)	(None)
selectorText		
style		

LINK 36

charset	(None)	onload
disabled		
href		
hreflang		
media		
rel		
rev		
styleSheet W5		
target		
type		

SCRIPT 36

defer	(None)	(None)
event		
htmlFor		
src		
text		
type		

event 25

altKey
altLeft IE5.5
behaviorCookie W5.5
behaviorPart W5.5
bookmarks W4
boundElements
button
cancelBubble
clientX
clientY
contentOverflow
ctrlKey
ctrlLeft IE5.5
dataFld W4
dataTransfer W5
fromElement
keyCode
nextPage W5.5
offsetX
offsetY
propertyName W5
qualifier W4
reason W4
recordset W4
repeat W5
returnValue
saveType W5.5
screenX
screenY
shiftKey
shiftLeft IE5.5
srcElement
srcFilter W4
srcUrn W5
toElement
type
wheelData W6
x
y

APPLET 40

align	(None)	oncellchange W5
altHTML		ondataavailable W5
archive W6		ondatasetchanged W5
code		ondatasetcomplete W5
codeBase		onload
height		onrowenter W5
hspace		onrowexit W5
name		onrowsdelete W5
object		onrowsinserted W5
vspace		onscroll
width		

OBJECT 40

align	(None)	oncellchange W5
altHTML		ondataavailable W5
BaseHref		ondatasetchanged W5
classid		ondatasetcomplete W5
code		onload
codeBase		onrowenter W5
codeType		onrowexit W5
data		onrowsdelete W5
height		onrowsinserted W5
hspace		onscroll
name		
object		
type		
vspace		
width		

EMBED 40

align	(None)	onload
height		onscroll
hidden		
name		
pluginspage		
src		
units		
width		

XML W5 41

src	(None)	(None)
XMLDocument		

W3C DOM (IE5+, NN6+, Mozilla, Safari) Quick Reference — Page 1 of 4

© 2004 Danny Goodman (www.dannyg.com). All Rights Reserved.

JSB5

All HTML Elements 15

attributes[]	addEventListener(" *evt*",*func*,*capt*)(2)	onblur
childNodes[]	appendChild(*elem*)	onclick
className	blur()	oncontextmenu (1)
clientHeight (1)	click()	ondblclick
clientLeft (1)	cloneNode(*deep*)	onfocus
clientTop (1)	dispatchEvent(*evt*)(2)	onkeydown
clientWidth (1)	focus()	onkeypress
dir	getAttribute(" *attr*")	onkeyup
firstChild	getAttributeNode(" *attr*")W6	onmousedown
id	getAttributeNodeNS(" *nsuri*","*local* ")(2)	onmousemove
innerHTML (1)	getAttributeNS(" *nsuri*","*local* ")(2)	onmouseout
lang	getElementsByTagName(" *tag*")	onmouseover
lastChild	hasAttribute(" *attr*")(2)	onmouseup
length (3)	hasAttributeNS(" *nsuri*","*local* ")(2)	onresize
localName	hasAttributes() (2)	
namespaceURI	hasChildNodes()	
nextSibling	insertBefore(*new*[,*ref*])	
nodeName	isSupported(" *feature*","*ver*")(2)	
nodeType	item(*index*)(3)	
nodeValue	normalize() (2)	
offsetHeight (1)	removeAttribute(" *attr*")	
offsetLeft (1)	removeAttributeNode(*node*)(2)	
offsetParent (1)	removeChild(*node*)	
offsetTop (1)	removeEventListener(" *evt*",*func*,*capt*)(2)	
offsetWidth (1)	replaceChild(*new*,*old*)	
ownerDocument	scrollIntoView() (1)	
parentNode	setAttribute(" *attr*",*val*)	
prefix	setAttributeNode(*node*)(2)	
previousSibling		
scrollHeight, scrollLeft, scrollTop, scrollWidth (1)		
style		
tabIndex		
tagName		
title		

(1)Non-W3C item originated in IE4 Object Model, but implemented in N6+/Moz/Saf for convenience.
(2)Not implemented in IE through 6. (3)Property or method of all object collections.

FRAMESET 16

cols	(None)	(None)
rows		

FRAME 16

contentDocument (1)	(None)	(None)
contentWindow N7		
frameBorder		
longDesc M5,N6		
marginHeight, marginWidth		
name		
noResize		
scrolling		
src		

(1)Not implemented in IE through 6.

HEAD 36

profile (1)

(1)Not implemented in IE through 6.

BASE 36

href
target

BASEFONT 36

color
face
size

META 36

charset
content
httpEquiv
name
scheme M5,W6
url

TITLE 36

text

W3C DOM Compatibility Guide

Baseline browsers for this section are IE5, NN6, Mozilla 1, and Safari 1. An item with no notation is compatible with these browsers (except occasionally Safari). Some items are new to Netscape 7 (N7) and Mozilla or WinIE6 (W6). Observe footnotes for items missing from IE, and see the book's text for Safari details. All HTML element objects share items from "All HTML" box on Page 1. "(None)" means no special methods or events for the current object.

IFRAME 16

align		
contentDocument (1)		
longDesc M5,W6		
marginHeight		
marginWidth		
scrolling		
src		

(1)Not implemented in IE through 6.

document 18

alinkColor	clear()	onblur
anchors[]	close()	onclick
applets[]	createAttribute(" *name*")	ondblclick
attributes[]	createComment(" *text*")W6	onfocus
bgColor	createDocumentFragment() W6	onkeydown
body	createElement(" *tag*")	onkeypress
characterSet (1)	createEvent(" *type*")(1)	onkeyup
childNodes[]	createRange()(1)	onmousedown
compatMode W6,N7	createTextNode(" *txt*")	onmousemove
cookie	createTreeWalker(*root*, *what*, *filterFunc*, *expand*)N7,(1)	onmouseout
defaultView (1)	focus()	onmouseover
doctype (1)	getElementById(" *ID*")	onmouseup
documentElement	getElementsByName(" *name*")	onresize
domain	getElementsByTagName(" *tag*")	
embeds[]	open("*mimeType*"[,*replace*])	
fgColor	write("*str*")	
firstChild	writeln("*str*")	
forms[]		
height (1)		
images[]		
implementation (1)		
lastChild		
lastModified		
linkColor		
links[]		
location		
namespaceURI		
nextSibling		
nodeName		
nodeType		
ownerDocument (1)		
parentNode		
plugins[]		
previousSibling		
referrer		
styleSheets[]		
title		
URL		
vlinkColor		
width (1)		

(1)Not implemented in IE through 6.

BODY 18

alink	(None)	(None)
background		
bgColor		
link		
scrollLeft, scrollTop N7,(1)		
text		
vLink		

(1)Originated in IE Object Model, but implemented in N7/Mozilla for convenience.

HTML 36

version (1)	(None)	(None)

(1)Not implemented in IE through 6.

© 2004 Danny Goodman (www.dannyg.com). All Rights Reserved.

JSB5

FORM 21

acceptCharset (1)	reset()	onreset
action	submit()	onsubmit
elements[]		
encoding		
enctype		
length		
method		
name		
target		

(1)Not implemented in IE through 6.

FIELDSET, LEGEND 21

align	(None)	(None)
form		

LABEL 35

accessKey	(None)	(None)
form M5,W6		
htmlFor		

BUTTON 22
INPUT (button, reset, submit, radio, checkbox)

(See original object model button, reset, submit, radio, and checkbox object listings.)

disabled

INPUT (image) 22

disabled	(None)	(None)
form		
name		
src		
type		

INPUT (text, password, hidden) 23

(See original object model text, password, and hidden object listings.)

disabled	(None)	(None)
maxLength		
readOnly		
size		

TEXTAREA 23

(See original object model textarea object listing.)

cols	(None)	(None)
disabled		
readOnly		
rows		

SELECT 24

(See original object model select object listing.)

disabled	options[*i*].remove()	(None)
length		
multiple		
size		
value		

OPTION 24

defaultSelected	(None)	(None)
disabled		
form		
label M5,W6		
selected		
text		
value		

OPTGROUP 24

form (1)	(None)	(None)
label (1)		

(1)Not implemented in IE/Windows through 6, but is implemented in IE5/Mac.

IMG 20

align	(None)	onAbort
alt		onError
border		onLoad
complete		
height		
href		
hspace		
isMap		
longDesc (1)		
lowsrc		
lowSrc (1)		
name		
naturalHeight, naturalWidth N6		
src		
useMap		
vspace		
width		

(1)Not implemented in IE through 6.

OL 37

compact
start
type

UL 37

compact
type

DL, DT, DD, DIR, MENU

compact

LI 37

type
value

A (anchor/link) 19

charset (1)
coords (1)
hash
host
hostname
href
hreflang (1)
name
pathname
port
protocol
rel
rev
search
shape (1)
target
type (1)

(1)Not implemented in IE through 5.5.

AREA 20

alt
coords
hash
host
hostname
href
noHref
pathname
port
protocol
search
shape
target

MAP 20

areas
name

MARQUEE (1) 35

behavior	start()
direction	stop()
scrollAmount	
scrollDelay	
width	

(1)From IE, but implemented in N7/Mozilla.

W3C DOM (IE5+, NN6+, Mozilla, Safari) Quick Reference — Page 3 of 4

© 2004 Danny Goodman (www.dannyg.com). All Rights Reserved.

JSB5

TABLE 37

align	createCaption()	onscroll
bgColor	createTFoot()	
border	createTHead()	
caption	deleteCaption()	
cellPadding	deleteRow(*i*)	
cellSpacing	deleteTFoot()	
frame	deleteTHead()	
height	insertRow(*i*)	
rows[]		
rules		
summary W6		
tBodies		
tFoot		
tHead		
width		

TBODY, TFOOT, THEAD 37

align	deleteRow(*i*)	(None)
bgColor	insertRow(*i*)	
ch W6		
chOff W6		
rows		
vAlign		

TR 37

align	deleteCell(*i*)	(None)
bgColor	insertCell(*i*)	
cells[]		
ch W6		
chOff W6		
rowIndex		
sectionRowIndex		
vAlign		

TD, TH 37

abbr M5,W6
align
axis M5,W6
background
bgColor
cellIndex
ch W6
chOff W6
colSpan
headers M5,W6
height
noWrap
rowSpan
vAlign
width

COL, COLGROUP 37

align
ch W6
chOff W6
span
vAlign
width

CAPTION 37

align
vAlign

H1...H6 35

align

HR 35

align
color
noShade
size
width

BR 35

clear

FONT 35

color
face
size

BLOCKQUOTE, Q 35

cite M5,W6

Node Types

ELEMENT_NODE	1
ATTRIBUTE_NODE	2
TEXT_NODE	3
CDATA_SECTION_NODE	4
ENTITY_REFERENCE_NODE	5
ENTITY_NODE	6
PROCESSING_INSTRUCTION_NODE	7
COMMENT_NODE	8
DOCUMENT_NODE	9
DOCUMENT_TYPE_NODE	10
DOCUMENT_FRAGMENT_NODE	11
NOTATION_NODE	12

Range(1) 35

collapsed	cloneContents() N7	(None)
commonAncestorContainer	cloneRange() N7	
endContainer	collapse([*start*])	
endOffset	compareBoundaryPoints(*type*,*src*) N7	
startContainer	compareNode(*node*)	
startOffet	comparePoint(*node*, *offset*)	
	createContextualFragment(" *text*")	
	deleteContents()	
	detach()	
	extractContents() N7	
	insertNode(*node*) N7	
	intersectsNode(*node*)	
	isPointInRange(*node*, *offset*)	
	selectNode(*node*)	
	selectNodeContents(*node*)	
	setEnd(*node*,*offset*)	
	setEndAfter(*node*)	
	setEndBefore(*node*)	
	setStart(*node*,*offset*)	
	setStartAfter(*node*)	
	setStartBefore(*node*)	
	surroundContents(*node*) N7	
	toString()	

(1)Not implemented in IE through 6.

Text(1) 35

data	appendData(" *text*")	(None)
length	deleteData(*offset*,*count*)	
	insertData(*offset*,"*text*")	
	replaceData(*offset*,*count*,"*text*")	
	splitText(*offset*)	
	substringData(*offset*,*count*)	

(1)Not implemented in IE through 6.

selection(1) 35

anchorNode	addRange(*range*)	(None)
anchorOffset	clearSelection()	
focusNode	collapse(*node*,*offset*)	
focusOffset	collapseToEnd()	
isCollapsed	collapseToStart()	
rangeCount	containsNode(*node*,*recurse*)	
	deleteFromDocument()	
	extend(*node*,*offset*)	
	getRangeAt()	
	removeAllRanges()	
	removeRange(*range*)	
	selectAllChildren(*node*)	
	toString()	

(1)Not implemented in IE through 6.

© 2004 Danny Goodman (www.dannyg.com). All Rights Reserved.

JSB5

STYLE 26

media	(None)	(None)
type		

styleSheet 26

cssRules[]	deleteRule(*index*)(1)	(None)
disabled	insertRule(* *rule*,*index*)(1)	
href		
media		
ownerNode(1)		
ownerRule(1)		
parentStyleSheet		
title		
type		

(1)Not implemented in IE through 5.5.

cssRule 26

cssText (2)	(None)	(None)
parentStyleSheet (2)		
selectorText		
style		
type (1)		

(1)Not implemented in IE through 6.
(2)Not implemented in IE through 6, but implemented in IE5/Mac.

LINK 36

charset	(None)	(None)
disabled		
href		
hreflang		
media		
rel		
rev		
sheet(1)		
target		
type		

(1)Not implemented in IE through 6.

SCRIPT 36

defer	(None)	(None)
event		
htmlFor		
src		
text		
type		

W3C DOM Compatibility Guide

Baseline browsers for this section are IE5, NN6, Mozilla 1, and Safari 1. An item with no notation is compatible with these browsers (except occasionally Safari). Some items are new to Netscape 7 (N7) and Mozilla or WinIE6 (W6). Observe footnotes for items missing from IE, and see the book's text for Safari details. All HTML element objects share items from "All HTML" box on Page 1. "(None)" means no special methods or events for the current object.

event(1) 25

altKey	initEvent(* *type*,*bubble*,*cancelable*)	(None)
bubbles	initKeyEvent(* *type*, *evtArgs*)	
button	initMouseEvent(* *type*, *evtArgs*)	
cancelBubble	initMutationEvent(* *type*, *evtArgs*)	
cancelable	initUIEvent(* *type*, *evtArgs*)	
charCode	preventDefault()	
clientX	stopPropagation()	
clientY		
ctrlKey		
currentTarget		
detail		
eventPhase		
isChar		
keyCode		
layerX		
layerY		
metaKey		
originalTarget		
pageX		
pageY		
relatedTarget		
screenX		
screenY		
shiftKey		
target		
timeStamp		
type		
view		

(1)Not implemented in IE through 6.

EMBED 40

align	(None)	onload
height		onscroll
name		
width		

APPLET 40

align	
alt W6	
archive W6	
code	
codeBase	
data	
height	
hspace	
name	
type	
useMap W6	
vspace	
width	

OBJECT 40

align	
alt W6	
baseURI N7	
border W6	
code	
codeBase	
codeType	
contentDocument (1)	
height	
hspace	
name	
object	
type	
vspace	
width	

(1)Not implemented in IE through 6.

APPENDIX B

JavaScript Reserved Words

Every programming language has a built-in vocabulary of keywords that you cannot use for the names of variables and the like. Because a JavaScript function is an object that uses the function name as an identifier for the object, you cannot employ reserved words for function names either. Netscape's list of reserved words closely echoes that of the Java language; thus, many of the keywords in the list do not — at least yet — apply to JavaScript. Remember that JavaScript keywords are case-sensitive. Although you may get away with using these words in other cases, it may lead to unnecessary confusion for someone reading your scripts.

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

APPENDIX C

Answers to Tutorial Exercises

This appendix provides answers to the tutorial exercises that appear in Part II of this book (Chapters 4 through 12).

Chapter 4 Answers

1. The music jukebox (a) and temperature calculator (d) are good client-side JavaScript applications. Even though the jukebox relies on server storage of the music files, you can create a more engaging and responsive user interface of buttons, swappable images, and information from a plugin, such as Windows Media Player or QuickTime. The temperature calculator is a natural, because all processing is done instantaneously on the client, rather than having to access the server for each conversion.

   The Web site visit counter (b) that accumulates the number of different visitors to a Web site is a server-side CGI application, because the count must be updated and maintained on the server. At best, a client-side counter could keep track of the number of visits the user has made to a site and report to the user how many times he or she has been to the site. The storage requires scripting the cookie (see Chapter 16). A chat room application (c) done properly requires server facilities to open up communication channels among all users connected simultaneously. Client-side scripting by itself cannot create a live chat environment.

2. **a.** Valid, because it is one contiguous word. InterCap spelling is fine.

   b. Valid, because an underscore character is acceptable between words.

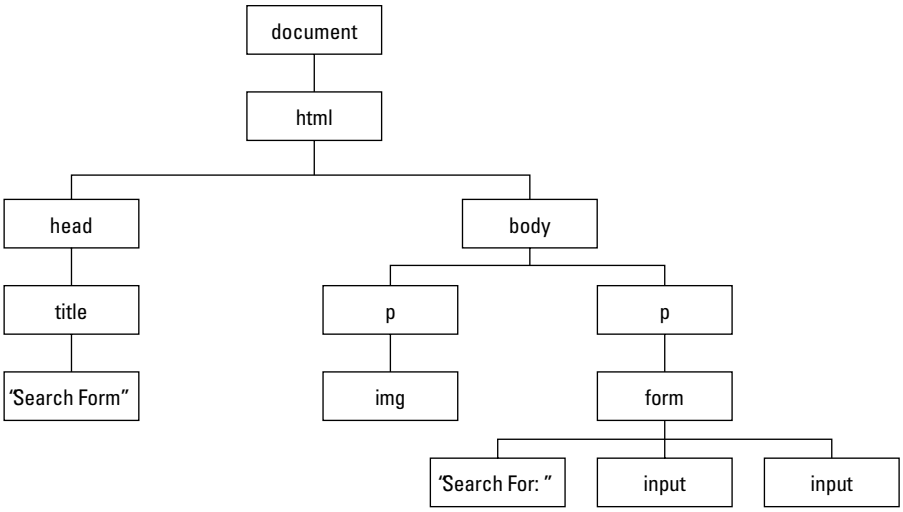
   c. Not valid, because an identifier cannot begin with a numeral.

   d. Not valid, because no spaces are allowed.

   e. Not valid, because apostrophes and most other punctuation are not allowed.

3. The diagram is shown below. The paragraph element reference is:

```
document.getElementById("formPar")
```



4. In common:

   - Both are types of nodes, derived from the basic DOM node
   - Both may be children of parent nodes that act as containers

   Different:

   - An element node is created by a tag, while a text node has no tag associated with it
   - A text node cannot be a parent to any other node, but an element node can be either a parent (branch node) or end node (leaf node)

5.
```
<input type="button" name="Hi" value="Howdy" onclick="alert('Hello to
you, too!')" />
```

Chapter 5 Answers

1.
```
<script type="text/javascript">
<!--
document.write("Hello, world.");
// -->
</script>
```

2.
```
<html>
<body>
<script type="text/javascript">
```

```
<!--
document.write("Hello, world.");
// -->
</script>
</body>
</html>
```

3.
```
<html>
<body>
<script type="text/javascript">
<!--
// write a welcome message to the world
document.write("Hello, world.");
// -->
</script>
</body>
</html>
```

4. My answer is written so that both event handlers call separate functions. You can also have each event handler invoke the `alert()` method inline. If you prefer to follow the XHTML format, include a space and forward slash character before the right angle bracket of the `input` element's tag.

```
<html>
<head>
<title>An onload script</title>
<script type="text/javascript">
<!--
function done() {
   alert("The page has finished loading.");
}
function alertUser() {
   alert("Ouch!");
}
// -->
</script>
</head>
<body onload="done()">
Here is some body text.
<form>
    <input type="button" name="oneButton" value="Press Me!"
onclick="alertUser()">
</form>
</body>
</html>
```

5.

a. The page displays two text fields.

b. The user enters text into the first field and either clicks or tabs out of the field to trigger the `onchange` event handler.

c. The function displays an all-uppercase version of one field into the other.

Chapter 6 Answers

1. **a.** Valid.

   b. Not valid. The variable needs to be a single word, such as `howMany` or `how_many`.

   c. Valid. The trailing semicolon is missing, but because it is optional for a one-line statement, browsers accept the statement as written.

   d. Not valid. The variable name cannot begin with a numeral. If the variable needs a number to help distinguish it from other similar variables, then put the numeral at the end: `address1`.

2. **a.** 4

   b. 40

   c. "4020"

   d. "Robert"

3. The functions are `parseInt()` and `parseFloat()`. Strings to be converted are passed as parameters to the functions: `parseInt(document.getElementById("entry").value)`.

4. Both text field values are strings that must be converted to numbers before they can be arithmetically added together. You can use the `parseFloat()` functions either on the variable assignment expressions (for example, `var value1 = parseFloat (document.getElementById("inputA").value)`) or in the addition expression `(document.getElementById("output").value = parseFloat(value1) + parseFloat(value2))`.

5. Concatenate means to join together two strings to become one string.

Chapter 7 Answers

1. The following answer shows the HTML markup portion in XHTML, where elements not acting as containers (notably the `input` elements) include a space and forward slash to simulate XHTML's required close tag.

```
<html>
<head>
<script type="text/javascript">
var USStates = new Array(51);
USStates[0] = "Alabama";
USStates[1] = "Alaska";
USStates[2] = "Arizona";
USStates[3] = "Arkansas";
USStates[4] = "California";
USStates[5] = "Colorado";
USStates[6] = "Connecticut";
USStates[7] = "Delaware";
USStates[8] = "District of Columbia";
USStates[9] = "Florida";
```

```
USStates[10] = "Georgia";
USStates[11] = "Hawaii";
USStates[12] = "Idaho";
USStates[13] = "Illinois";
USStates[14] = "Indiana";
USStates[15] = "Iowa";
USStates[16] = "Kansas";
USStates[17] = "Kentucky";
USStates[18] = "Louisiana";
USStates[19] = "Maine";
USStates[20] = "Maryland";
USStates[21] = "Massachusetts";
USStates[22] = "Michigan";
USStates[23] = "Minnesota";
USStates[24] = "Mississippi";
USStates[25] = "Missouri";
USStates[26] = "Montana";
USStates[27] = "Nebraska";
USStates[28] = "Nevada";
USStates[29] = "New Hampshire";
USStates[30] = "New Jersey";
USStates[31] = "New Mexico";
USStates[32] = "New York";
USStates[33] = "North Carolina";
USStates[34] = "North Dakota";
USStates[35] = "Ohio";
USStates[36] = "Oklahoma";
USStates[37] = "Oregon";
USStates[38] = "Pennsylvania";
USStates[39] = "Rhode Island";
USStates[40] = "South Carolina";
USStates[41] = "South Dakota";
USStates[42] = "Tennessee";
USStates[43] = "Texas";
USStates[44] = "Utah";
USStates[45] = "Vermont";
USStates[46] = "Virginia";
USStates[47] = "Washington";
USStates[48] = "West Virginia";
USStates[49] = "Wisconsin";
USStates[50] = "Wyoming";

var stateEntered = new Array(51);
stateEntered[0] = 1819;
stateEntered[1] = 1959;
stateEntered[2] = 1912;
stateEntered[3] = 1836;
stateEntered[4] = 1850;
stateEntered[5] = 1876;
stateEntered[6] = 1788;
stateEntered[7] = 1787;
```

```
stateEntered[8] = 0000;
stateEntered[9] = 1845;
stateEntered[10] = 1788;
stateEntered[11] = 1959;
stateEntered[12] = 1890;
stateEntered[13] = 1818;
stateEntered[14] = 1816;
stateEntered[15] = 1846;
stateEntered[16] = 1861;
stateEntered[17] = 1792;
stateEntered[18] = 1812;
stateEntered[19] = 1820;
stateEntered[20] = 1788;
stateEntered[21] = 1788;
stateEntered[22] = 1837;
stateEntered[23] = 1858;
stateEntered[24] = 1817;
stateEntered[25] = 1821;
stateEntered[26] = 1889;
stateEntered[27] = 1867;
stateEntered[28] = 1864;
stateEntered[29] = 1788;
stateEntered[30] = 1787;
stateEntered[31] = 1912;
stateEntered[32] = 1788;
stateEntered[33] = 1789;
stateEntered[34] = 1889;
stateEntered[35] = 1803;
stateEntered[36] = 1907;
stateEntered[37] = 1859;
stateEntered[38] = 1787;
stateEntered[39] = 1790;
stateEntered[40] = 1788;
stateEntered[41] = 1889;
stateEntered[42] = 1796;
stateEntered[43] = 1845;
stateEntered[44] = 1896;
stateEntered[45] = 1791;
stateEntered[46] = 1788;
stateEntered[47] = 1889;
stateEntered[48] = 1863;
stateEntered[49] = 1848;
stateEntered[50] = 1890;

function getStateDate() {
   var selectedState = document.getElementById("entry").value;
   for ( var i = 0; i < USStates.length; i++) {
      if (USStates[i] == selectedState) {
         break;
      }
   }
```

```
   alert("That state entered the Union in " + stateEntered[i] + ".");
}
</script>
</head>
<body>
<form name="entryForm">
Enter the name of a state:
<input type="text" name="entry" />
<input type="button" value="Look Up Entry Date"
onclick="getStateDate()" />
</form>
</body>
</html>
```

2. Several problems plague this function definition. Parentheses are missing from the first `if` construction's condition statement. Curly braces are missing from the second nested `if...else` construction. A mismatch of curly braces also exists for the entire function. The following is the correct form (changes and additions in boldface):

```
function format(ohmage) {
   var result;
   if (ohmage >= 10e6) {
      ohmage = ohmage / 10e6;
      result = ohmage + " Mohms";
   } else {
      if (ohmage >= 10e3) {
         ohmage = ohmage / 10e3;
         result = ohmage + " Kohms";
      } else {
         result = ohmage + " ohms";
      }
   }
   alert(result);
}
```

3. Here is one possibility:

```
for (var i = 1; i < tomatoes.length; i++) {
   if (tomatoes[i].looks == "mighty tasty") {
      break;
   }
}
var myTomato = tomatoes[i]
```

4. The new version defines a different local variable name for the dog.

```
<html>
<head>
<script type="text/javascript">
var aBoy = "Charlie Brown";    // global
var hisDog = "Snoopy";        // global
function demo() {
   var WallacesDog = "Gromit"; // local version of hisDog
```

```
   var output = WallacesDog + " does not belong to " + aBoy +
".<br>";
   document.write(output);
}
</script>
<body>
<script type="text/javascript">
demo();       // runs as document loads
document.write(hisDog + " belongs to " + aBoy + ".");
</script>
</body>
</html>
```

5. The application uses three parallel arrays and is structured very much like the solution to question 1. Learn to reuse code whenever you can.

```
<html>
<head>
<script type="text/javascript">
var planets = new Array(4);
planets[0] = "Mercury";
planets[1] = "Venus";
planets[2] = "Earth";
planets[3] = "Mars";

var distance = new Array(4);
distance[0] = "36 million miles";
distance[1] = "67 million miles";
distance[2] = "93 million miles";
distance[3] = "141 million miles";

var diameter = new Array(4);
diameter[0] = "3100 miles";
diameter[1] = "7700 miles";
diameter[2] = "7920 miles";
diameter[3] = "4200 miles";

function getPlanetData() {
   var selectedPlanet = document.getElementById("entry").value;
   for ( var i = 0; i < planets.length; i++) {
      if (planets[i] == selectedPlanet) {
         break;
      }
   }
   var msg = planets[i] + " is " + distance[i];
   msg += " from the Sun and ";
   msg += diameter[i] + " in diameter.";
   document.getElementById("output").value = msg;
}
</script>
</head>
<body>
```

```
<form name="entryForm">
Enter the name of a planet:
<input type="text" name="entry" id="entry" />
<input type="button" value="Look Up a Planet"
onclick="getPlanetData()" />
<br />
<input type="text" size="70" name="output" id="output" />
</body>
</html>
```

Chapter 8 Answers

1. **a.** Close, but no cigar. Array references are always plural:

   ```
   window.document.forms[0]
   ```

   b. Not valid: `self` refers to a window and `entryForm` must refer to a form. Where's the `document`? It should be `self.document.entryForm.entryField.value`.

   c. Valid. This reference points to the `name` property of the third form in the document.

   d. Not valid. The uppercase "D" in the method name is incorrect.

   e. Valid, assuming that `newWindow` is a variable holding a reference to a subwindow.

2. `window.status = "Welcome to my Web page."`

3. `document.write("<h1>Welcome to my Web page.</h1>")`

4. A script in the Body portion invokes a function that returns the text entered in a `prompt()` dialog box.

   ```
   <html>
   <head>
   <script type="text/javascript">
   function askName() {
      var name = prompt("What is your name, please?","");
      return name;
   }
   </script>
   </head>
   <body>
   <script type="text/javascript">
   document.write("Welcome to my web page, " + askName() + ".");
   </script>
   </body>
   </html>
   ```

5. The URL can be derived from the `href` property of the `location` object.

   ```
   <html>
   <head>
   <script type="text/javascript">
   ```

```
function showLocation() {
   alert("This page is at: " + location.href);
}
</script>
</head>
<body onload="showLocation()">
Blah, blah, blah.
</body>
</html>
```

Chapter 9 Answers

1. For Listing 9-1, pass the text input element object because that's the only object involved in the entire transaction.

```
<html>
<head>
<title>Text Object value Property</title>
<script type="text/javascript">
function upperMe(field) {
   field.value = field.value.toUpperCase();
}
</script>
</head>
<body>
<form onsubmit="return false">
<input type="text" name="convertor" value="sample"
onchange="upperMe(this)">
</form>
</body>
</html>
```

For Listing 9-2, the button invokes a function that communicates with a different element in the form. Pass the form object.

```
<html>
<head>
<title>Checkbox Inspector</title>
<script type="text/javascript">
function inspectBox(form) {
   if (form.checkThis.checked) {
      alert("The box is checked.");
   } else {
      alert("The box is not checked at the moment.");
   }
}
</script>
</head>
<body>
<form>
<input type="checkbox" name="checkThis">Check here<br>
```

```
<input type="button" value="Inspect Box"
onclick="inspectBox(this.form)">
</form>
</body>
</html>
```

For Listing 9-3, again the button invokes a function that looks at other elements in the form. Pass the form object.

```
<html>
<head>
<title>Extracting Highlighted Radio Button</title>
<script type="text/javascript">
function fullName(form) {
   for (var i = 0; i < form.stooges.length; i++) {
      if (form.stooges[i].checked) {
         break;
      }
   }
   alert("You chose " + form.stooges[i].value + ".");
}
</script>
</head>

<body>
<form>
<p>Select your favorite Stooge:
<input type="radio" name="stooges" value="Moe Howard" checked>Moe
<input type="radio" name="stooges" value="Larry Fine"> Larry
<input type="radio" name="stooges" value="Curly Howard"> Curly<br>
<input type="button" name="Viewer" value="View Full Name..."
onclick="fullName(this.form)"></p>
</form>
</body>
</html>
```

For Listing 9-4, all action is triggered by and confined to the `select` object. Pass only that object to the function.

```
<html>
<head>
<title>Select Navigation</title>
<script type="text/javascript">
function goThere(list) {
   location = list.options[list.selectedIndex].value;
}
</script>
</head>

<body>
<form>
Choose a place to go:
```

```
<select name="urlList" onchange="goThere(this)">
   <option selected value="index.html">Home Page
   <option value="store.html">Shop Our Store
   <option value="policies">Shipping Policies
   <option value="http://www.google.com">Search the Web
</select>
</form>
</body>
</html>
```

2. Here are the most likely ways to reference the text box object:

```
document.getElementById("email")
document.forms[0].elements[0]
document.forms["subscription"].elements[0]
document.subscription.elements[0]
document.forms[0].elements["email"]
document.forms["subscription"].elements["email"]
document.subscription.elements["email"]
document.forms[0].email
document.forms["subscription"].email
document.subscription.email
```

The reference `document.all.email` (or any reference starting with `document.all`) works only in Internet Explorer and other browsers that emulate IE, but not in Mozilla or Safari, as requested. Other valid references may include the W3C DOM `getElementsByTagName()` method. Since the question indicates that there is only one form on the page, the text box is the first `input` element in the page, indicating that `document.body.getElementsByTagName("input")[0]` would be valid for this page.

3. The `this` keyword refers to the text input object, so that `this.value` refers to the `value` property of that object.

```
function showText(txt) {
   alert(txt);
}
```

4.
```
document.accessories.acc1.value = "Leather Carrying Case";
```

```
document.forms[1].acc1.value = "Leather Carrying Case";
```

5. The `select` object invokes a function that does the job.

```
<html>
<head>
<title>Color Changer</title>
<script type="text/javascript">
function setColor(list) {
   var newColor = list.options[list.selectedIndex].value;
   document.bgColor = newColor;
}
</script>
</head>
```

```
<body>
<form>
Select a background color:
<select onchange="setColor(this)">
<option value="red">Stop
<option value="yellow">Caution
<option value="green">Go
</select>
</form>
</body>
</html>
```

Chapter 10 Answers

1. Use *string*.indexOf() to see if the field contains the "@" symbol.

```
<html>
<head>
<title>E-mail @ Validator</title>
<script type="text/javascript">
function checkAddress(form) {
   if (form.email.value.indexOf("@") == -1) {
      alert("Check the e-mail address for accuracy.");
      return false;
   }
   return true;
}
</script>
</head>

<body>
<form onsubmit="return checkAddress(this)">
Enter your e-mail address:
<input type="text" name="email" size="30"><br>
<input type="submit">
</form>
</body>
</html>
```

2. Remember that the substring goes up to, but does not include, the index of the second parameter. Spaces count as characters.

```
myString.substring(0,3)    // result = "Int"
myString.substring(11,17)  // result = "plorer"
myString.substring(5,12)   // result = "net Exp"
```

3. The missing for loop is in boldface. You could also use the increment operator on the count variable (++count) to add 1 to it for each letter "e."

```
function countE(form) {
   var count = 0;
   var inputString = form.mainstring.value.toLowerCase();
```

```
   for (var i = 0; i < inputString.length; i++) {
      if (inputString.charAt(i) == "e") {
         count += 1;
      }
   }
   var msg = "The string has " + count;
   msg += " instances of the letter e.";
   alert(msg);
}
```

4. The formula for the random throw of one die is in the chapter.

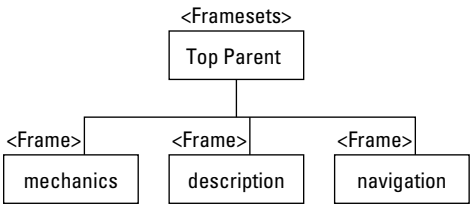
```
<html>
<head>
<title>Roll the Dice</title>
<script type="text/javascript">
function roll(form) {
   form.die1.value = Math.floor(Math.random() * 6) + 1
   form.die2.value = Math.floor(Math.random() * 6) + 1
}
</script>
</head>

<body>
<form>
<input type="text" name="die1" size="2">
<input type="text" name="die2" size="2"><br>
<input type="button" value="Roll the Dice" onclick="roll(this.form)">
</form>
</body>
</html>
```

5. If you used the `Math.round()` method in your calculations, that is fine for your current exposure to the `Math` object. Another method, `Math.ceil()`, may be more valuable because it rounds up any fractional value.

```
<html>
<head>
<title>Waiting for Santa</title>
<script type="text/javascript">
function daysToXMAS() {
   var oneDay = 1000 * 60 * 60 * 24;
   var today = new Date();
   var XMAS = new Date("December 25, 2001");
   var diff = XMAS.getTime() - today.getTime();
   return Math.ceil(diff/oneDay);
}
</script>
</head>

<body>
<script type="text/javascript">
```

```
document.write(daysToXMAS() + " days until Christmas.");
</script>
</body>
</html>
```

Chapter 11 Answers

1. `onload="parent.currCourse = 'history101'"`
2. 

3. All three frames are siblings, so references include the parent.

```
parent.mechanics.location.href = "french201M.html";
parent.description.location.href = "french201D.html";
```

4. A script in one of the documents is attempting to reference the `selector` object in one of the frames but the document has not fully loaded, causing the object to not yet be in the browser's object model. Rearrange the script so that it fires in response to the `onload` event handler of the framesetting document.
5. From the subwindow, the `opener` property refers back to the frame containing the `window.open()` method. To extend the reference to the frame's parent, the reference includes both pieces: `opener.parent.location`.

Chapter 12 Answers

1. As the document loads, the `<img>` tag creates a document image object. A memory image object is created with the `new Image()` constructor. Both objects have the same properties, and assigning a URL to the `src` property of a memory object loads the image into the browser's image cache.
2.

```
var janeImg = new Image(100,120);

janeImg.src = "jane.jpg";
```

3. `document.images["people"].src = janeImg.src;`
4. Surround `<img>` tags with link (`<a>`) tags, and use the link's `onclick`, `onmouseover`, and `onmouseout` event handlers. Set the image's `border` attribute to zero if you don't want the link highlight to appear around the image.
5. The following works in all W3C DOM-compatible browsers. The order of the first two statements may be swapped without affecting the script.

```
var newElem = document.createElement("a");
var newText = document.createTextNode("Next Page");
newElem.href = "page4.html";
newElem.appendChild(newText);
document.getElementById("forwardLink").appendChild(newElem);
```

✦ ✦ ✦

APPENDIX D

JavaScript and DOM Internet Resources

As an online technology, JavaScript has plenty of support online for scripters. Items recommended here were taken as a snapshot of Internet offerings in late 2003. But beware! Sites change. URLs change. Be prepared to hunt around for these items if the information provided here becomes out-of-date by the time you read this.

Support and Updates for This Book

The most up-to-date list of errata and other notes of interest pertaining to this edition of the *JavaScript Bible* can be found at the official Support Center, located at:

`http://www.dannyg.com/support/index.html`

If you are experiencing difficulty with the example listings in this book, first check with the Support Center to see if your question has been answered. As mentioned earlier, you are encouraged to enter the tutorial listings yourself to get used to typing JavaScript (and HTML) code. If, after copying the examples from Part II, you can't make something work (and a fix hasn't already been posted to the Support Center), send the file you've typed to me via e-mail, along with a description of what's not working for you. Also tell me the browser version and operating system that you're using. My e-mail address is `dannyg@dannyg.com`. Regretfully, I am unable to answer general questions about JavaScript or how to apply examples from the book to your own projects.

Newsgroups

The best places to get quick answers to your pressing questions are online newsgroups. Here are the top JavaScript-related newsgroups:

On most news servers:

`comp.lang.javascript`

On news://msnews.microsoft.com:

```
microsoft.public.scripting.jscript
microsoft.public.windows.inetexplorer.ie5.programming.dhtml
microsoft.public.windows.inetexplorer.ie5.programming.dhtml.scripting
microsoft.public.inetsdk.programming.scripting.jscript
```

On news://secnews.netscape.com:

```
netscape.public.mozilla.dom
netscape.public.mozilla.jseng
```

Before you post a question to a newsgroup, however, read about FAQs in the following section and also use the extremely valuable newsgroup archive search facility of Google Groups. Visit the Google Groups search page at:

```
http://groups.google.com/
```

Enter the keyword or phrase into the top text box, but then also try to narrow your search by limiting the newsgroup(s) to search. For example, if you have a question about weird behavior you are experiencing with the `borderCollapse` style property in IE, enter `borderCollapse` into the search field, and then try narrowing the search to a specific newsgroup (forum) such as `comp.lang.javascript`.

If you post a question to a newsgroup, you will most likely get a quick and intelligent response if you also provide either some sample code that's giving you a problem, or a link to a temporary file on your server that others can check out. Visualizing a problem you've spent days on is very hard for others. Be as specific as possible, including the browser(s) on which the code must run and the nature of the problem.

FAQs

One situation that arises with a popular and accessible technology, such as JavaScript and DHTML authoring, is that the same questions get asked over and over, as newcomers arrive on the scene daily. Rather than invoke the ire of newsgroup users, look through existing FAQ files to see if your concern has already been raised and answered. Here are some of the best JavaScript FAQ sites:

```
javascript.faqts.com
developer.irt.org/script/script.htm
```

For less-frequently asked questions — but previously asked and answered in a public form — use the Google Groups search, described earlier in this appendix.

Online Documentation

Locations of Web sites that dispense official documentation for one browser or another are extremely fluid. Therefore, the following information contains links only to top-level areas of appropriate Web sites, along with tips on what to look for after you are at the site.

Microsoft has condensed its developer documentation into a massive site called MSDN (Microsoft Developer Network). The place to begin is:

```
http://msdn.microsoft.com/workshop/
```

This page is the portal to many technologies, but the one most applicable to JavaScript and client-side scripting is one labeled "Web Development." Within that heading you'll find a section named "Scripting." Here you'll find plenty of documentation and technical articles for Microsoft scripting technologies, including JScript (Microsoft's flavor of JavaScript).

For Netscape browser technologies, start at:

```
http://developer.netscape.com/docs/
```

You can also find some interesting future-oriented developer documentation at:

```
http://www.mozilla.org/docs
```

Finally, you can read the industry standards for HTML, CSS, and ECMAScript technologies online. Be aware that these documents are primarily intended for developers of tools that we use — browsers, WYSIWYG editors, and so forth — to direct them on how their products should respond to tags, stylesheets, scripts, and so on. Reading these documents has frequently been cited as a cure for insomnia.

```
http://www.ecma-international.org/publications/standards/ECMA-262.HTM
http://www.w3.org/TR/html4
http://www.w3.org/Markup
http://www.w3.org/TR/REC-CSS2
http://www.w3.org/DOM/
```

Please note that just because a particular item is described in an industry standard doesn't mean that it is implemented in any or all browsers. In the real world, we must develop for the way the technologies are actually implemented in browsers.

World Wide Web

The number of Web sites devoted to JavaScript tips and tricks is mind-boggling. Many sites come and go in the middle of the night, leaving no trace of their former existence. If you are looking for more example code for applications not covered in this book, perhaps the best place to begin your journey is through the traditional search engines. Narrowing your search through careful keyword choice is vital. In addition to the Netscape and (heavily Windows-oriented) Microsoft developer Web sites (plus numerous online articles of mine listed at `http://www.dannyg.com/pubs/index.html`), a couple other venerable sites are:

```
http://builder.com
http://www.webreference.com
```

These sites are by no means the only worthwhile JavaScript and DHTML destinations on the Web. Sometimes having too many sources is as terrifying as having not enough. The links and newsgroups described in this appendix should take you a long way.

✦ ✦ ✦

APPENDIX E

What's on the CD-ROM

The accompanying Windows–Macintosh CD-ROM contains additional chapters including many more JavaScript examples, electronic versions of the Quick Reference shown in Appendix A for printing in several formats, a complete, searchable version of the entire book, and the Adobe Acrobat Reader.

System Requirements

To derive the most benefit from the example listings, you should have both Netscape Navigator 7 (or later) and Internet Explorer 6 (or later) installed on your computer. Although many scripts run in these and other browsers, several scripts demonstrate features that are available on only a limited range of browsers. To write scripts, you can use a simple text editor, word processor, or dedicated HTML editor.

To use the Adobe Acrobat Reader (version 6.0), you need the following:

- For Windows 98, Windows ME, Windows NT4.0 (with SP6), Windows 2000 (with SP2), Windows XP Pro/Home, or Windows XP Table PC Edition, you should be using a Pentium computer with 32MB of RAM and 60MB of hard disk space.
- Macintosh users require a PowerPC, OS X v10.2.2 or later, at least 32MB of RAM, and 70MB of disk space.

Disc Contents

When you view the contents of the CD-ROM, you will see files tailored for your operating system. The contents include the following items.

JavaScript listings for text editors

Starting with Part III of the book, almost all example listings are on the CD-ROM in the form of complete HTML files, which you can load into a browser to see the language item in operation. A directory called Listings contains the example files, with nested folders named for each chapter. Each HTML file's name is keyed to the listing number in the book. For example, the file for Listing 15-1 is named `lst15-01.htm`. Note that no listing files are provided for the tutorial chapters of Part II, because you are encouraged to enter HTML and scripting code manually.

For your convenience, the `_index.html` file in the `Listings` folder provides a front-end table of contents to the HTML files for the book's program listings. Open that file from your browser whenever you want to access the program listing files. If you intend to access that index page frequently, you can bookmark it in your browser(s). Using the index file to access the listing files can be very important in some cases, because several individual files must be opened within their associated framesets to work properly. Accessing the files through the `_index.html` file ensures that you open the frameset. The `_index.html` file also shows browser compatibility ratings for all the listings. This saves you time from opening listings that are not intended to run on your browser. To examine and modify the HTML source files, open them from your favorite text editor program (for Windows editors, be sure to specify the `.htm` file extension in the Open File dialog box).

You can open all example files directly from the CD-ROM, but if you copy them to your hard drive, access is faster and you will be able to experiment with modifying the files more readily. Copy the folder named `Listings` from the CD-ROM to any location on your hard drive.

JavaScript and Browser Object Quick Reference from Appendix A (Adobe Acrobat format)

If you like the quick reference in Appendix A, you can print it out on two sides of seven sheets of paper with the help of the Adobe Acrobat Reader, included with the CD-ROM. You can choose from four versions, depending on the type of layout you prefer and kind of printer you use. Open the `JS Object Reference` folder and read the file `Choose a Version.txt` for details on each Quick Reference printout type. Then read the "howtoprt" file for your desired version to understand details about Page Setup settings, paper feed options, and assembly after printing.

Adobe Acrobat Reader

The Adobe Acrobat Reader is a helpful program that enables you to view the reference example sections for Parts III and IV of the book, the Quick Reference from Appendix A, and the searchable version of this book, all of which are in `.pdf` format on the CD-ROM. To install and run Adobe Acrobat Reader, follow these steps:

For Windows

1. Start Windows Explorer and then open the Acrobat folder on the CD-ROM.
2. In the Acrobat folder, double-click the lone executable file and follow the instructions presented on-screen for installing Adobe Acrobat Reader.

For Macintosh

1. Open the Acrobat folder on the CD-ROM.
2. In the Acrobat folder, double-click the Adobe Acrobat Reader disk image icon (AdbeRdXX_enu_full.dmg), this will mount the disk image on your computer, then open the mounted image and copy the Adobe Reader folder to the Applications directory of your computer.

Acrobat version of book with topical references

In many places throughout the reference chapters of Parts III and IV, you see notations directing you to the CD-ROM for a particular topic being discussed. All of these topics are located in the chapters as they appear in complete Adobe Acrobat form on the CD-ROM. A single PDF file is located on the CD-ROM, and it serves as an electronic version of the entire book, complete with full topics that are listed as CD-ROM references in the printed book. For the fastest access to these topics, copy the entire PDF file for the book to your hard disk.

Like any Acrobat document, the PDF version of the book is searchable. Current versions of Acrobat should automatically load the Acrobat index file (with the `.pdx` extension) to supply indexed search capabilities (which is much faster than Acrobat's Find command).

To begin an actual search, click the Search icon (binoculars in front of a sheet of paper). Enter the text for which you're searching. To access the index and search facilities in future sessions, the CD-ROM must be in your CD-ROM drive; unless, of course, you've copied both the `.pdx` and `.pdf` files to your hard drive.

Troubleshooting

If you have difficulty installing or using the CD-ROM programs, try the following solutions:

- **Turn off any anti-virus software that you may have running.** Installers sometimes mimic virus activity and can make your computer incorrectly believe that a virus is infecting it. (Be sure to turn the antivirus software back on later.)
- **Close all running programs.** The more programs you're running, the less memory is available to other programs. Installers also typically update files and programs; if you keep other programs running, installation may not work properly.

If you still have trouble with the CD-ROM, please call the Wiley Product Technical Support phone number: (800) 762-2974. Outside the United States, call 1(317) 572-3994. You can also contact Wiley Product Technical Support through the Internet at: `http://www.wiley.com/techsupport`.Wiley Publishing will provide technical support only for installation and other general quality control items; for technical support on the applications themselves, consult the program's vendor or author.

To place additional orders or to request information about other Wiley products, please call (800) 225-5945.

✦ ✦ ✦

PART

VI

Bonus Chapters

Chapter 36
HTML Directive Objects

Chapter 37
Table and List Objects

Chapter 38
The Navigator and Other Environment Objects

Chapter 39
Positioned Objects

Chapter 40
Embedded Objects

Chapter 41
XML Objects

Chapter 42
The Regular Expression and RegExp Objects

Chapter 43
Data-Entry Validation

Chapter 44
Scripting Java Applets and Plug-Ins

Chapter 45
Debugging Scripts

Chapter 46
Security and Netscape Signed Scripts

Chapter 47
Cross-Browser Dynamic HTML Issues

Chapter 48
Internet Explorer Behaviors

Chapter 49
Application: Tables and Calendars

Chapter 50
Application: A Lookup Table

Chapter 51
Application: A "Poor Man's" Order Form

Chapter 52
Application: Outline-Style Table of Contents

Chapter 53
Application: Calculations and Graphics

Chapter 54
Application: Intelligent "Updated" Flags

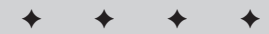
Chapter 55
Application: Decision Helper

Chapter 56
Application: Cross-Browser DHTML Map Puzzle

Chapter 57
Application: Transforming XML Data

CHAPTER 36

HTML Directive Objects

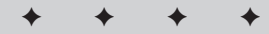


In This Chapter

Accessing non-displayed element objects

Linking operating-system specific stylesheet definitions

HTML, `head`, `link`, `title`, `meta`, `base`, and `script` elements



Thanks to the modern browser's desire to expose all HTML elements to the document object model, we can now (in IE4+ and W3C DOM browsers) access a variety of objects that represent many HTML elements that are normally invisible to the human viewer of a page. These elements are called *directive elements* because they predominantly contain instructions for the browser — instructions that direct the browser to locate associated content on the page, link in external specifications, treat content as executable script statements, and more.

As you browse through the objects of this chapter, you may wonder why they have so many properties that normally indicate that the elements occupy space on the rendered page. After all, how can a `meta` element have dimension or position on the page when it has no renderable content? The reason is that modern browsers internally employ some form of object-oriented behavior that lets all HTML elements — rendered or not — inherit the same set of properties, methods, and event handlers that any generic element has (see Chapter 15). The logical flaw is that unrendered elements can have properties and methods that don't genuinely apply to them. In such cases, their property values may be zero, an empty string, or an empty array. Yet the properties and methods exist in the objects just the same. Therefore, despite the large number of objects covered in this chapter, there are relatively few properties and methods that are not shared already with all HTML elements (as covered in Chapter 15).

HTML Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`version`		

Syntax

Accessing `html` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.body.parentNode.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
(IE4+)     [window.]document.body.parentElement.property | method([parameters])
```

About this object

The `html` element is the big wrapper around all other elements of the page. In the object tree, the `html` element sits between the all-encompassing `document` object and the element's most common children, the `head` and `body` elements. Other than one deprecated property (`version`), the `html` element object offers nothing of importance to the scripter — with one possible exception. When your script needs to use methods on the child nodes of the `html` element, you must invoke most of those methods from the point of view of the `html` element. Therefore, you should know how to create a reference to the `html` element object (shown in the preceding "Syntax" section) just in case you need it.

Property

version

Value: String. Read-Only
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The `version` property is an artifact of an "ancient" way an HTML document used to specify the HTML version of its content. These days, the preferred way to declare the HTML version for a document is through a Document Type Declaration (DTD) statement that precedes the `<html>` tag. An example of a modern DTD statement that accommodates HTML 4 plus deprecated elements and attributes as well as frameset support is

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Frameset//EN"
                  "http://www.w3.org/TR/REC-html40/frameset.dtd">
```

See `http://www.w3.org/TR/REC-html40/struct/global.html#h-7.2` for several other possibilities. A DTD statement does not affect the `version` property of an html element object.

head Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`profile`		

Syntax

Accessing `head` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

About this object

The purpose of the `head` element is primarily to act as a container for most of the other HTML directive elements. Other than as a reference point to the child elements nested within, the `head` element object rarely comes into play when scripting a document.

Property

`profile`

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `profile` property is the script version of the optional `profile` attribute of a `head` element. While the attribute and property are supported in many browsers, they are not used in practice yet. You can find details about the attribute at `http://www.w3.org/TR/REC-html40/struct/global.html#profiles`.

Related Items: `meta` element object.

base Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`href`		
`target`		

Syntax

Accessing `base` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

About this object

The base element enables the page author to specify a default server directory and/or link target for the entire page. If you omit the base element from the HTML, browsers use the current page's path as the base URL and the current window or frame as the default target. Occasionally, a page generated entirely by way of document.write() has difficulty establishing the same base URL as the document that generates the content, particularly if the primary page is written out by a server script (in Perl or in another language). Including a <base> tag in the dynamically written new page solves the problem; the new page can fetch images or other external elements via relative URLs within the page.

The two distinctive properties of the base element object are rarely scripted, if ever.

Properties

href

Value: URL string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The href property is generally an absolute URL to the directory you wish to declare as the default directory for the page. Even though browsers automatically set the base href to the document's own directory, this object and property do not have any values unless you explicitly set them in a <base> tag. In IE, changing this property after a page loads causes the page to re-resolve all relative URLs on the page to the new base href. Therefore, if images have relative URLs assigned to their src properties (either by way of the tag attribute or script), a change to the base element's href property forces the browser to look for those same relative URLs in the new directory. If the files aren't there, the images show up broken on the page.

target

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The target property governs the default window or frame that is to receive any content coming from a server in response to a click on a link or any other element that has its own target attribute. Valid values include the name of any frame (as assigned to the name attribute of the <frame> tag) or window (as defined by the second attribute of the window.open() method). You can also assign standard HTML targets (_blank, _parent, _self, and _top) to this property as strings.

basefont Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
color		
face		
size		

Syntax

Accessing `basefont` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

About this object

The `basefont` element enables authors to define a font face, size, and color for an entire section of an HTML document — or the entire document. Although page authors still frequently use the `basefont` element, font control in modern browsers should fall in the hands of stylesheets. (The element is deprecated in HTML 4.0.) The paradox of this is that the `basefont` element is accessible as a scriptable object only in browsers that support stylesheets. Even so, I recommend avoiding dynamic font changes by way of the `basefont` element and use scripts to control stylesheets instead.

The `basefont` element has no end tag, so IE's `outerHTML` property consists of all HTML in the document starting with the element itself.

The three distinctive properties of the `basefont` element object are rarely, if ever, scripted.

Properties

color
face
size

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

These three properties define the characteristics of font rendering for all content following the element's tag in the document. Color specifications can be hexadecimal triplets or Netscape color names (a list is available at `http://developer.netscape.com/docs/manuals/htmlguid/colortab.htm`). Font faces can include a list of comma-separated font face names. And because this is HTML as opposed to stylesheet fonts, the size property is in terms of the 1 through 7 scale of font sizes. You can also use relative sizes (for example, +1).

isindex Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`alt`		
`border`		
`checked`		

Continued

isindex

Properties	*Methods*	*Event Handlers*
complete		
dynsrc		
form		
height		
hspace		
indeterminate		
loop		
lowsrc		
maxLength		
name		
prompt		
readOnly		
size		
start		
status		
value		
vrml		
vspace		
width		

Syntax

Accessing `isindex` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

About this object

The `isindex` element is a holdover from the early beginnings of HTML. It offered the first text input field prior to the addition of `form` and `input` elements to the HTML specification. IE treats this element as if it were an `input` element, so `isindex` takes on all possible `input` element properties (including those of buttons). This element is deprecated in HTML 4.0 and should not be part of your development vocabulary. Use forms and genuine `input` elements instead (see Chapters 21–24).

link Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	Methods	Event Handlers
charset		onload
disabled		
href		
hreflang		
media		
rel		
rev		
sheet		
styleSheet		
target		
type		

Syntax

Accessing `link` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

About this object

The `link` element (not to be confused with the `a` element that is often referred to as a "link" element when it contains an `href` attribute pointing to another document) has many potential uses in pointing to external documents that relate to the current document. Its most common usage today is for linking an external stylesheet specification to the document. In fact, it's not uncommon for sophisticated site designs to use `document.write()` to generate the `<link>` tag so that operating-system–specific stylesheets are applied to the page. In the following code fragment (which goes inside a document's `head` element), the page loads a Macintosh-specific stylesheet when the page is running on a Macintosh; otherwise, it loads a Windows-specific stylesheet:

```
<script type="text/javascript">
var isMac = navigator.userAgent.indexOf("Mac") != -1;
var linkTagStart = "<link rel='stylesheet' type='text/css' href='";
var linkTagEnd = ".css'>";
if (isMac) {
    document.write(linkTagStart + "mac" + linkTagEnd;
} else {
    document.write(linkTagStart + "windows" + linkTagEnd;
}
</script>
```

Although it may appear that the `link` element can load a variety of content into a page, do not use it for multimedia (in which case you should use the `embed` or `object` elements) or external HTML (where you should use an `iframe` element).

Many of the properties of the `link` element object are script representations of HTML 4.0 attributes for the element. However, browsers don't take full advantage of the possibilities available from the `link` element yet. (For example, a browser can provide arrows to the previous and next documents in a series, as specified by the `rev` and `rel` attributes. But so far, no browser implements this.) Properties unique to this object offer scripted access (in various browser versions) to attribute values of the `link` element. Therefore, this chapter does not spend a lot of time on properties that are not in current use.

Properties

charset

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `charset` property advises the browser about the character encoding of the content that will arrive from the external document (assuming you also have the `href` attribute set). Values for this property must match the encoding naming conventions defined in an industry standard registry (`http://www.iana.org/assignments/character-sets`).

disabled

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

By changing the `disabled` property (default is `false`), you can turn externally linked content on and off. For example, you can define two different stylesheet links in a document that has two `<link>` tags with one's `disabled` attribute set. You can switch between the two stylesheets by setting the `disabled` property of one to `true` and the other to `false`.

href

Value: String. See Text
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Another way to swap stylesheets is to modify the value of a single `link` element object's `href` property (although the property is read-only in MacIE4+). The property's value is a URL string.

hrefLang

Value: String. Read/Write
Compatibility: WinIE6+, MacIE6+, NN6+, Moz1+, Safari1+

The `hrefLang` property is an advisory for the browser (if the browser takes advantage of it) about the written language used for the content to which the `link` element's `href` attribute points. Values for this property must be in the form of the standard language codes (for example, `en-us` for U.S. English).

media

Value: String. Read/Write
Compatibility: WinIE4+, MacIE-, NN6+, Moz1+, Safari1+

The media property is an advisory for the browser about the target output device intended for the content to which the link element's href attribute points. This is an outgrowth of HTML 4.0 efforts to make way for future browsers and content that can be optimized for devices such as printers, handheld computers, and audio digitizers. The W3C specifies a preliminary set of constant string values for this property's equivalent attribute. So far, browsers (at most) recognize all (default), print, and screen.

rel
rev

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The rel and rev properties are intended to define relationships in the forward and back directions with respect to the current document. Browsers have yet to exploit most of the potential of these attributes and properties. For the most part, the attributes solely direct the browser to treat the external content as a stylesheet definition file.

A long list of values is predefined for these properties, based on the corresponding attribute values specified in HTML 4.0. If the browser does not respond to a particular value, the value is simply ignored. You can string together multiple values in a space-delimited list inside a single string. Accepted values are as follows:

alternate	contents	index	start
appendix	copyright	next	stylesheet
bookmark	glossary	prev	subsection
chapter	help	section	

sheet

Value: Object. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

When a link element loads an external stylesheet, the W3C DOM sheet property of the link element object provides scripted access to the stylesheet rules that belong to that external file. Use properties of the sheet object to access specifics about the imported rules.

styleSheet

Value: Object. Read-Only

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

When a link element loads an external stylesheet, the IE-specific styleSheet property of the link element object provides scripted access to the stylesheet rules that belong to that external file. Use properties of the styleSheet object (see Chapter 26) to access specifics about the imported rules.

target

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

link.target

In the context of using `link` elements to point to other content associated with the current document (for example, the next and previous documents within a series), the `target` property can advise the browser which frame or window to use to display that content. For example, a suitably equipped browser can display a glossary in a separate window. No browsers currently implement these extended features of the `link` element, so the property is provided in browsers only for compatibility with the W3C standards. If the property were truly functional, it would accept values in the form of a string name for a frame or one of the window constants (`_blank`, `_parent`, `_self`, or `_top`).

`type`

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `type` property specifies the `mime` type for the content that will arrive from the external document to which the element's `href` attribute points. `link` elements are used primarily for Cascading Style Sheets, so the property value is `text/css`.

Event handlers

`onload`

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `onload` event handler fires when the external content pointed to by the `link` element's `href` attribute completes loading. WinIE5 fires this event handler even if the loading does not succeed, so use this event handler with care.

meta Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`charset`		
`content`		
`httpEquiv`		
`name`		
`url`		

Syntax

Accessing `meta` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

About this object

In computer terminology, *metadata* usually consists of extra information about the primary data of a document or information collection. In HTML documents, metadata can be additional hidden information about the document, such as the name of the author and keywords. If the browser is suitably equipped, metadata can also include some instructions, such as when to reload the page by itself. `meta` elements add all of this metadata to HTML documents. Both fact and folklore surround the application of `meta` elements within pages. One fact is that Internet search engine robots used to scour pages for certain kinds of keyword meta tags to help place your page within relevant categories when Web surfers are looking for specific content (but thanks to keyword loading by unscrupulous Web sites, search engines now rarely consider `meta` keywords). More on the folklore side is that browsers always respond to `meta` element wording that prevents browsers from copying pages into the cache — when in fact, this behavior is not universal among browsers.

Complete details about `meta` element usage is beyond the scope of this JavaScript book, but you should be aware of one widely accepted composition that enables you to set a page to reload itself (or another page) at a fixed time interval. This is especially useful if your page retrieves very timely information from a database. The format is

```
<meta http-equiv="refresh" content="n,url=url" />
```

n is the number of seconds to delay before reloading the page, and *url* is the complete URL of the page to be reloaded. Note that you can specify any page you like. This allows for a kind of slide show to be sequenced in a freestanding kiosk, because each page's `meta` element points to the next page in the series after a fixed amount of time.

Unique properties for the `meta` element object mimic the HTML attributes for the `<meta>` tag. Browsers read most attributes only at load time, which means that scripted changes to values after the page loads have no effect. These properties are rarely, if ever, accessed from a script, so I mention them here only briefly.

Properties

charset

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `charset` property advises the browser about the character encoding of the content for the page. Values for this property must match the encoding naming conventions defined in an industry standard registry (`http://www.iana.org/assignments/character-sets`).

content

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

For many applications of the `meta` element, the `content` property contains the primary value associated with the element. For example, search engines look for a `meta` element whose `name` attribute is `"keywords"`. The value of the `content` attribute is a comma-delimited string of keywords that the search engine reads and indexes in its own database. The `content` property simply represents the `content` attribute string. Changing the values by script obviously does nothing to alter the tag values of the page on the server.

httpEquiv

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

A `meta` element can simulate and extend the transmission of server instructions to the browser — instructions that normally arrive in the form of http headers. These header supplements are supplied in `meta` elements via the `http-equiv` attribute, which is represented in the object model by the `httpEquiv` property. Common values include `refresh` and `expires`. Each of these also requires a `content` attribute that provides necessary details for carrying out the instructions. If you assign a string value to the `httpEquiv` property, be sure the `content` property has a suitable string assigned to it.

name

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

A `meta` element that includes genuine metadata about the page (for example, author or keywords) usually has a `name` attribute that identifies what the metadata is (analogous to the name of a name-value pair). The `name` and `content` properties go hand in hand because the content string usually must be in a particular form for an external process (for example, a search engine) to read the data successfully. Values for the `name` attribute are rarely case-sensitive.

url

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

If a `meta` element needs to point to a document on the Internet for any reason, the URL of that document is assigned to the `url` attribute of the element. You can modify the value via the `url` property of a `meta` element object. I recommend a complete URL string for the `url` property value.

script Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`defer`		
`event`		
`htmlFor`		
`src`		
`text`		
`type`		

Syntax

Accessing `script` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

About this object

The `<script>` tag is well known to scripters, and modern browsers (IE4+ and W3C DOM-compatible) treat the `script` element as an object that, itself, can be scripted. The circularity of this description isn't as far-fetched as it sounds. Although scripting an existing script is a rarity in practice, it is not out of the question to generate a new `script` element after the page loads. If you use W3C DOM syntax to create a new `script` element, you then need to assign values to the properties that are normally set via the tag's attributes. Thus, scripting a script does make sense.

Unless you have experience with IE's option of binding event handlers to `<script>` tags (see Chapter 14), some of the properties described next will be foreign to you. Even so, these properties are now a part of the W3C DOM specification, so they are implemented in IE5+/NN6+/Mozilla/Safari as well.

Properties

defer

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The default process of loading a page that contains scripts is to wait for any immediate script execution to complete before the rest of the page loads. But if you include a `defer` attribute in the tag, modern browsers (except Safari 1.0) continue to load the rest of the page without waiting for immediate scripts to run. The `defer` property enables you to inspect or set that property; its default value is `false`. Once a page loads, any change you make to an existing `script` element's `defer` property has no effect.

event
htmlFor

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Internet Explorer enables you to bind events to script statements when you specify both a `for` and `event` attribute in the `<script>` tag. Statements inside the tag execute only when the object named by the `for` attribute receives the event named by the `event` attribute. You can examine the `event` attribute by way of the `script` element object's `event` property, and you can view the `for` attribute through the `htmlFor` property. Both properties simply mimic whatever values are assigned to their respective attributes, such as `onclick()` and `myDIV`.

src

Value: String. See Text
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `src` property is a string of the URL of an external `.js` script file to be linked into a page. You can change this property in IE after you load the external script, but the old script does not go away from the page. If the new script defines the same variable and function names, the new versions overwrite the old. Other browsers may not load a new external script.

text

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The full text of a `script` element is available for reading through the `text` property. Although IE5+ may give the impression that you can modify this property, the script that loads with the page is what is stored in the browser's memory. Thus, the original script statements continue to work even though the object's property is different.

type

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `type` attribute was added to the `<script>` tag in HTML 4.0 to help resolve the conflict that the deprecated `language` attribute created for all HTML elements. The value of the attribute (and thus the `type` property) is a `mime` type string. For JavaScript, that value is `text/javascript`.

title Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`text`		

Syntax

Accessing `title` element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

About this object

Before the `title` element was accessible to scripting as an object, the prescribed way to get to the content of the page's `<title>` tag was through the `document.title` property. Although that property is still available for backward compatibility, scripts written exclusively for newer browsers should access the `text` property of the `title` element object. As a useful exercise, you can modify Listing 18-17 (loaded via Listing 18-16) to use the IE4+ or W3C DOM syntax to retrieve and display the document's title.

Property

`text`

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

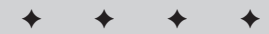
The `text` property represents the text between the start and end tags of the `title` element object (Safari 1.0 returns an empty string). This is simply a convenience property because the text can be referenced by other ways in IE4+ (`innerText` property), NN6+ (`innerHTML`), and W3C DOM (`firstChild.nodeValue`) syntaxes. For backward compatibility with earlier browsers, you can alternatively use the `document.title` property.

Related Items: `document.title` property.

✦ ✦ ✦

CHAPTER 37

Table and List Objects



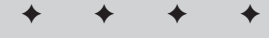
In This Chapter

Modifying table cell content

Adding and deleting table rows

`table`, `caption`, `tbody`, `tfoot`, `thead`, `col`, `colgroup`, `th`, `tr`, and `td` element objects

`ol`, `ul`, `li`, and `dl` list element objects



Tables are incredibly popular HTML constructions. When you consider that a lot of CGI programs search SQL databases and display data gathered from SQL tables, it's not unusual to find the table concept carried over from data storage to data display. Spreadsheet programs certainly put the notion of tabular display into the minds of most computer users.

One of the truly beneficial properties of tables in HTML is that they pack a lot of page organization and alignment punch in just a few tags and attributes. Even if you're not a graphics designer or a dedicated HTML jockey, you can get rows and columns of text and images to line up perfectly on the page. This behavior also lures many page designers to sculpt elaborately detailed pages out of what appear to be positioned elements. Earlier browsers didn't offer positioning facilities, so borderless tables were torqued into performing all kinds of placement tricks with the help of precisely sized, transparent images creating the illusion of white space between carefully placed elements. If you use some of the WYSIWYG authoring tools for HTML pages, you may not realize how much table-related HTML code is generated for you as you use the tool to drag an image to a particular location on the page.

Using tables to specify design and page layout is giving way to CSS techniques that achieve a similar look with less code. This trend is driven by Web standards goals of using HTML markup to denote context rather than layout. Even so, many Web pages need to display genuinely columnar data — a purpose for which HTML tables are still ideally suited. The first part of this chapter focuses on the scriptable aspects of `table` element objects and the shopping list of elements that support tables. All of these objects became scriptable objects starting with IE4 and NN6/W3C DOM-compatible browsers. Later in the chapter, I discuss element objects that create formatted lists in pages.

The Table Object Family Hierarchy

The repertoire of table-related elements expanded a bit with the HTML 4.0 specification, and the W3C DOM built upon that foundation. While most of this discussion is best left to HTML texts, the structure of a full-fledged table and the relationships among the elements — particularly the parent-child relationships — may affect your scripting and event handling.

You are probably very familiar with the most basic table structure that predates HTML 4.0. Such a table (in a 2×2 layout) can have the following form:

```
<table>
   <tr>
      <td></td>
      <td></td>
   </tr>
   <tr>
      <td></td>
      <td></td>
   </tr>
</table>
```

If you want to place a row of cells at the top of each column such that the contents of the cells act as headers for each column, add such a row as follows:

```
<table>
   <tr>
      <th></th>
      <th></th>
   </tr>
   <tr>
      <td></td>
      <td></td>
   </tr>
   <tr>
      <td></td>
      <td></td>
   </tr>
</table>
```

You can also include a caption associated with the table. Its tag goes immediately after the `table` element's start tag:

```
<table>
   <caption></caption>
   <tr>
      <th></th>
      <th></th>
   </tr>
   <tr>
      <td></td>
      <td></td>
   </tr>
   <tr>
      <td></td>
      <td></td>
   </tr>
</table>
```

In line with its emphasis on providing contextual tags, HTML 4.0 includes three tags that enable you to define groups of table rows according to whether they are the header, body, or footer of the table (`thead`, `tbody`, and `tfoot` elements, respectively). A table footer, for example, can display column totals. The only seemingly illogical rule about these elements is that you should define the `tfoot` element and its row contents before the `tbody` element(s) in the table. Even with this source code placement, the `tfoot` row appears at the bottom of the table.

Some browsers produce visual dividers between these sections (WinIE5+ does a nice job of this). Moreover, you can have multiple `tbody` sections within a table. Some browsers render dividers between these `tbody` sections (again, WinIE5+ does it well). Regardless of the built-in divider support, these contextual groupings also enable you to assign stylesheets to HTML tag selectors rather than having to dream up a scheme of class and `id` names tied to stylesheet rules. Building upon the skeletal table shown thus far, you add the `thead` and `tbody` elements like this:

```
<table>
   <caption></caption>
   <thead>
      <tr>
         <th></th>
         <th></th>
      </tr>
   </thead>
   <tbody>
      <tr>
         <td></td>
         <td></td>
      </tr>
      <tr>
         <td></td>
         <td></td>
      </tr>
   </tbody>
</table>
```

That's the extent of table-oriented HTML containers. The remaining two elements, `colgroup` and `col`, provide a different "slice" of the table for stylesheets and other visual groupings. One of the most obvious purposes of these two elements is to assign a width or other style to all cells in a particular column or group of columns. You can also use these elements to group adjacent columns so that dividers are drawn between groups of columns — if the browser (such as WinIE5+) supports dividers between column groups — without specifying global table borders. You can see an example of the HTML for a complex table in the HTML 4.0 specification (`http://www.w3.org/tr/REC-html40/struct/tables.html#h-11.5`). Elsewhere on that same page, you can find the formal specification for all table-related tags and attributes as defined by the W3C.

Populating table cells

Source material for a table's content can come from many different places. Most of the tables you see on the Web are hard-coded in the HTML. That is, the content of the table is fixed inside a static HTML file on the server.

But tables may also convey content from live databases or content that changes more frequently than the Web site's author manually updates other content. The source and your Web development infrastructure (not to mention your technical skills) dictate other avenues for populating tables.

After hard-coded HTML files, the next most common way to generate tables is through server-based CGI programs. These programs (written in Perl, C, and many other languages, including server-side JavaScript on those few servers that support it) generally compose a query for the database and then repackage the data returned from the database into HTML-formatted pages.

A more client-side–oriented approach is to let JavaScript apply the `document.write()` method to compose the table's tags as the page loads. Data for the cells can come from JavaScript arrays defined at the beginning of the document or defined in external `.js` library files that are linked in as the page loads. In the newest browsers, the data may come from blocks of XML-formatted data stuffed into the document. These solutions can work in situations where you need to update the table data periodically, but the table delivered to the client does not reflect the instantaneous state of a database. For example, a daily batch program on a server can capture the day's sales totals and write out a `.js` text file to a known place on the server. The file consists entirely of JavaScript array definitions. When the HTML page loads, the current `.js` file is automatically loaded into the page, and `document.write()` statements compose the table's HTML from the data supplied in the arrays. Although the script that assembles the HTML for the tables might appear formidable to a nonscripter, a nonscripter can also manually update the array data by following a template format supplied by the programmer.

Finally, if your page visitors run IE4+, you can take advantage of a Microsoft-specific technology called *data binding*. Data binding invokes the powers of one or more ActiveX controls that come with the IE browser (simulated in MacIE). These objects (collectively called Data Source Objects) let HTML pages access ODBC databases and structured text files (MacIE works only with text files). As the page loads, the table fills with data pulled live from the database. You can see an example of data binding in Chapter 15 under the description of the data binding property: `dataFld`. The HTML file carries tags for only one row of cells, but data binding fills in the rest of the rows and cells.

Modifying table cell content

You can modify the HTML content of a table cell directly in IE4+ and W3C DOM-compatible browsers. Some tricks with positioned elements in NN4 can, under some circumstances, make it appear to the user as if the table content is being modified.

By far, the most compatible way to modify a table cell's content in IE4+ and W3C DOM browsers is via the `td` element's `innerHTML` property (a Microsoft invention that is not sanctioned by the W3C DOM Level 2 but is supported elsewhere as a de facto standard). Even if the content is simply text that is to inherit the style format of the surrounding `td` element, you can still use the `innerHTML` property. If the size of the new content affects the dimensions of the cell's column width or row height, the browser reflows the rest of the table content around the new content.

If you prefer to follow the W3C DOM form of modifying an element's content (for IE5+ and NN6+/Mozilla/Safari), you can generate the new content via the `document.createElement()` or `document.createTextNode()` sequence and assign that new content to the cell by way of the `td` element's `replaceChild()` method.

The situation for NN4 is quite gnarled because the content you replace must be within its own layer (either a `layer` element or positioned container element, such as a `div` or `span`). No matter how you create the layer in your HTML, you must overcome the problem that a layer floats in its own plane and must be positioned precisely where the table cell is. Table cells are not objects in NN4, so you must create a positioning context in the cell by first creating a relative-positioned layer that can contain nothing more than an "invisible" nonbreaking space character (` `). The layer displaying the content must be absolute-positioned with respect to that relative-positioned layer. Nesting of layers in NN4 causes headaches, especially when scripts reference the deeply nested content — content that is, essentially, an HTML document inside the nested layer.

Listing 37-1 shows a synthesis of different techniques to effect cell content replacement, including script code branches that emulate the appearance of replacement in NN4. The table represents only one line of what might be an order form for several products. As the user makes a selection of the quantity, the extended total is displayed in the rightmost column.

You can find the key features of the NN4 implementation in the script that dynamically writes the table cell content within the HTML as the page loads. The cell begins with a relative-positioned `span` element. This `span` is positioned at the top left of the table cell, as planned. That spot now is the positioning context for the absolute-positioned `span` nested inside it. This second `span` is the layer whose document contains the displayed content. The content, itself, is yet another `span` element because it simplifies the application of a stylesheet rule (to display the total in red) when you replace the content. Because a newly written NN4 layer does not inherit the stylesheet of its next outermost layer, you must apply the style as part of the new content.

The initial `span` content contains a series of nonbreaking space characters that force NN4 to open space for eventual replacement content. Recall that an NN4 page does not reflow the page to accommodate resized content. This means that whatever you intend to insert in the table cell can be no larger than the original space allocated for it.

Although the page shown in Listing 37-1 consists of only one row of data, the scripts and naming conventions are intended to be carried out among multiple rows. The product name appears in several object names and `id`s in each row, and the scripts count on the convention being followed throughout. In fact, the regularity of the namings can allow the content for a table's row to form a script function that is invoked for each table row. The product code name can be passed as the parameter, and all object names and `id`s can be assembled in that function. The regularity of table content often lends itself to script-generated construction.

For NN4, when the table gets complicated, you will have more success defining the absolute-positioned elements outside of the table entirely. They should be defined on their own at the bottom of the `body`. You can still position them with respect to the relative-positioned elements in the table, but all such layers are now only one level deep within the main document.

Listing 37-1: **Replacing Table Cell Content**

```
<html>
   <head>
      <title>Modifying Table Cell Content</title>
      <style type="text/css">
      .absoluteWrap {position:absolute}
      .relativeWrap {position:relative}
      .total {color:red}
      </style>
      <script type="text/javascript">
      var Ver4 = parseInt(navigator.appVersion) == 4;
      var Ver4Up = parseInt(navigator.appVersion) >= 4;
      var Nav4 = ((navigator.appName == "Netscape") && Ver4);
      var modifiable;

      // calculate and display a row's total
```

Continued

Listing 37-1 *(continued)*

```
   function showTotal(qtyList) {
      var qty = qtyList.options[qtyList.selectedIndex].value;
      var prodID = qtyList.name;
      var total = "US$" + (qty * parseFloat(qtyList.form.elements[prodID +
         "Price"].value));
      var newCellHTML = "<span class='total'>" + total + "</span>";

      if(Nav4) {
         document.layers[prodID + "TotalWrapper"].document.layers[prodID +
            "Total"].document.write(newCellHTML);
         document.layers[prodID + "TotalWrapper"].document.layers[prodID +
            "Total"].document.close();
      } else if (modifiable) {
         if (document.all) {
            document.all(prodID + "Total").innerHTML = newCellHTML;
         } else {
            document.getElementById(prodID + "Total").innerHTML =
               newCellHTML;
         }
      }
   }

   // initialize global flag for browsers capable of modifiable content
   function init() {
      modifiable  = (Ver4Up && document.body && document.body.innerHTML);
   }

   // display content for all products (e.g., in case of Back navigation)
   function showAllTotals(form) {
      for (var i = 0; i < form.elements.length; i++) {
         if (form.elements[i].type == "select-one") {
            showTotal(form.elements[i]);
         }
      }
   }
   </script>
</head>
<body onload="init(); showAllTotals(document.orderForm)">
   <h1>Modifying Table Cell Content</h1>
   <hr />
   <form name="orderForm">
      <table border="1">
         <colgroup width="150"></colgroup>
         <colgroup width="100"></colgroup>
         <colgroup width="50"></colgroup>
         <colgroup width="100"></colgroup>
         <tr>
            <th>Product Description</th>
            <th>Price Each</th>
            <th>Quantity</th>
            <th>Total</th>
         </tr>
         <tr>
            <td>Wonder Widget 9000</td>
            <td>US$125.00</td>
            <td><select name="ww9000" onchange="showTotal(this)">
```

```
                  <option value="0">0</option>
                  <option value="1">1</option>
                  <option value="2">2</option>
                  <option value="3">3</option>
               </select> <input type="hidden" name="ww9000Price"
               value="125.00" /></td>
            <td>
               <script type="text/javascript">
               if (Nav4) {
                  var placeHolder =
                     "        ";
                  placeHolder +=
                     "       ";
                  document.write("<span id='ww9000TotalWrapper'
                     class='relativeWrap'>");
                  document.write("<span id='ww9000Total'
                     class='absoluteWrap'><\/span>");
                  document.write("<span>" + placeHolder +
                     "<\/span><\/span>");
               } else {
                  document.write("<span id='ww9000Total'
                     class='relativeWrap'><p> <\/p><\/span>");
               }
               </script>
            </td>
         </tr>
      </table>
   </form>
</body>
</html>
```

Modifying table rows

In IE4+ and W3C DOM browsers, all table-related elements are full-fledged objects within the browser's object model. This means that you are free to use your choice of DOM element modification techniques on the row and column makeup of a table. But due to the frequent complexity of tables and all of their nested elements, the code required to manage a table can balloon in size. To the rescue come some methods that enable you to add and remove rows and cells from a table. Despite minor differences in the implementations of these methods across DOMs, the syntax exhibits sufficient unanimity to allow one set of code to work on both browsers — especially for adding elements to a table.

Table 37-1 provides a quick summary of the key methods used to add or remove elements within a `table`, a table section (`thead`, `tbody`, or `tfoot`), and a row (`tr`). For simple tables (in other words, those that do not define `thead` or `tfoot` segments), you can work exclusively with the row modification methods of the `table` element object (and then the cell modification methods of the rows within the `table` element). The reason for the duplication of the row methods in the table section objects is that instead of having to worry about row index numbers lining up among the combined total of head, body, and foot rows, you can treat each segment as a distinct unit. For example, if you want to add a row just to the beginning of the `tfoot` section, you can use the `insertRow()` method for the `tfoot` element object and not have to count up the `tr` elements and perform arithmetic to arrive at the desired row number. Instead, simply use the `insertRow()` method on the `tfoot` element object and supply the method with parameters that ensure the row is inserted as the first row of the element.

IE5 for the Macintosh offers unpredictable results when inserting rows of a table via these methods. The browser does behave when modifying the HTML elements by accumulating the HTML for a row as a string and then adding the row to the table via IE DOM methods such as `insertAdjacentHTML()`. If your pages must modify the composition of tables after the page loads—and your audience includes Mac IE5 users—use the element and node insertion techniques rather than the methods shown in Table 37-1 and techniques described next.

Table 37-1: IE4+ and NN6+/W3C Table Modification Methods

table	*thead, tbody, tfoot*	*tr*
`insertRow()`	`insertRow()`	`insertCell()`
`deleteRow()`	`deleteRow()`	`deleteCell()`
`createTHead()`		
`deleteTHead()`		
`createTFoot()`		
`deleteTFoot()`		
`createCaption()`		
`deleteCaption()`		

The basic sequence for inserting a row into a table entails the following steps:

1. Invoke `insertRow()` and capture the returned reference to the new, unpopulated row.
2. Use the reference to the row to invoke `insertCell()` for each cell in the row, capturing the returned reference to each new, unpopulated cell.
3. Assign values to properties of the cell, including its content.

The following code fragment appends a new row to a table (`myTABLE`) and supplies information for the two cells in that row:

```
// parameter of -1 appends to table
// (you can use document.all.myTABLE.insertRow(-1) for IE4+ only)
var newRow = document.getElementById("myTABLE").insertRow(-1);
// parameter of 0 inserts at first cell position
var newCell = newRow.insertCell(0);
newCell.innerHTML = "Mighty Widget 2000";
// parameter of 1 inserts at second cell position
newCell = newRow.insertCell(1);
newCell.innerHTML = "Release Date TBA";
```

A key point to note about this sequence is that the `insertRow()` and `insertCell()` methods do their jobs before any content is handed over to the table. In other words, you first create the HTML space for the content and then add the content.

Listing 37-2 presents a living environment that adds and removes `thead`, `tr`, and `tfoot` elements to an empty table in the HTML. The only subelement inside the `table` element is a `tbody` element, which directs the insertion of table rows so as not to disturb any existing `thead` or `tfoot` elements. You can also see how to add or remove a caption from a table via caption-specific methods.

Each table row consists of the hours, minutes, seconds, and milliseconds of a time stamp generated when you add the row. The color of any freshly added row in the `tbody` is a darker color than the normal `tbody` rows. This is so you can see what happens when you specify an index value to the `insertRow()` method. Some of the code here concerns itself with enabling and disabling form controls and updating `select` elements, so don't be deterred by the length of Listing 37-2.

Listing 37-2: **Inserting/Removing Row Elements**

```
<html>
   <head>
      <title>Modifying Table Cell Content</title>
      <style type="text/css">
      thead {background-color:lightyellow; font-weight:bold}
      tfoot {background-color:lightgreen; font-weight:bold}
      #myTABLE {background-color:bisque}
      </style>
      <script type="text/javascript">
      var theTable, theTableBody;
      function init() {
         theTable = (document.all) ? document.all.myTABLE :
            document.getElementById("myTABLE");
         theTableBody = theTable.tBodies[0];
      }
      function appendRow(form) {
         insertTableRow(form, -1);
      }

      function addRow(form) {
         insertTableRow(form, form.insertIndex.value);
      }

      function insertTableRow(form, where) {
         var now = new Date();
         var nowData = [now.getHours(), now.getMinutes(), now.getSeconds(),
            now.getMilliseconds()];
         clearBGColors();
         var newCell;
         var newRow = theTableBody.insertRow(where);
         for (var i = 0; i < nowData.length; i++) {
            newCell = newRow.insertCell(i);
            newCell.innerHTML = nowData[i];
            newCell.style.backgroundColor = "salmon";
         }
         updateRowCounters(form);
      }

      function removeRow(form) {
         theTableBody.deleteRow(form.deleteIndex.value);
         updateRowCounters(form);
      }
```

Continued

Listing 37-2 *(continued)*

```
function insertTHEAD(form) {
   var THEADData = ["Hours","Minutes","Seconds","Milliseconds"];
   var newCell;
   var newTHEAD = theTable.createTHead();
   newTHEAD.id = "myTHEAD";
   var newRow = newTHEAD.insertRow(-1);
   for (var i = 0; i < THEADData.length; i++) {
      newCell = newRow.insertCell(i);
      newCell.innerHTML = THEADData[i];
   }
   updateRowCounters(form);
   form.addTHEAD.disabled = true;
   form.deleteTHEAD.disabled = false;
}

function removeTHEAD(form) {
   theTable.deleteTHead();
   updateRowCounters(form);
   form.addTHEAD.disabled = false;
   form.deleteTHEAD.disabled = true;
}

function insertTFOOT(form) {
   var TFOOTData = ["Hours","Minutes","Seconds","Milliseconds"];
   var newCell;
   var newTFOOT = theTable.createTFoot();
   newTFOOT.id = "myTFOOT";
   var newRow = newTFOOT.insertRow(-1);
   for (var i = 0; i < TFOOTData.length; i++) {
      newCell = newRow.insertCell(i);
      newCell.innerHTML = TFOOTData[i];
   }
   updateRowCounters(form);
   form.addTFOOT.disabled = true;
   form.deleteTFOOT.disabled = false;
}

function removeTFOOT(form) {
   theTable.deleteTFoot();
   updateRowCounters(form);
   form.addTFOOT.disabled = false;
   form.deleteTFOOT.disabled = true;
}

function insertCaption(form) {
   var captionData = form.captionText.value;
   var newCaption = theTable.createCaption();
   newCaption.innerHTML = captionData;
   form.addCaption.disabled = true;
   form.deleteCaption.disabled = false;
}

function removeCaption(form) {
   theTable.deleteCaption();
   form.addCaption.disabled = false;
   form.deleteCaption.disabled = true;
```

```
    }

    // housekeeping functions
    function updateRowCounters(form) {
       var sel1 = form.insertIndex;
       var sel2 = form.deleteIndex;
       sel1.options.length = 0;
       sel2.options.length = 0;
       for (var i = 0; i < theTableBody.rows.length; i++) {
          sel1.options[i] = new Option(i, i);
          sel2.options[i] = new Option(i, i);
       }
       form.removeRowBtn.disabled = (i==0);
    }

    function clearBGColors() {
       for (var i = 0; i < theTableBody.rows.length; i++) {
          for (var j = 0; j < theTableBody.rows[i].cells.length; j++) {
             theTableBody.rows[i].cells[j].style.backgroundColor = "";
          }
       }
    }
    </script>
</head>
<body onload="init()">
    <h1>Modifying Tables</h1>
    <hr />
    <form name="controls">
       <fieldset>
          <legend>Add/Remove Rows</legend>
          <table width="100%" cellspacing="20">
             <tr>
                <td><input type="button" value="Append 1 Row"
                   onclick="appendRow(this.form)" /></td>
                <td><input type="button" value="Insert 1 Row"
                   onclick="addRow(this.form)" /> at index: <select
                   name="insertIndex">
                      <option value="0">0</option>
                   </select></td>
                <td><input type="button" name="removeRowBtn"
                   value="Delete 1 Row" disabled="disabled"
                   onclick="removeRow(this.form)" /> at index: <select
                   name="deleteIndex">
                      <option value="0">0</option>
                   </select></td>
             </tr>
          </table>
       </fieldset>
       <fieldset>
          <legend>Add/Remove THEAD and TFOOT</legend>
          <table width="100%" cellspacing="20">
             <tr>
                <td><input type="button" name="addTHEAD" value="Insert THEAD"
                   onclick="insertTHEAD(this.form)" /><br />
                   <input type="button" name="deleteTHEAD"
                   value="Remove THEAD" disabled="disabled"
                   onclick="removeTHEAD(this.form)" /></td>
```

Continued

Listing 37-2 *(continued)*

```
                <td><input type="button" name="addTFOOT" value="Insert TFOOT"
                   onclick="insertTFOOT(this.form)" /><br />
                   <input type="button" name="deleteTFOOT"
                   value="Remove TFOOT" disabled="disabled"
                   onclick="removeTFOOT(this.form)" /></td>
             </tr>
          </table>
       </fieldset>
       <fieldset>
          <legend>Add/Remove Caption</legend>
          <table width="100%" cellspacing="20">
             <tr>
                <td><input type="button" name="addCaption"
                   value="Add Caption"
                   onclick="insertCaption(this.form)" /></td>
                <td>Text: <input type="text" name="captionText" size="40"
                   value="Sample Caption" /></td>
                <td><input type="button" name="deleteCaption"
                   value="Delete Caption" disabled="disabled"
                   onclick="removeCaption(this.form)" /></td>
             </tr>
          </table>
       </fieldset>
    </form>
    <hr />
    <table id="myTABLE" cellpadding="10" border="1">
       <tbody></tbody>
    </table>
  </body>
</html>
```

Modifying table columns

Unlike the table row-oriented elements, such as `tbody`, the `col` and `colgroup` elements are not containers of cells. Instead, these elements serve as directives for the rendering of columns within a table. But through scripting, you can add or remove one or more columns from a table on the fly. There is no magic to it; you simply insert or delete the same-indexed cell from every row of the table.

Listing 37-3 demonstrates adding and removing a left-hand column of a table. The table presents the four longest rivers in Africa, and the new column provides the numeric ranking. Thanks to the regularity of this table, the values for the rankings can be calculated dynamically. Note, too, that the `className` property of each new table cell is set to a class that has a stylesheet rule defined for it. Instead of inheriting the style of the table, the cells obey the more specific background color and font weight rules defined for the cells.

Listing 37-3: **Modifying Table Columns**

```
<html>
   <head>
      <title>Modifying Table Columns</title>
      <style type="text/css">
      thead {background-color:lightyellow; font-weight:bold}
      .rankCells {background-color:lightgreen; font-weight:bold}
      #myTABLE {background-color:bisque}
      </style>
      <script type="text/javascript">
      var theTable, theTableBody;
      function init() {
         theTable = (document.all) ? document.all.myTABLE :
            document.getElementById("myTABLE");
         theTableBody = theTable.tBodies[0];
      }

      function insertColumn(form) {
         var oneRow, newCell, rank;
         if (theTable.tHead) {
            oneRow = theTable.tHead.rows[0];
            newCell = oneRow.insertCell(0);
            newCell.innerHTML = "Ranking";
         }
         rank = 1;
         for (var i = 0; i < theTableBody.rows.length; i++) {
            oneRow = theTableBody.rows[i];
            newCell = oneRow.insertCell(0);
            newCell.className = "rankCells";
            newCell.innerHTML = rank++;
         }
         form.addColumn.disabled = true;
         form.removeColumn.disabled = false;
      }

      function deleteColumn(form) {
         var oneRow;
         if (theTable.tHead) {
            oneRow = theTable.tHead.rows[0];
            oneRow.deleteCell(0);
         }
         for (var i = 0; i < theTableBody.rows.length; i++) {
            oneRow = theTableBody.rows[i];
            oneRow.deleteCell(0);
         }
         form.addColumn.disabled = false;
         form.removeColumn.disabled = true;
      }
      </script>
   </head>
   <body onload="init()">
      <h1>Modifying Table Columns</h1>
      <hr />
```

Continued

Listing 37-3 *(continued)*

```
      <form name="controls">
         <fieldset>
            <legend>Add/Remove Left Column</legend>
            <table width="100%" cellspacing="20">
               <tr>
                  <td><input type="button" name="addColumn"
                     value="Insert Left Column"
                     onclick="insertColumn(this.form)" /></td>
                  <td><input type="button" name="removeColumn"
                     value="Remove Left Column" disabled="disabled"
                     onclick="deleteColumn(this.form)" /></td>
               </tr>
            </table>
         </fieldset>
      </form>
      <hr />
      <table id="myTABLE" cellpadding="5" border="1">
         <thead id="myTHEAD">
            <tr>
               <td>River</td>
               <td>Outflow</td>
               <td>Miles</td>
               <td>Kilometers</td>
            </tr>
         </thead>
         <tbody>
            <tr>
               <td>Nile</td>
               <td>Mediterranean</td>
               <td>4160</td>
               <td>6700</td>
            </tr>
            <tr>
               <td>Congo</td>
               <td>Atlantic Ocean</td>
               <td>2900</td>
               <td>4670</td>
            </tr>
            <tr>
               <td>Niger</td>
               <td>Atlantic Ocean</td>
               <td>2600</td>
               <td>4180</td>
            </tr>
            <tr>
               <td>Zambezi</td>
               <td>Indian Ocean</td>
               <td>1700</td>
               <td>2740</td>
            </tr>
         </tbody>
      </table>
   </body>
</html>
```

W3C DOM table object classes

If you ever read the W3C DOM Level 2 specification, notice that the objects defined for tables do not align themselves fully with the actual elements defined in the HTML 4.0 specification. That's not to say the DOM scoffs at the HTML spec; rather, the needs of a DOM with respect to tables differ a bit. For example, as far as the W3C DOM is concerned, the `thead`, `tbody`, and `tfoot` are all regarded as table sections and are thus known as `HTMLTableSectionElement` objects. In other words, in the W3C DOM, there is no particular distinction among the types of table section elements. They're all lumped together, and they bear the same properties and methods. With their strong adherence to the W3C DOM, Mozilla-based browsers and Safari stick to the W3C DOM object constructions.

When you work in both the IE and W3C DOMs at the same time, it's helpful to know the relationships between the object naming conventions used in each. Table 37-2 provides a quick cross-reference between the object types in both DOMs. None of the terminology in Table 37-2 affects the way scripts construct references to elements or the way elements are nested within one another. The containment hierarchy is driven by the HTML element containment — and that remains the same regardless of DOM exposure.

Table 37-2: Table Object Classifications

W3C DOM (NN6+/Mozilla/Safari)	*IE4+ and HTML*
`HTMLTableElement`	`table`
`HTMLTableCaptionElement`	`caption`
`HTMLTableColElement`	`col`, `colgroup`
`HTMLTableSectionElement`	`tbody`, `tfoot`, `thead`
`HTMLTableRowElement`	`tr`
`HTMLTableCellElement`	`td`, `th`

Although the following object-specific discussions list the objects according to their HTML tag name, I group these objects according to the W3C DOM classifications because element objects that share a classification also share the same properties, methods, and event handlers.

table Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`align`	`createCaption()`	`onscroll`
`background`	`createTFoot()`	
`bgColor`	`createTHead()`	
`border`	`deleteCaption()`	
`borderColor`	`deleteRow()`	
`borderColorDark`	`deleteTFoot()`	

Continued

table

Properties	*Methods*	*Event Handlers*
borderColorLight	deleteTHead()	
caption	firstPage()	
cellPadding	insertRow()	
cells	lastPage()	
cellSpacing	moveRow()	
cols	nextPage()	
datePageSize	previousPage()	
frame	refresh()	
height		
rows		
rules		
summary		
tbodies		
tFoot		
tHead		
width		

Syntax

Accessing `table` element object properties and methods:

```
(IE4+)        [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C)    [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About this object

The `table` element object is the outermost container of table-related information. The HTML element has a large number of attributes, most of which are echoed by their counterpart properties in the object model. You rarely will modify these properties if the values are set in the tag's attributes. However, if you construct a new `table` element object for insertion into the page, use these properties to assign values to the equivalents of the element's attributes.

A number of additional properties return collections of cell, row, and row section objects; still more properties return references to other, singular objects within the table (such as the `caption` element object). For example, if your script needs to iterate through all rows within just the `tbody` elements (in other words, without affecting the rows in the `thead` element), your script can perform a nested `for` loop to access each row:

```
var oneTBody, oneRow;
for (var i = 0; i < tableRef.tBodies.length; i++) {
   oneTBody = tableRef.tBodies[i];
   for (var j = 0; j < oneTBody.rows.length; j++) {
      oneRow = oneTBody.rows[j];
      // more stuff working on each row
   }
}
```

For a simple table that does not define table row sections, you can iterate through the `rows` collection property of a `table` element object. You can even access cells directly; but it may be easier to keep track of cells in a loop by going through them row by row (via the `cells` property of each `tr` element object).

A large number of methods enable you to modify the structure of a table (as described earlier in this chapter), but they primarily work with rows. Column modifications require a different approach, as also demonstrated earlier.

Properties

align

Value: String (`center`, `left`, `right`). Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `align` property controls the horizontal alignment of the table with respect to the next outermost container that provides positioning context. Most typically, the next outermost positioning container is the `body` element. Modifications to this property on an existing table cause the surrounding content to reflow on the page. Be sure you test the consequences of any modification with a variety of browser window sizes.

Example

Use The Evaluator (Chapter 13) to see the `align` property at work. The default value (`left`) is in force when the page loads. But you can shift the table to right-align with the body by entering the following statement into the top text box for IE5+ and NN6+/Mozilla/Safari:

```
document.getElementById("myTable").align = "right"
```

Related Item: `style.align` property.

background

Value: URL string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

Only IE4+ makes a provision for assigning a background image to a table, and the `background` property controls that value. You can swap out an image by assigning a new URL to the `background` property. The image appears in front of any background color assigned to the table. Thus, you can assign attributes for both characteristics so that there is at least a background color (and an image for IE users).

Example

Treat the `background` property of a table like you do the `src` property of an `img` element object. If you precache an image, you can assign the `src` property of the precached `image` object to the `background` property of the table for quick image changing. Such an assignment statement looks like the following:

```
document.all.myTable.background = imgArray["myTableAlternate"].src;
```

Related Item: `IMG.src` property.

bgColor

Value: Color value string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The bgColor attribute controls the background color of a table (the bgcolor attribute). Colors assigned to the entire table are overridden if colors are assigned to row, row groups, or cells within the table. If you set the bgColor property, the backgroundColor style property is not affected. Assign values in any acceptable color string format, such as hexadecimal triplets (for example, "#FCFC00") or the generally recognized plain-language names (for example, "cornflowerblue").

Example

Use The Evaluator (Chapter 13) to assign a color to the table. After looking at the table to see its initial state, enter the following IE5+/W3C statement into the top text box:

```
document.getElementById("myTable").bgColor = "lightgreen"
```

When you look at the table again, you see that only some of the cells turned to green. This is because colors also are assigned to table elements nested inside the outermost table element, and the color specification closest to the actual element wins the context.

Related Item: style.backgroundColor property.

border

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The border property controls the thickness of the table's borders. Values indicate the number of pixels thick the border should be. A value of zero removes all visible borders surrounding the table. Different browsers render table cell borders differently depending on background colors and other visual attributes of tables and table elements. Be sure to verify the appearance on as many browsers and operating systems as possible.

Example

To remove all traces of an outside border of a table (and, in some combinations of attributes of other table elements, borders between cells), use the following statement (in IE5+/W3C syntax):

```
document.getElementById("myTable").border = 0;
```

Related Item: borderColor property.

borderColor
borderColorDark
borderColorLight

Value: Color value string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

IE4+ provides attributes and corresponding properties to control the border colors of a table. When table borders have enough thickness to display a three-dimensional raised look, the appearance is created by generating two dark and two light edges (simulating a light source coming from the upper-left or lower-right corner). If you want to do a better job of specifying the color combinations for the light and dark edges, you can control them individually via the borderColorLight and borderColorDark properties, respectively. You can assign colors in

any valid color value (hexadecimal triplet or plain-language name); but when you read the property, the value is returned as a hexadecimal triplet (for example, "#008000").

Example

Assuming that you have set the initial light and dark color attributes of a table, the following function swaps the light and dark colors to shift the light source to the opposite corner:

```
function swapColors(tableRef) {
   var oldLight = tableRef.borderColorLight;
   tableRef.borderColorLight = tableRef.borderColorDark;
   tableRef.borderColorDark = oldLight;
}
```

Although you can easily invoke this function over and over by ending it with a `setTimeout()` method that calls this function after a fraction of a second, the results are very distracting to the person trying to read your page. Please don't do it.

Related Item: `td.borderColor` property.

caption

Value: `caption` element object reference. Read/Write (see text)
Compatibility: WinIE4+, MacIE5+, NN6+, Moz1+, Safari1+

The `caption` property returns a reference to the `caption` element object that is nested inside the current table. If there is no `caption` element, the value is `null`. You can use this property as a shortcut reference to the `caption` element if you need to read or modify that element's properties. The property is read/write in NN6+/Moz, provided you create a valid `caption` element object and assign that new object to the `caption` property.

Example

The following example, for use with The Evaluator (Chapter 13) in NN6+/Moz, demonstrates the sequence of assigning a new `caption` element object to a table. Although the table in The Evaluator already has a `caption` element, the following statements replace it with an entirely new one. Enter each of the following statements into the top text box, starting with the one that saves a long reference into a variable for multiple use at the end:

```
t = document.getElementById("myTable")
a = document.createElement("caption")
b = document.createTextNode("A Brand New Caption")
a.appendChild(b)
t.replaceChild(a, t.caption)
```

A view of the table shows that the new caption has replaced the old one because a table can have only one `caption` element.

Related Item: `caption` element object.

cellPadding
cellSpacing

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `cellPadding` property is a table-wide specification for the blank space inserted between the edge of a table cell and the content of the cell. One value affects the padding on all four sides. The effect of cell padding is especially apparent when there are borders between cells; in this case, the padding provides welcome breathing space between the border and content.

The `cellSpacing` property influences the thickness of borders between cells. If no visible borders are present between cells in a table, you can usually set either `cellpadding` or `cellspacing` to provide the desired blank space between cells.

Example

Use The Evaluator (Chapter 13) to adjust the `cellPadding` and `cellSpacing` properties of the demonstrator table. First, adjust the padding (IE5+/W3C syntax):

```
document.getElementById("myTable").cellPadding = 50
```

Now, adjust the cell spacing:

```
document.getElementById("myTable").cellSpacing = 15
```

Notice how `cellSpacing` affected the thickness of inter-cell borders.

Related Item: `border` property.

`cells`

Value: Array. Read-Only
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `cells` property returns an array (collection) of all `td` and `th` element objects within the entire table. From the perspective of the `table` element object, this "view" encompasses all cells — whether they are inside a table row segment (for example, a `thead`) or in a freestanding row. In the W3C DOM, the `cells` collection is accessible only as a property of a `tr` object. However, a `rows` collection is available from all table container elements, thus enabling you to iterate through all cells of all rows.

Example

Use The Evaluator (Chapter 13) with WinIE5+ to have JavaScript calculate the number of columns in the demonstrator table with the help of the `cells` and `rows` properties. Enter the following statement into the top text box:

```
document.all.myTable.cells.length/document.all.myTable.rows.length
```

The result is the number of columns in the table.

Related Items: `rows`, `tr.cells` properties.

`cols`

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `cols` property represents the IE-specific `cols` attribute for `table` elements. Specifying this attribute should speed table rendering. If you don't specify the attribute explicitly in your HTML, the property has a value of zero — the property does not tell you dynamically the size of your table. Although this property is read/write, you cannot use this property to add or remove columns from a table. Instead, use the table modification methods discussed later in this section.

Related Item: `rows` property.

dataPageSize

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

When using IE4+ data binding to obtain table data from a data source, there may be more rows or data (records) than you wish to display in one table. If so, you can define the number of rows (records) that constitutes a "page" of data within the table. With this limit installed for the table, you can then use the `firstPage()`, `previousPage()`, `nextPage()`, and `lastPage()` methods to access another page relative to the currently viewed page. While you usually establish this value via the `datapagesize` attribute of the `table` element, you can adjust it later via the `dataPageSize` property to show more or fewer records per "page" in the table.

Example

If you want to change the number of visible rows of linked data in the table to 15, use the following statement:

```
document.all.myTable.dataPageSize = 15;
```

Related Items: `dataSrc`, `dataFld` properties; `firstPage()`, `lastPage()`, `nextPage()`, `previousPage()` methods.

frame

Value: String constant. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `frame` property enables you to control which side or sides of the table's border should be displayed. Values for this property can be any of a fixed set of string constants. Table 37-3 lists the acceptable values. Hiding or showing table border edges under script control can have an effect on the layout and placement of both the table and surrounding elements. Note that Safari 1.0 does not change the border rendering when you change this property value.

Table 37-3: Table frame Property Values

Value	*Description*
`above`	Top edge only
`below`	Bottom edge only
`border`	All four sides (same as box)
`box`	All four sides (same as border)
`hsides`	Horizontal (top and bottom) edges only
`lhs`	Left-hand side edge only
`rhs`	Right-hand side edge only
`void`	No borders
`vsides`	Vertical (left and right) edges only

Example

Listing 37-4 presents a page that cycles through all possible settings for the `frame` property. The `frame` property value is displayed in the table's caption.

Listing 37-4: **Cycling Through Table frame Property Values**

```
<html>
   <head>
      <title>table.frame Property</title>

      <script type="text/javascript">
      var timeoutID;
      var frameValues = ["box", "above", "rhs", "below", "lhs", "hsides",
         "vsides", "border", "void"];
      function rotateBorder(i) {
         document.getElementById("myTABLE").frame = frameValues[i];
         document.getElementById("myCAPTION").innerHTML = frameValues[i];
         i = (++i == frameValues.length) ? 0 : i;
         timeoutID = setTimeout("rotateBorder(" + i + ")", 2000);
      }
      function stopRotate() {
         clearTimeout(timeoutID);
         document.getElementById("myTABLE").frame = "box";
         document.getElementById("myCAPTION").innerHTML = "box";
      }
      </script>
   </head>
   <body>
      <h1>table.frame Property</h1>
      <hr />
      <form name="controls">
         <fieldset>
            <legend>Cycle Table Edge Visibility</legend>
            <table width="100%" cellspacing="20">
               <tr>
                  <td><input type="button" value="Cycle"
                     onclick="rotateBorder(0)" /></td>
                  <td><input type="button" value="Stop"
                     onclick="stopRotate()" /></td>
               </tr>
            </table>
         </fieldset>
      </form>
      <hr />
      <table id="myTABLE" cellpadding="5" border="3" align="center">
         <caption id="myCAPTION">
            Default
         </caption>
         <thead id="myTHEAD">
            <tr>
               <th>River</th>
               <th>Outflow</th>
               <th>Miles</th>
               <th>Kilometers</th>
            </tr>
         </thead>
         <tbody>
            <tr>
               <td>Nile</td>
               <td>Mediterranean</td>
               <td>4160</td>
               <td>6700</td>
```

```
            </tr>
            <tr>
               <td>Congo</td>
               <td>Atlantic Ocean</td>
               <td>2900</td>
               <td>4670</td>
            </tr>
            <tr>
               <td>Niger</td>
               <td>Atlantic Ocean</td>
               <td>2600</td>
               <td>4180</td>
            </tr>
            <tr>
               <td>Zambezi</td>
               <td>Indian Ocean</td>
               <td>1700</td>
               <td>2740</td>
            </tr>
         </tbody>
      </table>
   </body>
</html>
```

Related Items: `border`, `borderColor`, `rules` properties.

`height`
`width`

Value: Integer or length string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `height` (IE4+) and `width` (IE4+/W3C) properties represent the `height` and `width` attributes assigned to the `table` element. If no values are assigned to the element in the tag, the properties do not reveal the rendered size of the table (use the `offsetHeight` and `offsetWidth` properties for that information). Values for these properties can be integers representing pixel dimensions or strings containing percentage values, just like the attribute values. Scripts can shrink the dimensions of a table no smaller than the minimum space required to render the cell content. Notice that only the `width` property is W3C DOM-sanctioned (as well as the corresponding property in the HTML 4.0 specification).

Example

Use The Evaluator (Chapter 13) to adjust the width of the demonstrator table. Begin by increasing the width to the full width of the page:

```
document.getElementById("myTable").width = "100%"
```

To restore the table to its minimum width, assign a very small value to the property:

```
document.getElementById("myTable").width = 50
```

If you have IE4+, you can perform similar experiments with the `height` property of the table.

Related Items: `offsetHeight`, `offsetWidth` properties.

rows

Value: Array of row objects. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `rows` property returns an array (collection) of `tr` element objects in the current table. This array includes rows in the `thead`, `tbody`, and `tfoot` row sections if the table is segmented. You can use the `rows` property to create a cross-browser script that accesses each cell of a table. Such a nested `for` loop looks like the following:

```
var oneCell;
for (var i = 0; i < tableRef.rows.length; i++) {
   for (var j = 0; j < tableRef.rows[i].cells.length; j++) {
      oneCell = tableRef.rows[i].cells[j];
      // more statements working with the cell
   }
}
```

If you want to limit the scope of the `rows` property to rows within a row segment (for example, just in the `tbody`), you can access this property for any of the three types of row segment objects.

Example

Use The Evaluator (Chapter 13) to examine the number of rows in the demonstrator table. Enter the following statement into the top text box:

```
document.getElementById("myTable").rows.length
```

In contrast, notice how the `rows` property sees only the rows within the demonstrator table's `tbody` element:

```
document.getElementById("myTbody").rows.length
```

Related Items: `tbody.rows`, `tfoot.rows`, `thead.rows` properties.

rules

Value: String constant. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

In contrast to the `frame` property, the `rules` property governs the display of borders between cells. Values for this property can be any of a fixed set of string constants. Table 37-4 lists the acceptable values. Hiding or showing table cell border edges under script control can have an effect on the layout and placement of both the table and surrounding elements. Note that Safari 1.0 does not change the rules rendering when you change this property value.

Table 37-4: Table rules Property Values

Value	*Description*
`all`	Borders around every cell
`cols`	Vertical borders between columns
`groups`	Vertical borders between column groups; horizontal borders between row groups
`none`	No borders between cells
`rows`	Horizontal borders between row groups

Example

Listing 37-5 presents a page that cycles through all possible settings for the `rules` property. The `rules` property value is displayed in the table's caption. When you run this script, notice the nice border display for this table's combination of `colgroup` and table row segment elements.

Listing 37-5: **Cycling Through Table rules Property Values**

```
<html>
   <head>
      <title>table.rules Property</title>
      <script type="text/javascript">
      var timeoutID;
      var rulesValues = ["all", "cols", "groups", "none", "rows"];
      function rotateBorder(i) {
         document.getElementById("myTABLE").rules = rulesValues[i];
         document.getElementById("myCAPTION").innerHTML = rulesValues[i];
         i = (++i == rulesValues.length) ? 0 : i;
         timeoutID = setTimeout("rotateBorder(" + i + ")", 2000);
      }
      function stopRotate() {
         clearTimeout(timeoutID);
         document.getElementById("myTABLE").rules = "all";
         document.getElementById("myCAPTION").innerHTML = "all";
      }
      </script>
   </head>
   <body>
      <h1>table.rules Property</h1>
      <hr />
      <form name="controls">
         <fieldset>
            <legend>Cycle Table Rule Visibility</legend>
            <table width="100%" cellspacing="20">
               <tr>
                  <td><input type="button" value="Cycle"
                     onclick="rotateBorder(0)" /></td>
                  <td><input type="button" value="Stop"
                     onclick="stopRotate()" /></td>
               </tr>
            </table>
         </fieldset>
      </form>
      <hr />
      <table id="myTABLE" cellpadding="5" border="3" align="center">
         <caption id="myCAPTION">
            Default
         </caption>
         <colgroup span="1">
         </colgroup>
         <colgroup span="3">
         </colgroup>
         <thead id="myTHEAD">
```

Continued

Listing 37-5 *(continued)*

```
            <tr>
               <th>River</th>
               <th>Outflow</th>
               <th>Miles</th>
               <th>Kilometers</th>
            </tr>
         </thead>
         <tbody>
            <tr>
               <td>Nile</td>
               <td>Mediterranean</td>
               <td>4160</td>
               <td>6700</td>
            </tr>
            <tr>
               <td>Congo</td>
               <td>Atlantic Ocean</td>
               <td>2900</td>
               <td>4670</td>
            </tr>
            <tr>
               <td>Niger</td>
               <td>Atlantic Ocean</td>
               <td>2600</td>
               <td>4180</td>
            </tr>
            <tr>
               <td>Zambezi</td>
               <td>Indian Ocean</td>
               <td>1700</td>
               <td>2740</td>
            </tr>
         </tbody>
      </table>
   </body>
</html>
```

Related Items: `border`, `borderColor`, `frame` properties.

summary

Value: String. Read/Write
Compatibility: WinIE6+, MacIE-, NN6+, Moz1+, Safari1+

The `summary` property represents the HTML 4.0 `summary` attribute. The text assigned to this attribute is intended for use by browsers that present a page's content through nonvisual means. For example, a browser equipped to use speech synthesis to read the page aloud can use the text of the summary to describe the table for the user.

Related Item: `caption` property.

tBodies

Value: Array of `tbody` element objects. Read-Only
Compatibility: WinIE4+, MacIE5+, NN6+, Moz1+, Safari1+

The tBodies property returns an array of all tbody elements in the table. Even if you don't specify a tbody element, every table contains an implied tbody element. Thus, to access a batch of rows of a simple table other than the thead and tfoot sections, you can use the tBodies[0] array notation. From there, you can get the rows of the table body section via the rows property.

Example

Use The Evaluator (Chapter 13) to access the tBodies array and reveal the number of rows in the one tbody segment of the demonstrator table. Enter the following statement into the top text box:

```
document.getElementById("myTable").tBodies[0].rows.length
```

Related Items: tFoot, tHead properties.

tFoot
tHead

Value: Row segment element object. Read/Write (see text)
Compatibility: WinIE4+, MacIE5+, NN6+, Moz1+, Safari1+

Each table can have (at most) one tfoot and one thead element. If you specify one of these for the table, the tFoot and tHead properties, respectively, return references to those element objects. These properties are read-only in IE, but NN6+/Moz enables you to assign valid tfoot and thead element objects to these properties in order to insert or replace the elements in the current table. The process for doing this is similar to the sequence described in the caption property. For either of these two elements, however, you have to construct the desired number of table cell objects (and row objects if you want multiple rows) for the newly created row segment object. See the discussions of these two objects for details on accessing rows and cells of the segments.

Related Items: tbody, tfoot, thead objects.

width

(See height)

Methods

createCaption()
deleteCaption()

Returns: Reference to new caption element object; Nothing.
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The createCaption() and deleteCaption() convenience methods enable you to add or remove a caption element object from the current table. When you create a new caption, the action simply inserts the equivalent of a blank caption element tag into the table element (this may not, however, be reflected in the source view of the page). You must populate the caption element with text or HTML before it appears on the page. Because the method returns a reference to the newly created object, you can use that reference to assign content to its innerHTML property or you can append a child text node.

Because a table can have only one caption element nested within, the deleteCaption() method belongs to the table element object. The method returns no value.

Example

See Listing 37-2 for an example of creating, inserting, and removing a `caption` element object from a table.

Related Item: `caption` element object.

createTFoot()
createTHead()
deleteTFoot()
deleteTHead()

Returns: Element references (create methods); Nothing.
Compatibility: WinIE4+, MacIE5+, NN6+, Moz1+, Safari1+

These four methods enable you to add or remove `tfoot` and `thead` table row section objects. When you create a `thead` or `tfoot` element, the methods return references to the newly inserted elements. But, as with `createCaption()`, these methods do nothing to display content. Instead, use the returned references to populate the row(s) of the header and footer with cells. Regardless of the number of rows associated with a `thead` or `tfoot` element, the `deleteTFoot()` and `deleteTHead()` methods remove all associated rows and return no values.

Although these methods are available in IE4, you may not have complete write access to the properties of the objects returned by the creation methods. For example, you may not be able to assign a value to the `id` property of the `tfoot` or `thead` element returned by their respective creation methods.

Example

See Listing 37-2 for an example of creating, inserting, and removing `tfoot` and `thead` elements object from a table.

Related Items: `tfoot`, `thead` element objects.

deleteRow(rowIndex)
insertRow(rowIndex)

Returns: Nothing; Reference to newly created row.
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `insertRow()` and `deleteRow()` convenience methods assist in adding `tr` elements to, and removing them from, a `table` element. Inserting a row does little more than the equivalent of inserting a pair of empty `tr` element tags into the HTML (although you may not see them in the source view of the page). It is up to the rest of your scripts to assign properties to the row and populate it with new cells (see the `insertCell()` method of the `tr` element object).

Attributes for both methods are zero-based index numbers. In the case of `insertRow()`, the number indicates the row *before* which the new row is to be inserted. To append the row to the end of the table, use `-1` as a shortcut parameter. To delete a row, use the index value for that row. Be aware that if you intend to employ `deleteRow()` to remove all rows from a table (presumably to repopulate the table with a new set), the most efficient way is to use a `while` loop that continues to remove the first row until there are no more:

```
while (tableRef.rows.length > 0) {
   tableRef.deleteRow(0);
}
```

Example

See Listing 37-2 for examples of inserting and deleting table rows.

Related Item: `td.insertCell()` method.

`firstPage()`
`lastPage()`

Returns: Nothing.
Compatibility: WinIE5+, MacIE5+, NN-, Moz-, Safari-

For tables that are bound to external data sources via IE4+ data binding, the `firstPage()` and `lastPage()` methods zoom to the first and last pages of the data, respectively. You must specify the table's data page size for the Data Source Object to know how many records to assign to a "page" of data. Note that while related methods — `nextPage()` and `previousPage()` — are available in IE4, `firstPage()` and `lastPage()` entered the picture in IE5.

Related Items: `dataPageSize`, `dataSrc`, `dataFld` properties; `nextPage()`, `previousPage()` methods.

`moveRow(sourceRowIndex, destinationRowIndex)`

Returns: Row element object.
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The IE5+ `moveRow()` convenience method enables you to move a row from one position to another within the same table. Both parameters are integer index values. The first parameter is the index of the row you want to move; the second is the index of the row to where you want to move the row. Because no movement takes place when the method is invoked, the removal of the source row does not impact the index count of the destination row. But after the method executes, the row that was in the destination row is now pushed down one row. This method returns a reference to the moved row.

You can accomplish this same functionality in W3C DOM-compatible syntax (for IE5+, NN6+, Mozilla, and Safari) via the `replaceChild()` method of the `table` element.

Example

If you want to shift the bottom row of a table to the top, you can use the shortcut reference to the last item's index value (`-1`) for the first parameter:

```
var movedRow = document.all.someTable.moveRow(-1, 0);
```

Related Item: `replaceChild()` method.

`nextPage()`
`previousPage()`

Returns: Nothing.

Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

For tables that are bound to external data sources via IE4+ data binding, the `nextPage()` and `previousPage()` methods jump ahead and back one page of the data, respectively. You must specify the table's data page size for the Data Source Object to know how many records to assign to a "page" of data. Typically, navigational buttons associated with the table invoke these methods.

table.nextPage()

Related Items: `dataPageSize`, `dataSrc`, `dataFld` properties; `firstPage()`, `lastPage()` methods.

`refresh()`

Returns: Nothing.
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

For tables that are bound to external data sources via IE4+ data binding, the `refresh()` method retrieves the current data source data for display in the table. A script can use `setTimeout()` to invoke a function that calls this method at an interval of your choice. If you frequently update the database associated with the table, this method can help keep the table up to date without requiring the client to download the entire page (and perhaps run into cache conflicts).

Related Items: `dataPageSize`, `dataSrc`, `dataFld` properties.

tbody, tfoot, and thead Element Objects

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`align`†	`deleteRow()`†	
`bgColor`†	`insertRow()`†	
`ch`	`moveRow()`†	
`chOff`		
`rows`†		
`vAlign`		

†See `table` element object.

Syntax

Accessing `tbody`, `tfoot`, and `thead` element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

Accessing `tbody` element object properties and methods:

```
(IE4+)     [window.]document.all.tableID.tBodies[i].property |
method([parameters])
(IE5+/W3C) [window.]document.getElementById("tableID").tBodies[i].property |
method([parameters])
```

Accessing `tfoot` element object properties and methods:

```
(IE4+)     [window.]document.all.tableID.tFoot.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("tableID").tFoot.property |
method([parameters])
```

Accessing `thead` element object properties and methods:

```
(IE4+)     [window.]document.all.tableID.tHead.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("tableID").tHead.property |
method([parameters])
```

Compatibility: WinIE4+, MacIE5, NN6+, Moz1+, Safari1+

About these objects

Each of these element objects represents a row grouping within a `table` element (an `HTMLTableSectionElement` in the syntax of the W3C DOM specification). A table can have only one `thead` and one `tfoot`, but it can have as many `tbody` elements as your table organization requires.

These elements share many properties and methods with the `table` element in that they all contain rows. The benefit of defining table segments is apparent if you use table rules (see the `table.rules` property earlier in this chapter) and if you wish to limit the scope of row activities only to rows within one segment. For instance, if your table has a `thead` that is to remain static, your scripts can merrily loop through the rows of only the `tbody` section without coming anywhere near the row(s) in the `thead`.

Properties

ch
chOff

Value: One-character string. Read/Write
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

The `ch` and `chOff` properties represent the optional `char` and `charoff` attributes of table row section elements in the HTML 4.0 specification. As of IE6 and NN7.1, these properties are yet to be implemented in a browser; eventually they will help align cell content within a column or column group similar to the way word processors allow for formatting features such as decimal tabs. For details on these attributes, see `http://www.w3.org/tr/REC-html40/struct/tables.html#adef-char`.

Related Items: `col`, `colgroup` objects.

vAlign

Value: String constant. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Providing the cell-oriented `vAlign` property for a table row section enables you to specify a vertical alignment to apply to all cells within that section rather than specify the `valign` attribute for each `td` element. By default, browsers render cell content with a middle vertical alignment within the cell. If you want to modify the setting for an existing table section (or assign it to a new one you create), the values must be one of the following string constants: `baseline`, `bottom`, `middle`, or `top`.

Example

Use The Evaluator (Chapter 13) to modify the vertical alignment of the content of the `tbody` element in the demonstrator table. Enter the following statement in the top text box to shift the content to the bottom of the cells:

```
document.getElementById("myTBody").vAlign = "bottom"
```

Notice that the cells of the `thead` element are untouched by the action imposed on the `tbody` element.

Related Item: `td.vAlign` property.

caption Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`align`†		
`vAlign`††		

†See `table` element object.

††See `tbody` element object.

Syntax

Accessing `caption` element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About this object

A `caption` element is a simple HTML container whose only prerequisite is that it must be nested inside a `table` element. That nesting allows the `table` element object to control insertion and removal of a `caption` element at will. You can modify the content of a `caption` element just like you do any HTML element (in DOMs that allow such modification). You can see an example of how the `table` element object uses some of its methods to create and remove a `caption` element in Listing 37-2.

The only properties that lift the `caption` element object above a mere contextual element (described in Chapter 15) are `vAlign` (IE4+) and the W3C DOM-sanctioned `align` (IE4+/NN6+/Mozilla/Safari). I describe these properties and their values for other objects in this chapter.

col and colgroup Element Objects

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`align`†		
`ch`††		
`chOff`†††		

Properties	Methods	Event Handlers
Span		
vAlign††		
width		

†See `table` element object.

††See `tbody` element object.

Syntax

Accessing `col` and `colgroup` element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About these objects

The purpose of the `col` and `colgroup` elements is to allow cells within one or more columns to be treated as a single entity for purposes of stylesheet and other style-related control. In other words, if you want one column of a table to be all boldface, you can assign that stylesheet rule to the `col` element that encompasses that column. All cells within that column inherit the stylesheet rule definition. Having two different element names allows for the nesting of column groups, which can come in handy for complex tables. For instance, consider a table that reports the forecasted and actual sales for a list of products across four quarters of a year. The left column of the table stands alone with the product item numbers. To the right is one large grouping of eight columns that encompasses the four pairs of forecasted/actual sales pairs. All eight columns of cells are to be formatted with a particular font style to help differentiate the pairs of columns for each quarter. You also want to assign a different background color. Therefore, you designate each pair of columns as its own subgroup within the eight-column master grouping. The `colgroup` and `col` tags for this nine-column table are as follows:

```
<col id="productIDs">
<colgroup id="fiscalYear" span="8" width="40">
   <col id="Q1" span="2">
   <col id="Q2" span="2">
   <col id="Q3" span="2">
   <col id="Q4" span="2">
</colgroup>
```

Up in the `head` section of this document are stylesheet rules similar to the following:

```
<style type="text/css">
#productIDs {font-weight:bold}
#fiscalYear {font-family: Courier, "Courier New", monospace}
#Q1 {background-color: lightyellow}
#Q2 {background-color: pink}
#Q3 {background-color: lightblue}
#Q4 {background-color: lightgreen}
</style>
```

The HTML code for the column groups demonstrates the two key attributes: `span` and `width`. Both of these attributes are reflected as properties of the objects, and I describe them in the following section. Notice, however, that `col` and `colgroup` elements act cumulatively and in source code order to define the column groups for the table. In other words, if the style of the left-hand column is not important, the table still requires the initial one-column `col` element before the eight-column `colgroup` element. Otherwise, the browser makes the first eight columns the column group. Therefore, it is a good idea to account for every column with `col` and/or `colgroup` elements if you intend to use any column grouping in your table.

From a scripter's point of view, you are more likely to modify styles for a column or column group than you are to alter properties such as `span` or `width`. But, if your scripts generate new tables, you may create new `col` or `colgroup` elements whose properties you definitely should initialize with values.

Properties

span

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `span` property represents the number of columns that the column group should encompass. Don't confuse this property with the `colSpan` property of `td` and `th` elements. A `col` or `colgroup` `span` does not have any impact on the rendering or combination of multiple cells into one. It simply draws an imaginary lasso around as many columns as are specified, signifying that these columns can be treated as a group for style purposes (and also for drawing of divider rules, if you set the table's `rules` property to `groups`).

Example

The following statement assigns a `span` of `3` to a newly created `colgroup` element stored in the variable `colGroupA`:

```
colGroupA.span = 3;
```

Related Item: `width` property.

width

Value: Length string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The only reason the `width` property is highlighted for these objects is that the property (and corresponding attribute) impacts the width of table cells inside the scope of the column grouping. For example, if you assign a width of 50 pixels to a `colgroup` whose `span` attribute is set to `3`, all cells in all three columns inherit the 50-pixel width specification. For more details on the values acceptable to this property, see the `table.width` property description earlier in this chapter.

Related Item: `table.width` property.

tr Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	Methods	Event Handlers
Align†	deleteCell()	
bgColor†	insertCell()	
borderColor†		
borderColorDark†		
borderColorLight†		
Cells		
ch††		
chOff††		
height		
rowIndex		
sectionRowIndex		
vAlign††		

†See `table` element object.

††See `tbody` element object.

Syntax

Accessing `tr` element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE4+)     [window.]document.all.tableID.rows[i].property | method([parameters])
(IE4+)     [window.]document.all.tableRowSectionID.rows[i].property |
method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID"). property |
method([parameters])
(IE5+/W3C) [window.]document.getElementById("tableID").rows[i].property |
method([parameters])
(IE5+/W3C)
[window.]document.getElementById("tableRowSectionID").rows[i].property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About this object

Table rows are important objects within the complex nesting of table-related elements and objects. When a table represents server database data, one row usually equals one record. And, although you can employ scripting to add columns to a table, the more common table modifications are to add or delete rows — hence the presence of the `table` element object's `insertRow()` and `deleteRow()` methods.

The primary job of the `tr` element is to act as a container for `td` elements. All the cells in a row inherit some attributes and properties that you apply to that row. An array of cell objects is available for iteration via `for` loops. A `tr` element object, therefore, also has methods that insert and remove individual cells in that row.

The number of columns in a row is determined by the number of `td` elements or, more specifically, by the number of columns that the cells intend to span. One row can have four `td` elements, while the next row can have only two `td` elements — each of which is defined to

occupy two columns. The row of the table with the most `td` elements and column reservations determines the column width for the entire table.

Of the properties just listed, the ones related to border color are available in IE4+ only. In IE4+, the border is drawn around each cell of the row rather than the entire row. The HTML 4.0 specification (and the W3C DOM Level 2 specification by extension) does not recognize border colors for rows alone, nor are stylesheet border rules inherited by the cell children of a row. However, you can define borders for individual cells or classes of cells.

Properties

cells

Value: Array of `td` element objects. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `cells` property returns an array (collection) of `td` element objects nested inside the current `tr` object. The `length` property of this array indicates the number of actual `td` elements in the row, which may not be the number of columns if one or more cells occupy multiple columns.

Use the `cells` property in `for` loops to iterate through all cells within a row. Assuming your script has a reference to a single row, the loop should look like the following:

```
for (var i = 0; i < rowRef.cells.length; i++) {
   oneCell = rowRef.cells[i];
   // more statements working with the cell
}
```

Example

Use The Evaluator (Chapter 13) to retrieve the number of `td` elements in the second row of the demonstrator table. Enter the following statement into the top text box (W3C DOM syntax shown here):

```
document.getElementById("myTable").rows[1].cells.length
```

Related Items: `table.rows`, `td.cellIndex` properties.

height

Value: Integer or length string. Read/Write
Compatibility: WinIE5+, MacIE5+, NN-, Moz-, Safari-

IE5+ enables page authors to predefine a height for a table row; this attribute is echoed by the `height` property. The value can be a number of pixels or a percentage length value. Note that this property does not reveal the rendered height of the row unless you explicitly set the attribute in the HTML. To get the actual height (in IE5+ and NN6+), use the `offsetHeight` property. You cannot adjust the `height` property to be smaller than the table normally renders the row.

Example

Use The Evaluator (Chapter 13) in IE5+ to expand the height of the second row of the demonstrator table. Enter the following statement into the top text box:

```
document.all.myTable.rows[1].height = 300
```

If you attempt to set the value very low, the rendered height goes no smaller than the default height.

Related Item: `offsetHeight` property (Chapter 15).

`rowIndex`
`sectionRowIndex`

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Each row occupies a position within the collection of rows in the table as well as within the collection of rows for a table section (`thead`, `tbody`, or `tfoot`). The `rowIndex` property returns the zero-based index value of the row inside the `rows` collection for the entire table, regardless of table section composition. In contrast, the `sectionRowIndex` property returns the zero-based index value of the row inside its row section container. If the table has no row sections defined for it, a single, all-encompassing `tbody` element is assumed; in this case, the `sectionRowIndex` and `rowIndex` values are equal.

These properties serve in functions that are passed a reference to a row. However, the functions might also need to know the position of the row within the table or section. Although there is no `tr` object property that returns a reference to the next outermost table row section or the table itself, the `parent` and parent's `parent` elements, respectively, can reference these objects.

Example

Use The Evaluator (Chapter 13) to explore the `rowIndex` and `sectionRowIndex` property values for the second physical row in the demonstrator table. Enter each of the following statements into the top text box (W3C DOM syntax shown here):

```
document.getElementById("myTable").rows[1].rowIndex
document.getElementById("myTable").rows[1].sectionRowIndex
```

The result of the first statement is `1` because the second row is the second row of the entire table. But the `sectionRowIndex` property returns `0` because this row is the first row of the `tbody` element in this particular table.

Related Items: `table.rows`, `tbody.rows`, `tfoot.rows`, `thead.rows` properties.

Methods

`deleteCell(cellIndex)`
`insertCell(cellIndex)`

Returns: Nothing; Reference to new cell.
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The act of inserting a row into a table is not complete until you also insert cells into the row. The `insertCell()` method does just that, with a parameter indicating the zero-based index of the cell's position among other cells in the row. A value of `-1` appends the cell to the end of existing cells in the row.

When you invoke the `insertCell()` method, it returns a reference to the new cell. This gives you the opportunity to adjust other properties of that cell before moving onto the next cell. For example, if you want to insert a cell that has a column span of `2`, you adjust the `colSpan` property of the cell whose reference just returned, as in the following:

tr.deleteCell()

```
var oneCell = tableRowRef.insertCell(-1);
oneCell.colSpan = 2;
```

Scripts that add rows and cells must make sure that they add the identical number of cells (or cell column spaces) from one row to the next. Otherwise, you have an unbalanced table with ugly blank spaces where you probably don't want them.

To remove a cell from a row, use the `deleteCell()` method. The parameter is a zero-based index value of the cell you want to remove. If all you want to do is replace the content of a cell, apply the new content to the `innerHTML` property of the `td` element. This is smoother and safer than deleting and reinserting a cell because any execution error that occurs in the process results in an unbalanced table. Finally, to rid yourself of all cells in a row, use the `deleteRow()` method of the `table` and table row section element objects.

Example

See Listing 37-2 for an example of inserting cells during the row insertion process.

Related Item: `table.insertRow()` method.

td and th Element Objects

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
abbr		
align†		
axis		
background†		
bgColor†		
borderColor†		
borderColorDark†		
borderColorLight†		
cellIndex		
ch††		
chOff††		
colSpan		
headers		
height		
noWrap		
rowSpan		
vAlign††		
width		

†See `table` element object.

††See `tbody` element object.

Syntax

Accessing `td` and `th` element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE4+)     [window.]document.all.tableID.cells[i].property |
method([parameters])
(IE4+)     [window.]document.all.tableRowSectionID.cells[i].property |
method([parameters])
(IE4+)     [window.]document.all.tableRowID.cells[i].property |
method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID"). property |
method([parameters])
(IE5+/W3C) [window.]document.getElementById("tableID").cells[i].property |
method([parameters])
(IE5+/W3C)
[window.]document.getElementById("tableRowSectionID").cells[i].property |
method([parameters])
(IE5+/W3C) [window.]document.getElementById("tableRowID").rows[i].property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About these objects

`td` (table data) and `th` (table header) elements create cells within a table. By common convention, a `th` element is rendered in today's browsers with a distinctive style — usually with a bold font and center alignment — but the context as a table header is a key ingredient of the table. A table cell is as deeply nested as you can get with table-related elements.

Properties of cells that are delivered in the HTML of the page are rarely modified (with the exception of the `innerHTML` property). But you still need full access to properties of cells if your scripts add rows to a table dynamically. After creating each blank table cell object, your scripts can adjust `colSpan`, `rowSpan`, `noWrap`, and other properties that influence the characteristics of that cell within the table.

See the beginning of this chapter for discussions and examples of how to add rows of cells and modify cell content under script control.

Properties

abbr
axis
headers

Value: See text. Read/Write
Compatibility: WinIE6+, MacIE5+, NN6+, Moz1+, Safari1+

These three properties are defined for table cell element objects in the W3C DOM. They all represent attributes for these elements in the HTML 4.0 specification. The purposes of these attributes and properties are geared toward browsers that provide alternate means of rendering content, such as through speech synthesis. Although these properties are definitely valid for W3C browsers, they have no practical effect. For general application, you can ignore these properties — consider them reserved for future use.

cellIndex

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `cellIndex` property returns an integer indicating the zero-based count of the current cell within its row. Thus, if a script is passed a reference to a cell, the `cellIndex` property reveals its position within the row. Inserting or deleting cells in the row at lower index values influences the `cellIndex` value after the alteration.

Example

You can rewrite the cell addition portion of Listing 37-2 to utilize the `cellIndex` property. The process entails modifying the `insertTableRow()` function so that it uses a `do...while` construction to keep adding cells to match the number of data slots. The function looks like the following (changes shown in boldface):

```
function insertTableRow(form, where) {
   var now = new Date();
   var nowData = [now.getHours(), now.getMinutes(), now.getSeconds(),
      now.getMilliseconds()];
   clearBGColors();
   var newCell;
   var newRow = theTableBody.insertRow(where);
   var i = 0;
   do {
      newCell = newRow.insertCell(i);
      newCell.innerHTML = nowData[i++];
      newCell.style.backgroundColor = "salmon";
   } while (newCell.cellIndex < nowData.length)
   updateRowCounters(form);
}
```

This version is merely for demonstration purposes and is not as efficient as the sequence shown in Listing 37-2. But the `cellIndex` property version can give you some implementation ideas for the property. It also shows how dynamic the property is, even for brand new cells.

Related Item: `tr.rowIndex` property.

colSpan
rowSpan

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `colSpan` and `rowSpan` properties represent the `colspan` and `rowspan` attributes of table cell elements. Assign values to these properties only when you are creating new table rows and cells — and you are firm in your table cell design. If you fail to assign the correct values to either of these properties, your table cell alignment will get out of whack. Modifying these property values on an existing table is extremely risky unless you are performing other cell manipulation to maintain the balance of rows and columns. Values for both properties are integers greater than or equal to 1.

Example

Use The Evaluator (Chapter 13) to witness how modifying either of these properties in an existing table can destroy the table. Enter the following statement into the top text box:

```
document.getElementById("myTable").rows[1].cells[0].colSpan = 3
```

Now that the first cell of the second row occupies the space of three columns, the browser has no choice but to shift the two other defined cells for that row out beyond the original boundary of the table. Experiment with the `rowSpan` property the same way. To restore the original settings, assign `1` to each property.

Related Item: `col.span` property.

height
width

Value: Integer and length string. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Table cells may be specified to be larger than their default rendered size. This usually happens in the `height` and `width` attributes of the cell. Settings of the `width` attribute of a `col` or `colgroup` element (IE4+ and NN6+/W3C) may also govern the width of a cell. A cell's height can be inherited from the `height` attribute setting of a table row or row section (IE4+). Both `height` and `width` attributes are deprecated in HTML 4.0 in favor of the `height` and `width` stylesheet attributes. That said, the `height` and `width` properties of a table cell echo only the settings of the explicit attributes in the cell's tag. If a stylesheet in the element tag governs a cell's dimensions, visit the cell object's `style` property to determine the dimensions. Explicit attributes override stylesheet rules.

Values for these two properties are length values. These can be pixel integers or percentage values as strings. Attempts to set the sizes smaller than their default rendered size results in a cell of default size. Also be aware that enlarging a cell affects the width of the entire column and/or height of the entire row occupied by that cell.

Example

Use The Evaluator (Chapter 13) to see the results of setting the `height` and `width` properties of an existing table cell. Enter each of the following statements into the top text box and study the results in the demonstrator table (W3C DOM syntax used here):

```
document.getElementById("myTable").rows[1].cell[1].height = 100
document.getElementById("myTable").rows[2].cell[0].width = 300
```

You can restore both cells to their original sizes by assigning very small values, such as `1` or `0`, to the properties. The browser prevents the cells from rendering any smaller than is necessary to show the content.

Related Items: `col.width`, `tr.height` properties.

noWrap

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The default behavior of a table cell is to wrap text lines within the cell if the text would extend beyond the right edge of the cell as calculated from the width of the entire table. But you can force the table to be wider to accommodate the text in an unwrapped line of text by setting the `noWrap` property (or `nowrap` attribute) of the cell to `true`. The `nowrap` attribute is deprecated in HTML 4.0.

Example

The following statement creates a new cell in a row and sets its `noWrap` property to prevent text from word-wrapping inside the cell:

td.noWrap

```
newCell = newRow.insertCell(-1);
newCell.noWrap = true;
```

You need to set this property only if the cell must behave differently than the default, word-wrapping style.

rowSpan

(See `colSpan`)

width

(See `height`)

ol Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`compact`		
`start`		
`type`		

Syntax

Accessing `ol` element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID"). property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About this object

The `ol` (ordered list) element is a container of `li` (list item) elements. An *ordered list* means that the list items have a sequence and are preceded by a number or letter to signify the position within the sequence. The few element-specific attributes are being deprecated in favor of stylesheet definitions. For the sake of backward compatibility with existing content, however, it is likely that many future generations of browsers will continue to support these deprecated attributes. These attributes are therefore available as properties of the element object.

Most of the special appearance of a list (notably indentation) is handled automatically by the browser's interpretation of how an ordered list should look. You have control over the numbering or lettering schemes and the starting point for those sequences. With CSS you can significantly override the browser's formatting — even eliminating the traditional list appearance via other choices for the display `style` property.

Properties

compact

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Although the property is defined for the browsers just shown, the `compact` property (and the deprecated attribute it echoes) has no impact on the density of the listing.

start

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `start` property governs which number or letter begins the sequence of leading characters for nested `li` items. If the `type` attribute specifies numbers, the corresponding number is used; if it specifies letters, the letter of the alphabet corresponding to the number becomes the starting character. You can change the numbering in the middle of a sequence via the `li.value` property.

It is an extremely rare case that requires you to modify this property for an existing `ol` element. But if your script is creating a new element for a segment of ordered list items that has some other content intervening from an earlier `ol` element, you can use the property to assign a starting value to the `ol` group.

Example

The following statements generate a new `ol` element and assign a value to the `start` property:

```
var newOL = document.createElement("ol");
newOL.start = 5;
```

Related Items: `type`, `li.value` properties.

type

Value: String constant. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

An `ol` element can use any of five different numbering schemes. Each scheme has a type code, whose value you can use for the `type` property. The following table shows the property values and examples:

Value	*Example*
A	A, B, C, ...
a	a, b, c, ...
I	I, II, III, ...
i	i, ii, iii, ...
1	1, 2, 3, ...

The default value is 1. You are free to adjust the property after the table has rendered, and you can even stipulate a different type for specific li elements nested inside (see the li.type property). If you want to have further nesting with a different numbering scheme, you can nest the ol elements and specify the desired type for each nesting level, as shown in the following HTML example:

```
<ol type="A">
   <li>One</li>
   <li>Two</li>
   <li>Three</li>
      <ol type="a">
         <li>Sub One</li>
         <li>Sub Two</li>
         <li>Sub Three</li>
      </ol>
   <li>Four</li>
</ol>
```

Indenting the HTML is optional, but it may help you to keep the nesting straight.

Example

The following statements generate a new ol element and assign a value to the type property so that the sequence letters are uppercase Roman numerals:

```
var newOL = document.createElement("ol");
newOL.type = "I";
```

Related Items: start, UL.type, LI.type properties.

ul Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
compact†		
type		

†See ol element object.

Syntax

Accessing ul element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About this object

The ul (unordered list) element is a container of li (list item) elements. An *unordered list* means that the list items have no sequence and are preceded by symbols that don't signify any

particular order. The few element-specific attributes are being deprecated in favor of stylesheet definitions. For the sake of backward compatibility with existing content, however, it is likely that many future generations of browsers will continue to support these deprecated attributes. These attributes are therefore available as properties of the element object.

Most of the special appearance of a list (notably indentation) is handled automatically by the browser's interpretation of how an ordered list should look. You have control over the three possible characters that precede each item. With CSS you can significantly override the browser's formatting — even eliminating the traditional list appearance via other choices for the display `style` property.

Properties

type

Value: String constant. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

A `ul` element can use any of three different leading characters. Each character type has a type code whose value you can employ for the `type` property. Property values are `circle`, `disc`, and `square`. The difference between a `circle` and `disc` is that the `circle` is unfilled, while the `disc` is solid. The default value is `disc`.

Example

The following statements generate a new `ul` element and assign a value to the `type` property so that the bullet characters are empty circles:

```
var newUL = document.createElement("ul");
newUL.type = "circle";
```

Related Items: `OL.type`, `UL.type` properties.

li Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`type`		
`value`		

Syntax

Accessing `li` element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID").property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

li

About this object

An li (list item) element contains the HTML that is displayed for each item within an ol or ul list. Note that you can put any HTML you want inside a list item, including images. Attributes and properties of this element enable you to override the specifications declared in the ol or ul containers (except in MacIE).

Properties

type

Value: String constant. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

Because either an ol or ul container can own an li element, the type property accepts any of the values that you assign to the type properties of both the ol and ul element objects. See the ol.type and ul.type properties earlier in this chapter for lists of those values.

Exercise caution, however, if you attempt to mix and match types. For example, if you try to set the li.type property of an li element to circle inside an ol element, the results vary from browser to browser. NN6+/Moz, for example, follows your command; however, IE may display some other characters.

Example

See the examples for the ol.type and ul.type properties earlier in this chapter.

Related Items: ol.type, ul.type properties.

value

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The value property governs which number or letter is used for the current list item inside an ordered list. Employ this attribute and property to override the natural progression. Because these sequence characters can be letters, numbers, or Roman numerals, the integer you specify for this property is converted to the numbering scheme in force by the li or ol element's type property.

Example

The following statements generate a new li element and assign a value to the start property:

```
var newLI = document.createElement("li");
newLI.start = 5;
```

Related Item: ol.start property.

dl, dt, and dd Element Objects

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	Methods	Event Handlers
compact†		

†See ol element object.

Syntax

Accessing dl, dt, and dd element object properties and methods:

```
(IE4+)       [window.] document.all.elemID.property | method([parameters])
(IE5+/W3C)   [window.] document.getElementById("elemID"). property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About these objects

Three elements — dl, dt, and dd — provide context and (optionally) formatting for definitions in a document. The dl element is the outer wrapper signifying a definition list. Each definition term should be inside a dt element, while the definition description should be in the nested dd element. The HTML for a simple definition list has the following structure:

```
<dl>
   <dt>First term</dt>
   <dd>First term's definition</dd>
   <dt>Second  term</dt>
   <dd>Second  term's definition</dd>
</dl>
```

Although there are no specific requirements for rendering definition lists by convention, the term and description are usually on different lines with the description indented. With CSS you can significantly override the browser's formatting — even eliminating the traditional list appearance via other choices for the display style property.

All three of these elements are treated as element objects, sharing the same properties, methods, and event handlers of generic element objects. The only one of the three that has anything special is the dl element, which has a compact property. WinIE4+ does respond to this attribute and property by putting the description and term on the same line if the term is shorter than the usual indentation space of the description.

dir and menu Element Objects

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	Methods	Event Handlers
compact†		

†See ol element object.

Syntax

Accessing `dir` and `menu` element object properties and methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/W3C) [window.]document.getElementById("elemID"). property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

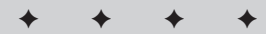
About these objects

The `dir` and `menu` elements are treated in modern browsers as if they were `ul` elements for unordered lists of items. Both elements are deprecated in HTML 4.0; yet, because they are acknowledged in that standard, they are also acknowledged in the W3C DOM (and the IE DOM, too). Originally intended to assist in creating single and double columns of text (long since supplanted by tables), usage of these elements has fallen out of favor and is discouraged.

✦ ✦ ✦

CHAPTER 38

The Navigator and Other Environment Objects

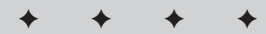


In This Chapter

Determining which browser the user has

Branching scripts according to the user's operating system

Detecting plug-in support



Client-side scripting primarily focuses on the document inside a browser window and the content of the document. As discussed in Chapter 16, the window, too, is an important part of how you apply JavaScript on the client. But stepping out even one more level is the browser application itself. Scripts sometimes need to know about the browser and the computing environment in which it runs so that they can tailor dynamic content for the current browser and operating system.

To that end, browsers provide objects that expose as much about the client computer and the browser as is feasible within accepted principles of preserving a user's privacy. In addition to providing some of the same information that CGI programs on the server receive as environment variables, these browser-level objects in some browsers also include information about how well equipped the browser is with regard to plug-ins and Java. Another object defined for version 4 browsers and on reveals information about the user's video monitor, which may influence the way your scripts calculate information displayed on the page.

The objects in this chapter don't show up on the document object hierarchy diagrams, except as free-standing groups (see Appendix A). The IE4+ object model, however, incorporates these environmental objects as properties of the `window` object. Because the `window` reference is optional, you can omit it and wind up with a cross-browser, compatible script in many cases.

Although these objects were first implemented outside of the object model hierarchy, they are now treated as belonging to the `window` object. As you learn in this chapter, the IE for Windows methodology can be a bit roundabout. And yet the Macintosh version of IE5 adopted the approach initiated by NN3. Go figure.

clientInformation Object (IE4+) and navigator Object (All)

Properties	Methods	Event Handlers
appCodeName	javaEnabled()	
appMinorVersion	preference()	
appName	taintEnabled()	
appVersion		
browserLanguage		
cookieEnabled		
cpuClass		
language		
mimeTypes		
onLine		
oscpu		
platform		
plugins		
product		
productSub		
securityPolicy		
systemLanguage		
userAgent		
userLanguage		
userProfile		
vendor		
vendorSub		

Syntax

Accessing `clientInformation` and `navigator` object properties and methods:

```
(All)         navigator.property | method()
(IE4+/NN6+)   [window.]navigator.property | method()
(IE4+/NN6+)   [window.]clientInformation.property | method()
```

About this object

In Chapter 16, I repeatedly mention that the `window` object is the top banana of the document object hierarchy. In other programming environments, you likely can find a level higher than the window — perhaps referred to as the *application level*. You may think that an object known as the `navigator` object is that all-encompassing object. That is not the case, however.

Although Netscape originally invented the navigator object for the Navigator 2 browser, Microsoft Internet Explorer also supports this object in its object model. For those who exhibit partisan feelings toward Microsoft, IE4+ provides an alternate object—clientInformation—that acts as an alias to the navigator object. You are free to use the IE-specific terminology if your development is intended only for IE browsers. All properties and methods of the navigator and clientInformation objects are identical. In the rest of this section, all references to the navigator object also apply to the clientInformation object.

Be aware that the number of properties for this object has grown with virtually every browser version. Moreover, other than some basic items that have been around since the early days, most of the more recent properties are browser-specific. Observe the compatibility ratings for each of the following properties very carefully.

Most of the properties of the navigator object deal with the browser program the user runs to view documents. Properties include those for extracting the version of the browser and the platform of the client running the browser. Because so many properties of the navigator object are related to one another, I begin this discussion by grouping four of the most popular ones together.

Properties

appCodeName
appName
appVersion
userAgent

Value: String. Read-Only
Compatibility: WinIE3+, MacIE3+, NN2+, Moz1+, Safari1+

These four properties reveal just about everything that browser-sniffing code needs to know about the user's browser brand, version, and other tidbits. Of these four, only the last three are particularly valuable. The first property in the list, appCodeName, defines a class of client that encompasses essentially every standard browser. The value returned by browsers, Mozilla, is the code name of the first browser engine on which NN and IE browsers at one time were based (the NCSA Mosaic browser). This information does nothing to help your scripts distinguish among browser flavors, so you can ignore the property. But the other three properties are the ones with all the goodies.

The appName property returns the official name for the browser application. For Netscape and Mozilla browsers, the appName value is Netscape; for Internet Explorer, the value is Microsoft Internet Explorer. The situation is murkier for other browsers. For example, Opera gives users a preference option to have the browser identify itself as IE, NN, or Opera—and the appName property value changes accordingly. And Safari 1.0 identifies itself as Netscape in the appName property. Thus, the appName property is no longer a reliable browser brand determinant.

The appVersion and userAgent properties provide more meaningful detail. I start with the appVersion property because it is revealing and, at times, misleading.

Using the appVersion property

A typical `appVersion` property value looks like the following (one from NN7, one from IE6):

```
5.0 (Windows; en-US)
4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.0.3705)
```

Because most version decisions are based on numeric comparisons (for example, the version is equal to or greater than 4), you frequently need to extract just the number part of the string returned by the `appVersion` property. The cleanest way to do this is via the `parseInt()` or `parseFloat()` methods. Use

```
parseInt(navigator.appVersion)
```

if you are interested only in the number to the left of the decimal; to get the complete leading floating-point number, use

```
parseFloat(navigator.appVersion)
```

All other characters after the leading numbers are ignored.

Also notice that the number does not always accurately represent the version of the browser at hand. For instance, IE6 reports that it is version 4.0. The number is more indicative of a broad generation number rather than a specific browser version number. In other words, the browser exhibits characteristics of the first browsers to wear the `appVersion` of 4 (IE 4.0, it turns out). Although this means that IE6 can use everything that is in the language and object model of IE4, this obviously doesn't help your script to know if the browser is capable of IE6 scripting features.

At the same time, however, buried elsewhere in the `appVersion` string is the wording `MSIE 6.0`—the "true" version of the browser. IE uses this technique to distinguish the actual version number from the generational number. Therefore, for IE, you may have to dig deeper by using string methods such as `indexOf()` to see if the `appVersion` contains the desired string. For example, to see if the browser is a variant of IE6, you can test for just `"MSIE 6"` as follows:

```
var isIE6x = navigator.appVersion.indexOf("MSIE 6") != -1
```

There is a hazard in doing this kind of testing, however. Going forward, your code will break if future versions of IE have larger version numbers. Therefore, if you want to use IE6 features with an IE8 browser (assuming such a browser becomes available), your testing for the presence of `"MSIE 6"` fails and the script thinks that it cannot use IE6 features even though they most certainly would be available in IE8. To find out if the current IE browser is the same or newer than a particular version, you must use JavaScript string parsing to deal with the `MSIE x.x` substring of the `appVersion` (or `userAgent`) property. The following example shows one function that extracts the precise IE version name and another function that confirms whether the version is at least IE6.0 for Windows.

```
var ua = navigator.userAgent;
function getIEVersion() {
   var IEOffset = ua.indexOf("MSIE ");
   return parseFloat(ua.substring(IEOffset + 5, ua.indexOf(";", IEOffset)));
}
function qualifyBrowser() {
   var qualified = false;
   if (navigator.appName == "Microsoft Internet Explorer") {
      if (parseInt(getIEVersion()) >= 6) {
         if (ua.indexOf("Windows") != -1) {
            qualified = true;
         }
```

```
      }
   }
   if (!qualified) {
      var msg = "These scripts are currently certified to run on:\n";
      msg += " - MS Internet Explorer 6.0 or later for Windows\n";
      alert(msg);
   }
   return qualified;
}
```

As clever as the preceding code looks, using it assumes that the version string surrounding the `MSIE` characters will be immutable in the future. We do not have that kind of guarantee, so you have to remain vigilant for possible changes in future versions.

Thus, with each browser generation's pollution of the `appVersion` and `userAgent` properties, the properties become increasingly less useful for browser sniffing — unless you wish to burden your code with a lot of general-purpose sniffing code, very little of which any one browser uses.

Even NN is not free of problems. For example, the main numbering in the `appVersion` property for NN7 is `5` (in other words, the fifth generation of the original Mozilla code). A potentially thornier problem arises due to Netscape's decision to eliminate some nonstandard NN4 DOM features from the NN6 DOM (`layer` objects and some event object behaviors). Many scripters followed the previously recommended technique of "prepare for the future" by using an `appVersion` of `4` as a minimum:

```
var isNN4 = parseInt(navigator.appVersion) >= 4;
```

But any code that relies on the `isNN4` variable to branch to code that talks to the dead-end NN4 objects and properties breaks when it runs in NN6+.

The bottom-line question is, "What do I do for browser version detection?" Unfortunately, there are dozens of answers to that question, depending on what you need browser detection to do and what level of code you produce.

At one end of the spectrum is code that tries to be many things to many browsers, implementing multiple levels of features for many different generations of browser. This is clearly the most difficult tactic, and you have to create quite a long list of variables for the conditions for which you establish branches. Some branches may work on one combination of browsers, while you may need to split other branches differently because the scripted features have more browser-specific implementations.

At the other end of the spectrum is the code that tries to support, say, only IE5+ and other modern browsers with W3C DOM-compatible syntax to the extent that both browser families implement the object model features. Life for this scripter is much easier in that the amount of branching is little or none depending on what the scripts do with the objects.

Between these two extremes, situations call for many different solutions. Object detection (for example, seeing if `document.images` exists before manipulating image objects) is a good solution at times, but not so much for determining the browser version as for knowing whether some code that addresses those objects works. As described in Chapter 14, it is hazardous to use the existence of, say, `document.all` as an indicator that the browser is IE4+. Some other browser in the future may also implement the `document.all` property, but not necessarily all the other IE4+ objects and syntax. Code that thinks it's running in IE4+ just because `document.all` exists can easily break if `document.all` is implemented in another browser but not all the rest of the IE4+ DOM. Using object detection to branch code that addresses the detected objects is, however, very desirable in the long run because it frees your code from getting trapped in the ever-changing browser version game.

Don't write off the `appVersion` and `userAgent` properties entirely. The combination of features that you script may benefit from some of the data in that string, especially when the decisions are made in concert with the `navigator.appName` property. A number of other properties can also provide the sufficient clues for your code to perform the branching that your application needs. For instance, it may be very helpful to your scripts to know whether the `navigator.platform` property informs them that they are running in a Windows or Macintosh environment because of the way each operating system renders fonts.

userAgent property details

The string returned by the `navigator.userAgent` property contains a more complete rundown of the browser. The `userAgent` property is a string similar to the `USER_AGENT` header that the browser sends to the server at certain points during the connection process between client and server.

Unfortunately, there is no standard for the way information in the `userAgent` property is formatted. It may be instructive, however, to view what kinds of values come from a variety of browsers on different platforms. Table 38-1 shows some of the values that your scripts are likely to see. This table does not include, of course, the many values that are not reflected by browsers that do not support JavaScript. The purpose of the table is to show you just a sampling of data that the property can contain from a variety of browsers and operating systems (particularly enlightening if you do not have access to Macintosh or UNIX computers).

Table 38-1: Typical navigator.userAgent Values

navigator.userAgent	*Description*
`Mozilla/5.0 (Macintosh; U; PPC Mac OS X X Mach-O; en-US; rv:1.4) Gecko/2003624`	Mozilla 1.4 for MacOS X
`Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.4 Gecko/2003624 Netscape/7.1 (ax)`	Navigator 7.1 for Windows, running under Windows XP Home Edition
`Mozilla/4.74 [en] (X11; U; Linux 2.2.154mdksmp i686)`	Navigator 4.74, English edition for Linux with U.S. encryption
`Mozilla/4.73 (Macintosh; U; PPC)`	Navigator 4.73 for PowerPC Macintosh with U.S. encryption
`Mozilla/4.02 [en] (Win95; I; Nav)`	Navigator-only version of Communicator 4.02, English edition for Windows 95, and export encryption
`Mozilla/4.01 [fr] (Win95; I)`	Navigator 4.01, French edition for Windows 95, export encryption
`Mozilla/3.01Gold (Win95; I)`	Navigator 3.01 Gold for Windows 95
`Mozilla/3.01 (Macintosh; I; PPC)`	Navigator 3.01 for PowerPC Macintosh
`Mozilla/3.01 (X11; I; HP-UX A.09.05 9000/720)`	Navigator 3.01 for HP-UX on RS-9000
`Mozilla/3.01 (X11; I; SunOS 5.4 sun4m)`	Navigator 3.01 for SunOS 5.4
`Mozilla/3.01Gold [de] (Win16; I)`	Navigator 3.01, German edition for Windows 3.0*x*
`Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.0.3705)`	IE 6.0 running under Windows XP Home Edition
`Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)`	IE 5.5 running under Windows NT 5.0

navigator.userAgent	Description
`Mozilla/4.0 (compatible; MSIE 5.0; Windows 98; DigExt)`	IE 5.0 for Windows 98 with digital signature
`Mozilla/4.0 (compatible; MSIE 5.0; Mac_PowerPC)`	IE 5.0 running on a PowerPC-equipped Macintosh
`Mozilla/3.0 WebTV/1.2 (compatible; MSIE 2.0)`	IE 2 built into a WebTV box, emulating Navigator 3 (its scripting compatibility with Navigator 3 is in question)
`Mozilla/2.0 (compatible; MSIE 3.0; 3.0; Windows 3.1)`	IE 3 (version for America Online software `AOL` version 3) for Windows 3.1, emulating Navigator 2
`Mozilla/2.0 (compatible; MSIE 3.02; Update a; Windows 95)`	IE 3.02, Update a for Windows 95, emulating Navigator 2
`Mozilla/2.0 (compatible; MSIE 3.0B; Windows NT)`	IE 3 (beta) emulating Navigator 2
`Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-us) AppleWebKit/85 (KHTML, like Gecko) Safari/85`	Apple Safari 1.0 (release no. 85) for MacOS X
`Opera/7.10 (Windows 98; U)  [en]`	Opera 7.1 (set to identify as Opera) for Windows 98

Because the `userAgent` property contains a lot of the same information as the `appVersion` property, the same cautions just described apply to the `userAgent` string and the environment data it returns.

Speaking of compatibility and browser versions, the question often arises whether your scripts should distinguish among incremental releases within a browser's generation (for example, 6.0, 6.01, 6.02, and so on). The latest incremental release occasionally contains bug fixes and (rarely) new features on which you may rely. If that is the case, I suggest you look for this information when the page loads and recommend to the user that he or she download the latest browser version. Beyond that, I suggest scripting for the latest version of a given generation and not bothering with branching for incremental releases.

See Chapters 13 and 14 for more information about designing pages for cross-platform deployment.

Example

Listing 38-1 provides a number of reusable functions that your scripts can employ to determine a variety of information about the currently running browser. This is not intended in any way to be an all-inclusive browser-sniffing routine; instead, I offer samples of how to extract information from the key `navigator` properties to determine various browser conditions.

All functions in Listing 38-1 return a Boolean value inline with the pseudo-question presented in the function's name. For example, the `isWindows()` function returns `true` if the browser is any type of Windows browser; otherwise, it returns `false`. (In Internet Explorer 3, the values are `0` for `false` and `-1` for `true`, but those values are perfectly usable in `if` conditional phrases). If this kind of browser detection occurs frequently in your pages, consider moving these functions into an external `.js` source library for inclusion in your pages (see Chapter 13).

When you load this page, it presents fields that display the results of each function depending on the type of browser and client operating system you use.

Listing 38-1: **Functions to Examine Browsers**

```
<html>
   <head>
      <title>UserAgent Property Library</title>
      <script type="text/javascript">
      // basic brand determination
      function isNav() {
         return (navigator.appName == "Netscape" && !isOpera() && !isSafari());
      }

      function isIE() {
         return (navigator.appName == "Microsoft Internet Explorer");
      }

      function isOpera() {
         return (navigator.userAgent.indexOf("Opera") != -1);
      }

      function isSafari() {
         return (navigator.userAgent.indexOf("Safari") != -1);
      }

      // operating system platforms
      function isWindows() {
         return (navigator.appVersion.indexOf("Win") != -1);
      }

      function isWin95NT() {
         return (isWindows() && (navigator.appVersion.indexOf("Win16") == -1 &&
navigator.appVersion.indexOf("Windows 3.1") == -1));
      }

      function isMac() {
         return (navigator.appVersion.indexOf("Mac") != -1);
      }

      function isMacPPC() {
         return (isMac() && (navigator.appVersion.indexOf("PPC") != -1 ||
navigator.appVersion.indexOf("PowerPC") != -1));
      }

      function isUnix() {
         return (navigator.appVersion.indexOf("X11") != -1);
      }

      // browser versions
      function isGeneration2() {
         return (parseInt(navigator.appVersion) == 2);
      }

      function isGeneration3() {
         return (parseInt(navigator.appVersion) == 3);
      }

      function isGeneration3Min() {
         return (parseInt(navigator.appVersion.charAt(0)) >= 3);
      }
```

```
   function isNav4_7() {
      return (isNav() && parseFloat(navigator.appVersion) == 4.7);
   }

   function isMSIE4Min() {
      return (isIE() && navigator.appVersion.indexOf("MSIE") != -1);
   }

   function isMSIE6_0() {
      return (navigator.appVersion.indexOf("MSIE 6.0") != -1);
   }

   function isNN6Min() {
      return (isNav() && parseInt(navigator.appVersion) >= 5);
   }

   // element referencing syntax
   function isDocAll() {
      return (document.all) ? true : false;
   }

   function isDocW3C() {
      return (document.getElementById) ? true : false;
   }

   // fill in the blanks
   function checkBrowser() {
      var form = document.forms[0];
      form.brandNN.value = isNav();
      form.brandIE.value = isIE();
      form.brandSaf.value = isSafari();
      form.brandOp.value = isOpera();
      form.win.value = isWindows();
      form.win32.value = isWin95NT();
      form.mac.value = isMac();
      form.ppc.value = isMacPPC();
      form.unix.value = isUnix();
      form.ver3Only.value = isGeneration3();
      form.ver3Up.value = isGeneration3Min();
      form.Nav4_7.value = isNav4_7();
      form.Nav6Up.value = isNN6Min();
      form.MSIE4.value = isMSIE4Min();
      form.MSIE6_0.value = isMSIE6_0();
      form.doc_all.value = isDocAll();
      form.doc_w3c.value = isDocW3C();
   }
   </script>
</head>
<body onload="checkBrowser()">
   <h1>About This Browser</h1>
   <form>
      <h2>Brand</h2>
      Netscape Navigator: <input type="text" name="brandNN" size="5" />
      Internet Explorer: <input type="text" name="brandIE" size="5" />
      Apple Safari: <input type="text" name="brandSaf" size="5" />
      Opera: <input type="text" name="brandOp" size="5" />
      <hr />
      <h2>Browser Version</h2>
```

Continued

Listing 38-1 *(continued)*

```
        3.0x Only (any brand): <input type="text" name="ver3Only" size="5" />
        <p>3 or Later (any brand): <input type="text" name="ver3Up"
           size="5" /></p>
        <p>Navigator 4.7: <input type="text" name="Nav4_7" size="5" /></p>
        <p>Navigator 6+: <input type="text" name="Nav6Up" size="5" /></p>
        <p>MSIE 4+: <input type="text" name="MSIE4" size="5" /></p>
        <p>MSIE 6.0: <input type="text" name="MSIE6_0" size="5" /></p>
        <hr />
        <h2>OS Platform</h2>
        Windows: <input type="text" name="win" size="5" /> Windows
        95/98/2000/NT: <input type="text" name="win32" size="5" />
        <p>Macintosh: <input type="text" name="mac" size="5" /> Mac PowerPC:
           <input type="text" name="ppc" size="5" /></p>
        <p>Unix: <input type="text" name="unix" size="5" /></p>
        <hr />
        <h2>Element Referencing Style</h2>
        Use <tt>document.all</tt>: <input type="text" name="doc_all"
        size="5" />
        <p>Use <tt>document.getElementById()</tt>: <input type="text"
           name="doc_w3c" size="5" /></p>
     </form>
  </body>
</html>
```

Related Items: `appMinorVersion`, `cpuClass`, `oscpu`, `platform` properties.

appMinorVersion

Value: One-character string. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

In IE parlance, the *minor version* is indicated by the first digit to the right of the decimal in a full version number. But the "version number" referred to here is the number that the `navigator.appVersion` property reports, not the actual version of the browser. For example, although IE5.5 seems to have a version number of 5 and a minor version number of 5, the `appVersion` reports version 4.0. In this case, the `minorAppVersion` reports 0. Thus, you cannot use the `appMinorVersion` property to detect differences between, say, IE5 and IE5.5. That information is buried deeper within the string returned by `appVersion` and `userAgent`.

Example

Use The Evaluator (Chapter 13) to examine the two related version properties of your IE browser(s). Type the following two statements into the top text box and observe the results:

```
navigator.appVersion
navigator.minorAppVersion
```

There is a good chance that the values returned are not related to the browser version number shown after `MSIE` in the `appVersion` value.

Related Item: `appVersion` property.

browserLanguage

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The browserLanguage property in IE4+ (and the language property in NN4+) returns the identifier for a localized language version of the program (it has nothing to do with scripting or programming language). The value of the browserLanguage property almost always is the same as the other IE language-related properties, unless the user changes the Windows control panel for regional settings after installing IE. In that case, browserLanguage returns the original language of the browser application, while the other properties report the language indicated in the system-level preferences panel.

Users of the multilanguage version of Windows 2000/XP can choose alternate languages for menus and dialog boxes. The browserLanguage property returns the language you choose for those settings.

These short strings may resemble, but are not identical to, the URL suffixes for countries. Moreover, when a language has multiple dialects, the dialect can also be a part of the identifier. For example, en is the identifier for English. However, en-us (or en-US) represents the American dialect of English, while en-gb (or en-GB) represents the dialect recognized in Great Britain. NN sometimes includes these values as part of the userAgent data as well. Table 38-2 shows a sampling of language identifiers used for all language-related properties of the navigator object.

Table 38-2: Sample navigator.browserLanguage Values

navigator.language	*Language*
en	English
de	German
es	Spanish
fr	French
ja	Japanese
da	Danish
it	Italian
ko	Korean
nl	Dutch
pt	Brazilian Portuguese
sv	Swedish

You can assume that a user of a particular language version of the browser or system is also interested in content in the same language. If your site offers multiple language paths, you can use this property setting to automate the navigation to the proper section for the user.

Related Items: navigator.userAgent, navigator.language, navigator.systemLanguage, navigator.userLanguage properties.

cookieEnabled

Value: Boolean. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `cookieEnabled` property allows your scripts to determine easily if the browser has cookie functionality turned on. You can surround cookie-related statements with an `if` construction as follows:

```
if (navigator.cookieEnabled) {
   // do cookie stuff here
}
```

This works reliably only on browsers that implement the property. Because older browsers do not have this `navigator` object property, the `if` condition appears `false` (even though cookies may be turned on).

You can still check for cookie functionality in older browsers, but only clumsily. The technique entails assigning a "dummy" cookie value to the `document.cookie` property and attempting to read back the cookie value. If the value is there, cookies are enabled.

Example

Use The Evaluator (Chapter 13) to see the value of the `navigator.cookieEnabled` property on your browsers. Enter the following statement into the top text box:

```
navigator.cookieEnabled
```

Feel free to change the cookie preferences setting temporarily to see the new value of the property. You do not have to relaunch the browser for the new setting to take effect.

Related Item: `document.cookie` property.

cpuClass

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `cpuClass` property returns one of several fixed strings that identifies the family of central processing units running IE. Possible values and their meanings are as follows:

cpuClass	*Description*
`x86`	Intel processor (and some emulators)
`PPC`	Motorola Power PC processor (for example, Macintosh)
`68K`	Motorola 68000-family processor (for example, Macintosh)
`Alpha`	Digital Equipment Alpha processor
`Other`	Other processors, such as SPARC

The processor is not a good guide to determining the operating system because you can run multiple operating systems on most of the preceding processor families. Moreover, the `cpuClass` value represents the processor that the browser "thinks" it is running on. For example, when a Windows version of IE is hosted by the *Virtual PC* emulator on a PowerPC Macintosh, the `cpuClass` is reported as `x86` even though the actual hardware processor is `PPC`.

Example

Use The Evaluator (Chapter 13) to see how IE reports the `cpuClass` of your PC. Enter the following statement into the top text box:

```
navigator.cpuClass
```

Related Item: `navigator.oscpu` property.

language

Value: String. Read-Only
Compatibility: WinIE-, MacIE-, NN4+, Moz1+, Safari1+

The NN4+ `language` property returns the language code for the browser application. Although the comparable IE property (`navigator.browserLanguage`) has morphed in later versions to focus on the operating system language, NN's property deals exclusively with the language for which the browser application is written.

Related Item: `navigator.browserLanguage` property.

mimeTypes

Value: Array of `mimeType` objects. Read-Only
Compatibility: WinIE-, MacIE5, NN4+, Moz1+, Safari1+

A *`mime` (Multipurpose Internet Mail Extension)* type is a file format for information that travels across the Internet. Browsers usually have a limited capability for displaying or playing information beyond HTML text and one or two image standards (`.gif` and `.jpg` are the most common formats). To fill in the gap, browsers maintain an internal list of `mime` types with corresponding instructions on what to do when information of a particular `mime` type arrives at the client. For example, when a CGI program serves up an audio stream in an audio format, the browser locates that `mime` type in its table (the `mime` type is among the first chunk of information to reach the browser from the server) and then launches a helper application or activates a plug-in capable of playing that `mime` type. Your browser is not equipped to display every `mime` type, but it does know how to alert you when you don't have the helper application or plug-in needed to handle an incoming file. For instance, the browser may ask if you want to save the file for later use or switch to a Web page containing more information about the necessary plug-in.

The `mimeTypes` property of the `navigator` object is simply the array of `mime` types about which your browser knows (see the "mimeType Object" section later in this chapter). NN3+ browsers come with dozens of `mime` types already listed in their tables (even if the browser doesn't have the capability to handle all those items automatically). If you have third-party plug-ins in Navigator's plug-ins directory/folder or helper applications registered with Navigator, that array contains these new entries as well.

If your Web pages are media-rich, you want to be sure that each visitor's browser is capable of playing the media your page has to offer. With JavaScript and NN3+, you can cycle through the `mimeTypes` array to find a match for the `mime` type of your media. Then use the properties of the `mimeType` object (detailed later in this chapter) to ensure the optimum plug-in is available. If your media still requires a helper application instead of a plug-in, the array only lists the `mime` type; thus, you can't determine whether a helper application is assigned to this `mime` type from the array list.

Example

For examples of the `mimeTypes` property and details about using the `mimeType` object, see the discussion of this object later in the chapter. A number of simple examples showing how to use this property to see whether the `navigator` object has a particular MIME type do not go far enough in determining whether a plug-in is installed and enabled to play the incoming data.

Related Item: `navigator.plugins` property; `mimeType` object.

onLine

Value: Boolean. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `onLine` property lets scripts determine the state of the offline browsing setting for the browser. Bear in mind that this property does not reveal whether the page is accessed via the Net or a local hard disk. The browser can be in online mode and still access a local page; in this case, the `onLine` property returns `true`.

With the offline browsing capabilities of IE4+, users may prefer to download copies of pages they wish to reference frequently (perhaps on a disconnected laptop computer). In such cases, your pages may want to avoid network-reliant content when accessed offline. For example, if your page includes a link to a live audio feed, you can dynamically generate that link with JavaScript — but do so only if the user is online:

```
if (navigator.onLine) {
   document.write("<a href='broadcast.rna'>Listen to Audio</a>");
}
```

Example

Use The Evaluator (Chapter 13) to see the online state of your IE browsers. Enter the following statement into the top text box:

```
navigator.onLine
```

Verify your browsing mode by checking the Work Offline choice in the File menu. If it is checked, the `onLine` property should return `false`.

oscpu

Value: String. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `oscpu` property of Mozilla-based browsers returns a string that reveals OS- or CPU-related information about the user's environment. The precise string varies widely with the client OS. For instance, a Windows 98 machine reports `Win98`, while a Macintosh running MacOS X reports `PPC Mac OS X Mach-O`. The string formats for Windows NT/XP versions are not standardized, so they offer values such as `WinNT4.0` and `Windows NT 5.0`. Windows XP reports as being Windows NT in the `oscpu` property (as it does in the `userAgent` property). UNIX platforms reveal more details, such as the system version and hardware.

Example

Use The Evaluator (Chapter 13) with NN6 to see what your client machine reports to you by entering the following statement into the top text box:

```
navigator.oscpu
```

Related Item: `navigator.cpuClass` property.

platform

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The navigator.platform value reflects the operating system according to the codes established initially by Netscape for its userAgent values. Table 38-3 lists typical values of several operating systems.

In the long list of browser detection functions in Listing 38-1, I elected not to use the navigator.platform property because it is not backward-compatible. Meanwhile, the other properties in that listing are available to all scriptable browsers.

Table 38-3: Sample navigator.platform Values

navigator.platform	*Operating System*
Win32	Windows XP
Win98	Windows 98
WinNT	Windows NT
Win16	Windows 3.*x*
Mac68k	Mac (680x0 CPU)
MacPPC	Mac (PowerPC CPU)
SunOS	Solaris

Notice that the navigator.platform property does not go into versioning of the operating system. Only the raw name is provided.

Example

Use The Evaluator (Chapter 13) to see what your computer reports as its operating system. Enter the following statement into the top text box:

```
navigator.platform
```

Related Item: navigator.userAgent property.

plugins

Value: Array of plug-in objects. Read-Only
Compatibility: WinIE-, MacIE5, NN3+, Moz1+, Safari1+

You rarely find users involved with Web page design who have not heard about *plug-ins* — the technology that enables you to embed new media types and foreign file formats directly into Web documents. For instance, instead of requiring you to view a video clip in a separate window atop the main browser window, a plug-in enables you to make that viewer as much a part of the page design as a static image. The same goes for audio players, 3-D animation, chat sessions — even the display of Microsoft Office documents, such as PowerPoint and Word.

When many browsers launch, they create an internal list of available plug-ins located in a special directory/folder (the name varies with the browser and operating system). The navigator.plugins array lists the items registered at launch time. Each plug-in is, itself, an object with several properties.

The Windows version of IE4+ supports this property only to return an empty array. In other words, the property is defined, but it does not contain `plugin` objects — a nonexistent object in IE for Windows. But on the Macintosh side, IE5 supports the way Netscape Navigator, Mozilla, and Safari allow script inspection of `mime` types and plug-ins. To see ways of determining plug-in support for WinIE, see the section '"Plug-in" detection in WinIE' later in this chapter.

Having your scripts investigate the visitor's browser for a particular installed plug-in is a valuable capability if you want to guide the user through the process of downloading and installing a plug-in (if the system does not have it currently).

Example

For examples of the `plugins` property and for details about using the `plugin` object, see the section "plugin Object" later in this chapter. Also see Chapter 40 on the CD-ROM for information on embedded element objects.

Related Items: `navigator.mimeTypes property`; `plugin` object.

product
productSub
vendor
vendorSub

Value: String. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari1+

With the Mozilla browser engine being developed in an Open Source environment, any number of vendors might adapt the engine for any number of browser products. Some distributors of the browser, such as ISPs and computer manufacturers, may also tailor the browser slightly for their customers. These four properties can reveal some of the pedigree of the browser currently running scripts on the page.

Two categories of properties — one for the product, one for the vendor — each have a pair of fields (a primary and secondary field) that can be populated as the vendor sees fit. Some of this information may contain data, such as an identifying number of the *build* (development version) used to generate the product. A script at a computer maker's Web site page may look for a particular series of values in these properties to welcome the customer or to advise the customer of a later build version that is recommended as an upgrade.

Example

Use The Evaluator (Chapter 13) on NN6+/Moz to see the values returned for these four properties (Safari 1.0 doesn't support `vendorSub`). Enter each of the following statements into the top text box of the page and see the values for each in the Results box:

```
navigator.product
navigator.productSub
navigator.vendor
navigator.vendorSub
```

Also check the value of the `navigator.userAgent` property to see how many of these four property values are revealed in the `userAgent` property.

Related Item: `navigator.userAgent` property.

securityPolicy

Value: String. Read-Only
Compatibility: WinIE-, MacIE-, NN4+, Moz1+, Safari-

The Netscape-specific `securityPolicy` property returns a string that indicates which cryptographic scheme is implemented in the current browser. Typical string values include `US and CA domestic policy` and `export policy`. Each policy indicates the number of bits used for encryption, usually governed by technology export laws. The corresponding IE property is `document.security`. All Mozilla-based browsers have only one version, and the property's value is an empty string.

Related Item: `document.security` property.

systemLanguage
userLanguage

Value: Language code string. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

These two IE-specific properties report the language code of the written language specified for the operating system. For most operating system versions, these two values are the same. Some Windows versions enable you to set system preferences differently for the base operating system and the language for a given user. Both of these property values can differ from the `navigator.browserLanguage` property if the user downloads and installs the browser with the system set to one language and then changes the system settings to another language.

Example

Use The Evaluator (Chapter 13) with your IE4+ browser to compare the values of the three language-related properties running on your computer. Enter each of the following statements into the top text box:

```
navigator.browserLanguage
navigator.systemLanguage
navigator.userLanguage
```

Don't be surprised if all three properties return the same value.

Related Item: `navigator.browserLanguage` property.

userAgent

(See `appCodeName`)

userLanguage

(See `systemLanguage`)

userProfile

Value: `userProfile` object. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `userProfile` property returns a reference to the IE `userProfile` object. This object provides scripted access to a limited range of user profile settings with the user's permission. For details, see the `userProfile` object discussion later in this chapter.

Related Item: `userProfile` object.

vendor
vendorSub

(See product)

Methods

javaEnabled()

Returns: Boolean.
Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

Although most modern browsers ship with Java support turned on, a user can easily turn it off in a preferences dialog box (or even elect not to install it with the browser). Some corporate installations may also turn off Java as the default setting for their users. If your pages specify Java applets, you don't normally have to worry about this property because the applet tag's alternate text fills the page in the places where the applet normally goes. But if you script applets from JavaScript (via LiveConnect, Chapter 44), you don't want your scripts making calls to applets or Java classes if Java support is turned off. In a similar vein, if you create a page with JavaScript, you can fashion two different layouts depending on the availability of Java.

The navigator.javaEnabled() method returns a Boolean value reflecting the preferences setting. This value does not reflect Java support in the browser necessarily (and especially not the Java version supported), but rather whether Java is turned on inside the browsers for which this method is supported. A script cannot change the browser's preference setting, but its value does change immediately upon toggling the Preference setting.

Related Items: navigator.preference() method; LiveConnect (Chapter 44).

preference(*name* [, *val*])

Returns: Preference value.
Compatibility: WinIE-, MacIE-, NN4+, Moz-, Safari-

The user normally sets browser preferences. Until NN4 and the advent of signed scripts, almost all settings were completely out of view of scripts — even when it made sense to expose them. But with signed scripts and the navigator.preference() method, many NN preferences are now viewable and settable with the user's permission. These preferences were exposed to scripting primarily for the purposes of centralized configuration administration for enterprise installations. I don't recommend altering the browser preferences of a public Web site visitor, even if given permission to do so — the user may not know how much trouble you can cause.

When you want to read a particular preference setting, you pass only the preference name parameter with the method. Reading a preference requires a signed script with the target of UniversalPreferencesRead (see Chapter 46 on the CD-ROM). To change a preference, pass both the preference name and the value (with a signed script target of UniversalPreferencesWrite).

Table 38-4 shows a handful of scriptable preferences in Mozilla-based browsers (learn more about Mozilla preferences at http://www.mozilla.org/catalog/end-user/customizing/briefprefs.html). Most items have corresponding entries in the preferences window in NN4+ (shown in parentheses). Notice that the preference name uses dot syntax. The cookie security level is a single preference value with a matrix of integer values indicating the level.

Table 38-4: navigator.preference() Values Sampler

navigator.preference	*Value*	*Preference Dialog Listing*
`security.enable_java`	Boolean	(Advanced) Enables Java
`javascript.enabled`	Boolean	(Advanced) Enables JavaScript
`autoupdate.enabled`	Boolean	(Advanced) Enables AutoInstall
`network.cookie.cookieBehavior`	`0`	(Advanced) Accepts all cookies
`network.cookie.cookieBehavior`	`1`	(Advanced) Accepts only cookies that get sent back to the originating server
`network.cookie.cookieBehavior`	`2`	(Advanced) Disables cookies
`network.cookie.warnAboutCookies`	Boolean	(Advanced) Warns you before accepting a cookie

One preference to watch out for is the one that disables JavaScript. If you disable JavaScript, only the user can reenable JavaScript by manually changing the setting in the browser's preferences dialog box.

Example

The page in Listing 38-2 displays checkboxes or radio buttons for several preference settings plus one text box to show a preference setting value for the size of the browser's disk cache. You will receive a security warning each time the scripts enable the Privilege Manager.

One function reads all the preferences and sets the form control values accordingly. Another function sets a preference when you click its checkbox. Rerunning the `showPreferences()` function also helps verify that you set the preference.

Listing 38-2: **Reading and Writing Browser Preferences**

```
<html>
   <head>
      <title>Reading/Writing Browser Preferences</title>
      <script type="text/javascript">
      function setPreference(pref, value) {
         netscape.security.PrivilegeManager.enablePrivilege(
            "UniversalPreferencesWrite");
         navigator.preference(pref, value);
         netscape.security.PrivilegeManager.revertPrivilege(
            "UniversalPreferencesWrite");
         showPreferences();
      }

      function showPreferences() {
         var form = document.forms[0];
         netscape.security.PrivilegeManager.enablePrivilege(
            "UniversalPreferencesRead");
         form.cacheSize.value =
```

Continued

Listing 38-2 *(continued)*

```
navigator.preference("browser.cache.disk.capacity");
         form.autoIEnable.checked = navigator.preference("autoupdate.enabled");
         var cookieSetting =
navigator.preference("network.cookie.cookieBehavior");
         for (var i = 0; i < 3; i++) {
            document.getElementById("cookie" + i).checked = (i == cookieSetting)
?
               true : false;
         }
         var toolbarSetting =
navigator.preference("browser.chrome.toolbar_style");
         for (var i = 0; i < 3; i++) {
            document.getElementById("toolbar" + i).checked = (i ==
toolbarSetting) ?
               true : false;
         }
         form.cookieWarn.checked = navigator.preference(
            "network.cookie.warnAboutCookies");
         netscape.security.PrivilegeManager.revertPrivilege(
            "UniversalPreferencesRead");
      }
      </script>
   </head>
   <body onload="showPreferences()">
      <h1>Mozilla Browser Preferences Settings Sampler</h1>
      <hr />
      <form>
         <input type="checkbox" name="autoIEnable"
         onclick="setPreference('autoupdate.enabled',this.checked)"
/>AutoInstall
         Enabled<br />
         <b>Toolbar Buttons: </b>
         <input type="radio" name="toolbarPriv" id="toolbar0"
         onclick="setPreference('browser.chrome.toolbar_style',0)" />Toolbar
         Pictures Only
         <input type="radio" name="toolbarPriv" id="toolbar1"
         onclick="setPreference('browser.chrome.toolbar_styler',1)" />Toolbar
         Text Only
         <input type="radio" name="toolbarPriv" id="toolbar2"
         onclick="setPreference('browser.chrome.toolbar_style',2)" />Toolbar
         Pictures & Text<br />
         <b>Cookie Permissions: </b>
         <input type="radio" name="cookiePriv" id="cookie0"
         onclick="setPreference('network.cookie.cookieBehavior',0)" />Accept
         All Cookies
         <input type="radio" name="cookiePriv" id="cookie1"
         onclick="setPreference('network.cookie.cookieBehavior',1)" />Accept
         Only Cookies Sent Back to Server
         <input type="radio" name="cookiePriv" id="cookie2"
         onclick="setPreference('network.cookie.cookieBehavior',2)" />Disable
         Cookies<br />
         <input type="checkbox" name="cookieWarn"
```

```
         onclick="setPreference('network.cookie.warnAboutCookies',this.checked)"
/>
         Warn Before Accepting Cookies<br />
         Disk cache is <input type="text" name="cacheSize" size="10" />
         KB<br />
      </form>
   </body>
</html>
```

Related Item: `navigator.javaEnabled()` method.

`taintEnabled()`

Returns: Boolean.
Compatibility: WinIE4+, MacIE4+, NN3+, Moz-, Safari-

Navigator 3 featured a partially implemented security feature called *data tainting*, which was turned off by default. This feature was replaced by signed scripts; but for backward compatibility, the `navigator.taintEnabled()` method is available in more modern browsers that don't employ tainting (in which case, the method always returns `false`). Do not employ this method in your scripts.

mimeType Object

Properties	*Methods*	*Event Handlers*
`description`		
`enabledPlugin`		
`type`		
`suffixes`		

Syntax

Accessing `mimeType` properties:

```
navigator.mimeTypes[i].property
navigator.mimeTypes["MIMEtype"].property
```

Compatibility: WinIE-, MacIE5, NN3+, Moz1+, Safari1+

About this object

A `mimeType` object is essentially an entry in the internal array of `mime` types about which the browser knows. NN3+, for example, ships with an internal list of more than five dozen `mime` types. Only a handful of these types are associated with helper applications or plug-ins. But add to that list all of the plug-ins and other helpers you've added, and the number of `mime` types can grow to more than a hundred.

The mime type for the data is usually among the first bits of information to arrive at a browser from the server. A mime type consists of two pieces of information: type and subtype. The traditional way of representing these pieces is as a pair separated by a slash, as in

```
text/html
image/gif
audio/wav
video/quicktime
application/pdf
application/x-zip-compressed
```

If a file does not contain the mime type "header" (or a CGI program sending the file does not precede the transmission with the mime type string), the browser receives the data as a text/plain mime type. When you load the file from a local hard drive, the browser looks to the filename's extension (the suffix after the period) to figure out the file's type.

Regardless of the way it determines the mime type of the incoming data, the browser then acts according to instructions it maintains internally. You can see these settings by looking at preferences settings usually associated with the name "Applications."

By having the mimeType object available to JavaScript, your page can query a visitor's NN3+, MacIE5, or Safari browser to discover whether it has a particular mime type listed currently and whether the browser has a corresponding plug-in installed and enabled. In such queries, the mimeType and plugin objects work together to help scripts make these determinations. (For plug-in detection for WinIE, see the section '"Plug-in" detection in WinIE' later in this chapter.)

Because of the close relationship between mimeType and plugin objects, I save the examples of using these objects and their properties for a section later in this chapter. There you can see how to build functions into your scripts that enable you to examine how well a visitor's browser is equipped for either a mime type or data that requires a specific plug-in. In the meantime, be sure that you understand the properties of both objects.

Properties

description

Value: String. Read-Only
Compatibility: WinIE-, MacIE5, NN3+, Moz1+, Safari1+

While registering with the browser at launch time, plug-ins provide the browser with an extra field of information: a plain-language description of the plug-in. If a particular mime type has a plug-in associated with it and enabled for it, the plug-in's description passes through to become the description of the mimeType object. For example, the Adobe Acrobat plug-in (whose mime type is application/pdf) supplies the following description fields:

```
(NN3/NN4)    Acrobat
(NN6+)       Acrobat (*.pdf)
```

When a mime type does not have a plug-in associated with it (either no plug-in is installed or a helper application is used instead), you often see the type property value repeated in the description field.

enabledPlugin

Value: plugin object. Read-Only
Compatibility: WinIE-, MacIE5, NN3+, Moz1+, Safari1+

The descriptions of the mimeType and plugin objects seem to come full circle when you reach the mimeType.enabledPlugin property. The reason is that the property is a vital link between a known mime type and the plug-in that the browser engages when data of that type arrives.

Knowing which plug-in is associated with a mime type is very important when you have more than one plug-in capable of playing a given mime type. For example, the Crescendo midi audio plug-in can take the place of the default audio plug-in if you set up your browser that way. Therefore, all midi data streams play through the Crescendo plug-in. If you prefer to have your Web page's midi sound played only through another plug-in, your script needs to know which plug-in is set to receive your data and perhaps alert the user accordingly. These kinds of conflicts are not common, except where there is strong competition for players of various audio and video media. For other kinds of content, each plug-in developer typically creates a new type of data that has a unique mime type. But you have no guarantee of such uniqueness, so I highly recommend a careful check of mime type and plug-in if you want your page to look professional.

The enabledPlugin property evaluates to a plugin object. Therefore, you can dig a bit deeper with this information to fetch the name or filename properties of a plug-in directly from a mimeType object. You can use The Evaluator (with NN3+, MacIE5, and Safari) to study the relationship between mimeType and plugin objects:

1. Enter the following statement into the bottom text box to examine the properties of a mimeType object:

   ```
   navigator.mimeTypes[0]
   ```

   Notice that the mimeTypes array returns an object.

2. Inspect the plugin object from the bottom text box:

   ```
   navigator.mimeTypes[0].enabledPlugin
   ```

   You then see properties and values for a plugin object (described later in this chapter).

3. Check the plugin object for a different mimeType object by using a different index value:

   ```
   navigator.mimeTypes[7].enabledPlugin
   ```

The mimeTypes array index values vary almost with every browser, depending on what the user has installed. Therefore, do not rely on the index position in a script to assume that a particular mimeType object is in that position on all browsers.

Example

See the section "Looking for mime Types and Plug-ins" later in this chapter.

Related Item: plugin object.

type

Value: String. Read-Only
Compatibility: WinIE-, MacIE5, NN3+, Moz1+, Safari1+

A mimeType object's type property is the combination of the type and subtype commonly used to identify the kind of data coming from the server. CGI programs, for example, typically precede a data transmission with a special header string in the following format:

```
Content-type: type/subtype
```

This string prompts a browser to look up how to treat an incoming data stream of this kind. As you see later in this chapter, knowing whether a particular `mime` type is listed in the `navigator.mimeTypes` array is not enough. A good script must dig deeper to uncover additional information about what is truly available for your data.

The `type` property has a special place in the `mimeType` object in that its string value can act as the index to the `navigator.mimeTypes` array. Therefore, to get straight to the `mimeType` object for, say, the `audio/wav` `mime` type, your script can reference it directly through the `mimeTypes` array:

```
navigator.mimeTypes["audio/wav"]
```

This same reference can then get you straight to the enabled plug-in (if any) for the `mime` type:

```
navigator.mimeTypes["audio/wav"].enabledPlugin
```

Example

See the section "Looking for `mime` Types and Plug-ins" later in this chapter.

Related Item: `description` property.

`suffixes`

Value: String. Read-Only
Compatibility: WinIE-, MacIE5, NN3+, Moz1+, Safari1+

Every `mime` type has one or more filename extensions, or suffixes, associated with it. You can read this information for any `mimeType` object via the `suffixes` property. The value of this property is a string. If the `mime` type has more than one suffix associated with it, the string contains a comma-delimited listing as in

```
mpg, mpeg, mpe
```

Multiple versions of a suffix have no distinction among them. Those `mime` types that are best described in four or more characters (derived from a meaningful acronym, such as mpeg) have three-character versions to accommodate the "8-dot-3" filename conventions of MS-DOS and its derivatives.

Example

See the section "Looking for `mime` Types and Plug-ins" later in this chapter.

plugin Object

Properties	*Methods*	*Event Handlers*
`name`	`refresh()`	
`filename`		
`description`		
`length`		

Syntax

Accessing `plugin` object properties or method:

```
navigator.plugins[i].property | method()
navigator.plugins["plugInName"].property | method()
```

Compatibility: WinIE-, MacIE5, NN3+, Moz1+, Safari1+

About this object

The `plugin` object offers a view of the plug-in mechanism from the browser's perspective: The software items registered with the browser at launch time stand ready for any matching `mime` type that comes from the Net. One of the main purposes of having these objects scriptable is to let your scripts determine whether a desired plug-in is currently registered with the browser and to help with installing a plug-in.

The close association between the `plugin` and `mimeType` objects, demonstrated by the `mimeType.enabledPlugin` property, is equally visible coming from the direction of the plug-in. A `plugin` object evaluates to an array of `mime` types that the plug-in interprets. Use The Evaluator (Chapter 13) to experiment with `mime` types from the point of view of a plug-in. Begin by finding the name of the plug-in that your browser uses for a common animation `mime` type:

1. Enter the following statement into the top text box:

   ```
   navigator.mimeTypes["application/x-shockwave-flash"].enabledPlugin.name
   ```

   If you use NN7+, MacIE5, or Safari, the value returned is probably `"Shockwave Flash"`. Copy the name into the clipboard so that you can use it in subsequent statements. The remaining examples show `"LiveAudio"` where you should paste in your plug-in's name.

2. Enter the following statement into the top text box:

   ```
   navigator.plugins["Shockwave Flash"].length
   ```

   Instead of the typical index value for the array notation, use the actual name of the plug-in. This expression evaluates to a number indicating the total number of different `mime` types that the plug-in recognizes.

3. Look at the first `mime` type specified for the plug-in by entering the following statement into the top text box:

   ```
   navigator.plugins["Shockwave Flash"][0].type
   ```

The two successive pairs of square brackets is not a typo: Because the entry in the `plugins` array evaluates to an array itself, the second set of square brackets describes the index of the array returned by `plugins["Shockwave Flash"]`—a period does not separate the sets of brackets. In other words, this statement evaluates to the `type` property of the first `mimeType` object contained by the Shockwave/Flash plug-in.

I doubt that you will have to use this kind of construction much; if you know the name of the desired plug-in, you know what `mime` types it already supports. In most cases, you come at the search from the `mime` type direction and look for a specific, enabled plug-in. See the section "Looking for `mime` Types and Plug-ins" later in this chapter for details on how to use the `plugin` object in a production setting.

Properties

name
filename
description
length

Value: String. Read-Only
Compatibility: WinIE-, MacIE5, NN3+, Moz1+, Safari1+

The first three properties of the `plugin` object provide descriptive information about the plug-in file. The plug-in developer supplies the name and description. It's unclear whether future versions of plug-ins will differentiate themselves from earlier ones via either of these fields. Thus, while there is no explicit property that defines a plug-in's version number, that information may be part of the string returned by the `name` or `description` properties.

Be aware that plug-in authors may not assign the same name to every OS platform version of a plug-in. Be prepared for discrepancies across platforms. You should hope that the plug-in that you're interested in has a uniform name across platforms because the value of the `name` property can function as an index to the `navigator.plugins` array to access a particular `plugin` object directly.

Another piece of information available from a script is the plug-in's filename. On some platforms, such as Windows, this data comes in the form of a complete pathname to the plug-in DLL file; on other OS platforms, only the plug-in filename appears.

Finally, the `length` property of a `plugin` object counts the number of `mime` types that the plug-in recognizes (but is not enabled for necessarily). Although you can use this information to loop through all possible `mime` types for a plug-in, a more instructive way is to have your scripts approach the issue via the `mime` type (as discussed later in this chapter).

Example

See the section "Looking for `mime` Types and Plug-ins" later in this chapter.

Related Item: mimeType.description property.

Methods

refresh()

Returns: Nothing.
Compatibility: WinIE-, MacIE5, NN3+, Moz1+, Safari1+

You may have guessed that many browsers determine their lists of installed plug-ins while they launch. If you drop a new plug-in file into the plug-ins directory/folder, you have to quit the browser and relaunch it before the browser sees the new plug-in file. But that isn't a very friendly approach if you take pains to guide a user through downloading and installing a new plug-in file. The minute the user quits the browser, you have a slim chance of getting that person right back. That's where the `refresh()` method comes in.

The `refresh()` method is directed primarily at the browser, but the syntax of the call reminds the browser to refresh just the plug-ins:

```
navigator.plugins.refresh()
```

Interestingly, this command works only for adding a plug-in to the existing collection. If the user removes a plug-in and invokes this method, the removed one stays in the `navigator.plugins` array — although it may not be available for use. Only the act of quitting and relaunching the browser makes a plug-in removal take full effect.

Looking for mime Types and Plug-Ins

If you go to great lengths to add new media and data types to your Web pages, you certainly want your visitors to reap the benefits of those additions. But you cannot guarantee that they have the requisite plug-ins installed to accommodate that fancy data. Most modern browser versions provide a bit of internal "smarts" by noticing when data requiring an uninstalled plug-in is about to load and trying to help the user install a missing plug-in. You may wish, however, to take more control over the process by examining the user's browser plug-in functionality prior to loading the external data file.

The best source of information, when available, is the software developer of the plug-in. Macromedia, for example, provides numerous technical notes on its Web site (`www.macromedia.com`) about plug-in detection for its various plug-ins and versions. Unfortunately, that kind of assistance is not always easy to find from other vendors.

A lot of the discussion thus far in this chapter addresses the objects that make plug-in and `mime` type support detection possible in some browsers. Netscape for NN3 initially introduced these objects. Since then, they have been adopted by MacIE5 and Safari. Microsoft makes it possible — but not easy — to determine whether a particular plug-in is available for WinIE. The approach for WinIE is entirely different from what I have covered so far; if you wish to perform cross-browser detection, you have to branch your code accordingly. I outline each approach next in its own section, starting with the NN3+/MacIE5/Safari way.

Overview: Using mimeType and plugin objects

The value of performing your own inspection of plug-in support is that you can maintain better control of your site visitors who don't have the necessary plug-in yet. Rather than merely providing a link to the plug-in's download site, you can build a more complete interface around the download and installation of the plug-in without losing your visitor. I have some suggestions about such an interface at the end of this discussion.

How you go about inspecting a visitor's plug-in library depends on what information you have about the data file or stream and how precise you must be in locating a particular plug-in. Some plug-ins may override `mime` type settings that you normally expect to find in a browser. Another issue that complicates matters is that the same plug-in may have a different name (`navigator.plugins[i].name` property), depending on the operating system. Therefore, searching your script for the presence of a plug-in by name is not good enough if the name differs from the Macintosh version to the Windows version. Fortunately, this is less of an issue today than it was in earlier plug-in generations.

One other point that can help you decide the precise approach to take is which information about the plug-in — support for the data `mime` type or the presence of a particular plug-in — is important to your page and scripts. If your scripts rely on the existence of a plug-in that you can script via LiveConnect, be sure that the plug-in is present and enabled for the desired `mime` type (so that the plug-in is ensured of loading when it encounters a reference to the URL of the external data). But if you care only that a plug-in of any kind supports your data's `mime` type, you can simply make sure that any plug-in is enabled for your MIME type.

To help you jump-start the process in your scripts, I discuss three utility functions you can use in your own scripts. These functions are excerpts from a long listing (Listing 38-3), which is

located in its entirety among the Chapter 38 listing files on this CD-ROM. The pieces not shown here are merely user interface elements that enable you to experiment with these functions.

Verifying a mime type

Listing 38-3a is a function whose narrow purpose is to determine if the browser currently has plug-in support enabled for a given `mime` type (in the *type*/*subtype* format as a string). The first `if` construction verifies that there is a `mimeType` object for the supplied `mime` type string. If such an object exists, the next `if` construction determines whether the `enabledPlugin` property of the `mimeType` object returns a valid object. If so, the function returns `true` — meaning that the `mime` type has a plug-in (of unknown supplier) available to play the external media.

Listing 38-3a: **Verifying a MIME Type**

```
// Pass "<type>/<subtype>" string to this function to find
// out if the MIME type is registered with this browser
// and that at least some plug-in is enabled for that type.
function mimeIsReady(mime_type) {
   if (navigator.mimeTypes[mime_type]) {
      if (navigator.mimeTypes[mime_type].enabledPlugin) {
         return true;
      }
   }
   return false;
}
```

Verifying a plug-in

In Listing 38-3b, you let JavaScript see if the browser has a specific plug-in registered in the `navigator.plugins` array. This method approaches the installation question from a different angle. Instead of querying the browser about a known `mime` type, the function inquires about the presence of a known plug-in. But because more than one registered plug-in can support a given `mime` type, this function explores one step further to see whether at least one of the plug-in's `mime` types (of any kind) is enabled in the browser.

Listing 38-3b: **Verifying a Plug-In**

```
// Pass the name of a plug-in for this function to see
// if the plug-in is registered with this browser and
// that it is enabled for at least one MIME type of any kind.
function pluginIsReady(plug_in) {
   plug_in = plug_in.toLowerCase();
   for (var i = 0; i < navigator.plugins.length; i++) {
      if (navigator.plugins[i].name.toLowerCase().indexOf(plug_in) != -1) {
         for (var j = 0; j < navigator.plugins[i].length; j++) {
            if (navigator.plugins[i][j].enabledPlugin) {
               return true;
            }
         }
         return false;
      }
```

```
   }
   return false;
}
```

The parameter for the `pluginIsReady()` function is a string consisting of the plug-in's name. As discussed earlier, the precise name may vary from OS to OS or from version to version. The function here assumes that you aren't concerned about plug-in versioning. It also assumes (with reasonably good experience behind the assumption) that a brand-name plug-in contains a string with the brand in it. Thus, the `pluginIsRead()` function simply looks for the existence of the passed name within the `plugin` object's `name` property. For example, this function accepts `"QuickTime"` as a parameter and agrees that there is a match with the plug-in named `"QuickTime Plug-in 6.0.2"`. The script loops through all registered plug-ins for a substring comparison (converting both strings to all lowercase to help overcome discrepancies in capitalization).

Next comes a second repeat loop, which looks through the `mime` types associated with a plug-in (in this case, only a plug-in whose name contains the parameter string). Notice the use of the strange, double-array syntax for the most nested `if` statement: For a given plug-in (denoted by the `i` index), you have to loop through all items in the `mime` types array (`j`) connected to that plug-in. The conditional phrase for the last `if` statement has an implied comparison against `null` (see another way of explicitly showing the `null` comparison in Listing 38-3a). The conditional statement evaluates to either an object or `null`, which JavaScript can accept as `true` or `false`, respectively. The point is that if an enabled plug-in is found for the given `mime` type of the given plug-in, this function returns `true`.

Verifying both plug-in and mime type

The last utility function (Listing 38-3c) is the safest way of determining whether a visitor's browser is equipped with the "right stuff" to play your media. This function requires both a MIME type and plug-in name as parameters and also makes sure that both items are supported and enabled in the browser before returning `true`.

Listing 38-3c: Verifying Plug-in and mime Type

```
// Pass "<type>/<subtype>" and plug-in name strings for this
// function to see if both the MIME type and plug-in are
// registered with this browser, and that the plug-in is
// enabled for the desired MIME type.
function mimeAndPluginReady(mime_type,plug_in) {
   if (mimeIsReady(mime_type)) {
      var plugInOfRecord = navigator.mimeTypes[mime_type].enabledPlugin;
      plug_in = plug_in.toLowerCase();
      for (var i = 0; i < navigator.plugins.length; i++) {
         if (navigator.plugins[i].name.toLowerCase().indexOf(plug_in) != -1) {
            if (navigator.plugins[i] == plugInOfRecord) {
               return true;
            }
         }
      }
   }
   return false;
}
```

This function starts by calling the `mimeIsReady()` function from Listing 38-3a. After that, the function resembles the one in Listing 38-3b until you reach the most nested statements. Here, instead of looking for any old `mime` type, you insist on the existence of an explicit match between the `mime` type passed as a parameter and an enabled `mime` type associated with the plug-in. To see how these functions work on your NN3+, MacIE5, or Safari browser, open the complete file (`lst38-03.htm`) from this CD-ROM. The actual listing also includes code that branches around IE for Windows and other browsers that don't support this way of inspecting `mime` types and plug-ins.

Managing manual plug-in installation

If your scripts determine that a visitor does not have the plug-in your data expects, you may want to consider providing an electronic guide to installing the plug-in. One way to do this is to open a new frameset (in the main window). One frame can contain step-by-step instructions with links to the plug-in's download site. The download site's page can appear in the other frame of this temporary window. The steps must take into account all installation requirements for every platform, or, alternatively, you can create a separate installation document for each unique class of platform. For instance, you must decode Macintosh files frequently from binhex format and then uncompress them before you move them into the plug-ins folder. Other plug-ins have their own, separate installation program. The final step should include a call to

```
navigator.plugins.refresh()
```

to make sure that the browser updates its internal listings. After that, the script can return to the `document.referrer`, which should be the page that sends the visitor to the installation pages. All in all, the process is cumbersome — it's not like downloading a Java applet. But if you provide some guidance, you stand a better chance of the user returning to play your cool media. Also consider letting the browser's own updating facilities handle the job (albeit not as smoothly in many cases) by simply loading the data into the page, ready or not.

"Plug-in" detection in WinIE

WinIE4+ provides some built-in facilities that may take the place of plug-in detection in some circumstances. First of all, it's important to recognize that WinIE does not use the term "plug-in" in the same way that Netscape and other browsers use it. Due to the integration between IE and the Windows operating system, WinIE employs system-wide ActiveX controls to handle the job of rendering external content. Some of these controls are designed to be accessed from outside their walls, thus allowing client-side scripts to get and set properties or invoke methods built into the controls. These controls behave a lot like plug-ins, so you frequently see them referenced as "plug-ins," as they are in this book.

WinIE prefers the `<object>` tag for both loading the plug-in (ActiveX control) and assigning external content to it. One of the attributes of the `object` element is `classid`, which points to a monstrously long string of hexadecimal numbers known as the `guid` (Globally Unique Identifier). When the browser encounters one of these `guid`s, it looks into the Windows Registry to get the path to the actual plug-in file. If the plug-in is not installed on the user's machine, the object doesn't load and any other HTML nested inside the `<object>` tag renders instead. Thus, you can display a static image placeholder or HTML message about the lack of the plug-in. But plug-in detection comes in most handy when your scripts need to communicate with the plug-in, such as directing an embedded Windows Media Player plug-in to change sound files or to play. When you build code around a scriptable plug-in, your scripts should make sure that the plug-in object is indeed present so they don't generate errors.

The idea of using the <object> tag instead of the <embed> tag is that the <object> tag loads a specific plug-in, whereas the mime type of the data referenced by the <embed> tag lets the browser determine which plug-in to use for that mime type. It's not uncommon, therefore, to see an <object> tag definition surround an <embed> tag — both referencing the same external data file. If the optimum plug-in fails to load, the <embed> tag is observed, and the browser tries to find any plug-in for the file's mime type.

With an object element as part of the HTML page, the element itself is a valid object — even if the plug-in fails to load. Therefore, you must do more to validate the existence of the loaded plug-in than simply test for the existence of the object element. To that end, you need to know at least one scriptable property of the plug-in. Unfortunately, not all scriptable plug-ins are fully documented, so you occasionally must perform some detective work to determine which scriptable properties are available. While you're on the search for clues, you can also determine the version of the plug-in and make it a minimum version that your object element allows to load.

Tracking down plug-in details

Not everyone has access to the Microsoft programming development environments (for example, Visual Studio) through which you can find out all kinds of information about an installed ActiveX control. If you don't have access, you can still dig deep to get most (if not all) of the information you need. The tools you can use include the Windows Registry Editor (regedit), The Evaluator (Chapter 13), and, of course, your text editor and WinIE4+ browser. The following steps take you through finding out everything you need to know about the Windows Media Player control:

1. If you don't know the guid for the Media Player (most people get it by copying someone else's code that employs it), you can use the Registry Editor (regedit.exe) to find it. Open the Registry Editor (in Win95/98/NT/XP, choose Run from the Start menu and enter regedit; if that option is not available in your Windows version, search for the file named regedit).
2. Expand the HKEY_CLASSES_ROOT folder.
3. Scroll down to the nested folder named CLSID, and click that folder.
4. Choose Edit/Find, and enter Windows Media Player. If you were searching for a different plug-in, you would enter an identifying name (usually the product name) in this place.
5. Keep pressing F3 (Find Next) until the editor lands upon a folder whose default value (in the right side of the Registry Editor window) shows Windows Media Player.
6. The number inside curly braces next to the highlighted folder is the plug-in's guid. Right-click the number and choose Copy Key Name. Paste the number into your document somewhere for future reference. Eventually, it will be part of the value assigned to the classid attribute of the object element.
7. Expand the highlighted folder.
8. Click the folder named InprocServer32. The default value should show a pathname to the actual ActiveX control for the Windows Media Player plug-in.
9. Right-click the (Default) name for the path and choose Modify. The full pathname is visible in an editable field.
10. Armed with this pathname information, open My Computer and locate the actual file inside a directory listing.

11. Right-click the file and choose Properties.
12. Click the Version tab (if present).
13. Copy the version number (generally four sets of numbers delimited by commas), and paste it into your document for future reference. Eventually, it will be assigned to the `codebase` attribute of the `object` element.

    You are now ready to try loading the plug-in as an object and look for properties you can test for.
14. Add an `object` tag to The Evaluator source code. This can go inside the `head` or just before the `</body>` tag. For example, your tag should look something like the following:

    ```
    <object id="wmp" width="1" height="1"
    classid="CLSID:0A4286EA-E355-44FB-8086-AF3DF7645BD9"
    codebase="#Version=9,0,0,2980">
    </object>
    ```

    Copy and paste the numbers for the `guid` and version. Two points to watch out for: First, be sure that the `guid` value is preceded by `CLSID:` in the value assigned to `classid`; second, be sure the version numbers are preceded by the prefix shown.
15. Load (or reload) the page in WinIE4+.

    At this point, the `wmp` object should exist. If the associated plug-in loads successfully, the `wmp` object's properties include properties exposed by the plug-in.
16. Enter `wmp` into the bottom text box to inspect properties of the `wmp` object. Be patient: It may take many seconds for the retrieval of all properties.

    In case you can't readily distinguish between the `object` element object properties and properties of the scriptable plug-in, scroll down to the `wmp.innerHTML` property and its values. When an object loads successfully, any parameters that it accepts are reflected in the `innerHTML` for the `object` element. Each `param` element has a name — the name of one of the scriptable properties of the plug-in.
17. Look for one of the properties that has some kind of value by default (in other words, other than an empty string or `false`). In Windows Media Player, this can be `CreationDate`. Use this property as an object detection condition in scripts that need to access the Windows Media Player properties or methods:

    ```
    if (wmp && wmp.CreationDate) {
       // statements that "talk to" plug-in
    }
    ```

Setting a minimum version number

The four numbers that you grab in step 13 in the previous section represent the version of the plug-in as installed on your computer. Unless you have a way of verifying that your external content runs on earlier versions of the plug-in (if there are earlier versions), you can safely specify *your* version as the minimum.

Specificity rankings for the four numbers of a version decrease as you move from left to right. For example, version 9,0,25,2 is later than 9,0,0,0; version 10,0,0,0 is later than both of them. If you specify 9,0,25,2, and the user has 9,0,24,0 installed, the plug-in does not load and the object isn't available for scripting. On the other hand, a user with 9,0,26,0 has the object present because the `codebase` attribute for the version specifies a minimum allowable version to load.

When an object requires VBScript

Not all objects that load via the `object` element are scriptable through JavaScript (JScript). Occasionally, an object is designed so that its properties are exposed only to VBScript. This happens, for example, with the Microsoft Windows Media Rights Manager (DRM) object. To find out if the browser (operating system) is equipped with DRM, your page loads the object via the `object` element as usual; however, a separate VBScript section must access the object to test for the existence of one of its properties. Because script segments written in either language can access each other, this isn't a problem provided you know what the property or method is for the object. The following fragment from the Head section of a document demonstrates how JavaScript and VBScript can interact so that JavaScript code can branch based on the availability of DRM:

```
<head>
<object id="drmObj" height="1" width="1"
classid="CLSID:760C4B83-E211-11D2-BF3E-00805FBE84A6"></object>
<script type="text/vbscript">
function hasDRM()
   on error resume next
   drmObj.StoreLicense("")
   if (err.number = 0) then
      hasDRM = true
   else
      hasDRM = false
   end if
end function
</script>
<script type="text/javascript">
var gHasDRM;
if (drmObj && hasDRM()) {
   gHasDRM = true;
} else {
   gHasDRM = false;
}
</script>
</head>
```

The JavaScript segment sets a Boolean global variable to indicate whether the object has loaded correctly. Part of the job is accomplished via the `hasDRM()` function in the VBScript segment. From VBScript, the `drmObj` object responds to the `StoreLicense()` method call, but it throws a VBScript error indicating that no parameter was sent along with the method. Any subsequent scripts in this page can use the `gHasDRM` global variable as a conditional expression before performing any actions requiring the object (which works in tandem with the Windows Media Player).

screen Object

Properties	*Methods*	*Event Handlers*
`availHeight`		
`availLeft`		
`availTop`		
`availWidth`		

Continued

screen

Properties	*Methods*	*Event Handlers*
bufferDepth		
colorDepth		
fontSmoothingEnabled		
height		
pixelDepth		
updateInterval		
width		

Syntax

Accessing `screen` object properties:

```
(All)    screen.property
(IE4+)   [window.]navigator.screen.property
```

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

About this object

Browsers other than from the earliest generations provide a `screen` object that lets your scripts inquire about the size and color settings of the video monitor used to display a page. Properties are carefully designed to reveal not only the raw width and height of the monitor (in pixels), but also what the available width and height are once you take into account the operating system's screen-hogging interface elements (for example, the Windows taskbar and the Mac menu bar).

Internet Explorer 4 provides a `screen` object, although it appears as a property of the `window` object in the IE4+ object model. Only three properties of the IE4+ `screen` object — `height`, `width`, and `colorDepth` — share the same syntax as NN4+'s `screen` object.

Properties

availHeight
availWidth
height
width

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

With the availability of window sizing methods in version 4 browsers and later, your scripts may want to know how large the user's monitor is. This is particularly important if you set up an application to run in kiosk mode, which occupies the entire screen. Two pairs of properties let scripts extract the dimensions of the screen. All dimensions are in pixels.

You can extract the gross height and width of the monitor from the `screen.height` and `screen.width` properties. Thus, a monitor rated as a 1024 × 768 monitor returns values of 1024 and 768 for `width` and `height`, respectively.

But not every pixel of the screen's gross size is available as displayable area for a window. To the rescue come the `screen.availWidth` and `screen.availHeight` properties. For example, 32-bit Windows operating systems display the taskbar. The default location for this bar is at the bottom of the window, but users can reorient it along any edge of the screen. If the default behavior of always showing the taskbar is in force, the bar takes away from the screen real estate available for window display (unless you intentionally size or position a window so that part of the window extends under the bar). When along the top or bottom edge of the screen, the taskbar occupies 28 vertical pixels; when positioned along one of the sides, the bar occupies 60 horizontal pixels. On the Macintosh platform, the 20-pixel-deep menu bar occupies a top strip of the screen. Although you can position and size windows so the menu bar partially covers them, it is not a good idea to open a window in (or move a window into) that location.

You can use the available screen size values as settings for window properties. For example, to arrange a window so that it occupies all available space on the monitor, you must position the window at the top left of the screen and then set the outer window dimensions to the available sizes as follows:

```
function maximize() {
   window.moveTo(0,0);
   window.resizeTo(screen.availWidth, screen.availHeight);
}
```

The preceding function positions the window appropriately on the Macintosh just below the menu bar so that the menu bar does not obscure the window. If, however, the client is running Windows and the user positions the taskbar at the top of the screen, the window is partially hidden under the taskbar (you cannot query the available screen space's coordinates). Also in Windows, the appearance is not exactly the same as a maximized window. See the discussion of the `window.resizeTo()` method in Chapter 16 for more details. Note that MacIE generally returns a value for `screen.availHeight` that is about 24 pixels fewer than the actual available height (even after taking into account the Mac menu bar).

For Navigator 3+ and Mozilla, you can use LiveConnect to access a native Java class that reveals the overall screen size (not the available screen size). If the user runs Navigator 3 and Java is enabled, you can place the following script fragment in the Head portion of your document to set variables with screen width and height:

```
var toolkit = java.awt.Toolkit.getDefaultToolkit();
var screenSize = toolkit.getScreenSize();
```

The `screenSize` variable is an object whose properties (`width` and `height`) contain the pixel measures of the current screen. This LiveConnect technique works only in Netscape and Mozilla browsers (IE does not provide direct access to Java classes). In fact, you can also extract the screen resolution (pixels per inch) in the same manner. The following statement, added after the preceding ones, sets the variable resolution to that value:

```
var resolution = toolkit.getScreenResolution();
```

Related Items: `window.innerHeight`, `window.innerWidth`, `window.outerHeight`, `window.outerWidth` properties; `window.moveTo()`, `window.resizeTo()` methods.

availLeft
availTop

FTR:screen.availLeft

Value: Integer. Read-Only
Compatibility: WinIE-, MacIE-, NN4+, Moz1+, Safari1+

The availLeft and availTop properties return the pixel measure of where the available space of the screen begins. The only time these values are anything other than zero is when a user positions the taskbar along the left or top edges of the screen. For example, if the user positions the taskbar along the top of the screen, you do not want to position a window any higher than the 28 pixels occupied by the taskbar. There are no corresponding properties for IE.

Example

If you are a Windows user, you can experiment with these NN4+ properties via The Evaluator (Chapter 13). With the taskbar at the bottom of the screen, enter these two statements into the top text box:

```
screen.availLeft
screen.availTop
```

Next, drag the taskbar to the top of the screen and try both statements again. Now, drag the taskbar to the left edge of the screen and try the statements once more.

Related Items: screen.availWidth, screen.availHeight properties; window.moveTo() method.

bufferDepth

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

By default, IE does not use any offscreen buffering of page content. But adjusting the bufferDepth property enables you to turn on offscreen buffering and control the color depth of the buffer. Using offscreen buffering may improve the smoothness of path-oriented animation through positioning.

The default value (buffering turned off) is 0. By setting the property to -1, you instruct IE to set the color depth of the offscreen buffer to the same color depth as the screen (as set in the control panel). This should be the optimum value, but you can also force the offscreen buffer to have one of the following bit depths: 1, 4, 8, 15, 16, 24, or 32.

Related Items: screen.colorDepth, screen.pixelDepth properties.

colorDepth
pixelDepth

Value: Integer. Read-Only
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1_

You can design a page with different color models in mind because your scripts can query the client to find out how many colors the user sets the monitor to display. This is helpful if you have more subtle color schemes that require 16-bit color settings or images tailored to specific palette sizes.

Both the screen.colorDepth and screen.pixelDepth properties return the number of color bits to which the color client computer's video display control panel is set. The screen.colorDepth value may take into account a custom color palette; so for NN4+, you may prefer to rely only on the screen.pixelDepth value. (IE4+, however, supports only the screen.colorDepth property of this pair.) You can use this value to determine which of two image versions to load, as shown in the following script fragment that runs as the document loads:

```
if (screen.colorDepth > 8 ) {
   document.write("<img src='logoHI.jpg' height='60' width='100' />");
} else {
   document.write("<img src='logoLO.jpg' height=width='100' />");
}
```

In this example, the `logoHI.jpg` image is designed for 16-bit displays or better, while the colors in `logoLO.jpg` are tuned for 8-bit display.

Related Item: `screen.bufferDepth` property.

`fontSmoothingEnabled`

Value: Boolean. Read-Only
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

Some versions of the Windows OS have a Display control panel setting for "Smooth Edges" on screen fonts. The `fontSmoothingEnabled` property lets your script see the state of that setting. This setting can affect, for example, which stylesheet you enable because it has font specifications that look good only when smoothing is enabled. A default installation of Windows has this feature turned off.

`updateInterval`

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The `updateInterval` property is the number of milliseconds between screen updates. The default value of zero lets IE arbitrate among the demands for screen updates in a highly animated setting. If you set this value to a large number, more screen updates are accumulated in a buffer — preventing some animated steps from updating the screen.

userProfile Object

Properties	*Methods*	*Event Handlers*
	`addReadRequest()`	
	`clearRequest()`	
	`doReadRequest()`	
	`getAttribute()`	

Syntax

Accessing `userProfile` object methods:

```
(IE4+)  [window.]navigator.userProfile.method()
```

Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

About this object

The `userProfile` object is an IE-specific (and Windows, at that) property that acts as the gateway to the user profile information that the client computer collects from the user. You can retrieve none of this information via JavaScript without permission from the user. Access to this information is performed in a strict sequence, part of which enables you to define how the request for this private information is worded when the user is presented with the request.

User profile data consists of nearly 30 fields of personal information about the user's contact information. Each of these fields has a name, which by and large conforms to the vCard standard. Your scripts can request one or more specific fields from the list, rather than having to deal with the entire set of fields.

The sequence for accessing this data entails four basic steps:

1. Put the request for each vCard field into a queue that is maintained in the browser's memory (via the `addReadRequest()` method).
2. Execute the batch request, which displays a detailed dialog box to the user (via the `doReadRequest()` method). If a user profile is in effect, the user sees which fields you are requesting plus the data in the vCard. The user then has the chance to deselect one or more of your choices — or disallow access completely.
3. Get each attribute by name (via the `getAttribute()` method). You invoke this method once for each vCard field.
4. Clear the queue of requests (via the `clearRequest()` method).

Returned values are strings. Thus, you can prefill the customer information for an order form or capture the information in hidden fields that are submitted with a visible form.

Listing 38-4 demonstrates the use of the four key methods of the `userProfile` object. After the page loads, it attempts to extract the data from every vCard field and displays both the attribute name and the value as associated with the current user profile in a table. Notice that the names of the attributes are hard-wired because the object does not provide a list of implemented attributes.

Listing 38-4: **Accessing userProfile Data**

```
<html>
  <head>
     <title>userProfile Object</title>
     <script type="text/javascript">
     var attrs = ["Business.City","Business.Country","Business.Fax",
        "Business.Phone","Business.State","Business.StreetAddress",
        "Business.URL","Business.Zipcode","Cellular","Company","Department",
        "DisplayName","Email","FirstName","Gender","Home.City","Home.Country",
        "Home.Fax","Home.Phone","Home.State","Home.StreetAddress",
        "Home.Zipcode","Homepage","JobTitle","LastName","MiddleName",
        "Office","Pager"];
     function loadTable() {
        // make sure this executes only in IE4+ for Windows
        if ((navigator.userAgent.indexOf("Win") != -1) &&
           navigator.userProfile) {
           var newRow, newCell, attrValue;
           // queue up requests for every vCard attribute
           for (var i = 0; i < attrs.length; i++) {
              navigator.userProfile.addReadRequest("vCard." + attrs[i]);
```

```
            }
            // dispatch the request to let user accept or deny access
            navigator.userProfile.doReadRequest(1, "JavaScript Bible");
            // append rows to the table with attribute/value pairs
            for (var j = 0; j < attrs.length; j++) {
               newRow = document.all.attrTable.insertRow(-1);
               newRow.bgColor = "#FFFF99";
               newCell = newRow.insertCell(0);
               newCell.innerText = "vCard." + attrs[j];
               newCell = newRow.insertCell(1);
               // get the actual value
               attrValue = navigator.userProfile.getAttribute("vCard." +
                  attrs[j]);
               newCell.innerHTML = (attrValue) ? attrValue : " ";
            }
            // clean up after ourselves
            navigator.userProfile.clearRequest();
         } else {
            alert("This example requires IE4+ for Windows.");
         }
      }
      </script>
   </head>
   <body onload="loadTable()">
      <h1>userProfile Object</h1>
      <hr />
      <table id="attrTable" border="1" cellpadding="5">
         <tr bgcolor="#CCFFFF">
            <th>vCard Property</th>
            <th>Value</th>
         </tr>
      </table>
   </body>
</html>
```

It appears that the newer the version of Windows that the user runs, the more likely that user profile data is available. Even so, there may be little more than name and address data for those users who are careful not to fill out optional fields of Microsoft Web site forms requesting personal information.

Comparable information may be available from NN4+ users on any OS platform via signed scripts that access `ldap` preferences. See the discussion earlier in this chapter about the `navigator.preference()` method.

Methods

addReadRequest("*attributeName*")

Returns: Boolean.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

Before the user is asked for permission to reveal any personal information, you must queue up requests — even if there is just one field in which you are interested. For each field, use the `addReadRequest()` method and specify as the parameter a string of the attribute name. Acceptable attribute names are as follows:

```
vCard.Business.City
vCard.Business.Country
vCard.Business.Fax
vCard.Business.Phone
vCard.Business.State
vCard.Business.StreetAddress
vCard.Business.URL
vCard.Business.Zipcode
vCard.Cellular
vCard.Company
vCard.Department
vCard.DisplayName
vCard.Email
vCard.FirstName
vCard.Gender
vCard.Home.City
vCard.Home.Country
vCard.Home.Fax
vCard.Home.Phone
vCard.Home.State
vCard.Home.StreetAddress
vCard.Home.Zipcode
vCard.Homepage
vCard.JobTitle
vCard.LastName
vCard.MiddleName
vCard.Office
vCard.Pager
```

All attribute values are case-insensitive.

This method returns a Boolean value of `true` if the addition to the queue succeeds. A returned value of `false` usually means that the attribute value is not valid or that a request for that attribute name is already in the queue. If you fail to clear the queue after compiling one list of attributes, attempts to read the attribute result in a return value of `false`.

Example

See Listing 38-4 for an example of the `addReadRequest()` method in action. You can also invoke it from the top text box in The Evaluator. For example, enter the following statement to queue one request:

```
navigator.userProfile.addReadRequest("vCard.LastName")
```

To continue the process, see examples for `doReadRequest()` and `getAttribute()` later in this chapter.

Related Items: `clearRequest()`, `doReadRequest()`, and `getAttribute()` methods.

`clearRequest()`

Returns: Nothing.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

After retrieving the attributes whose names are stacked in the request queue, invoke the `clearRequest()` method to empty the queue. It is always good programming practice to clean up after yourself, especially when security concerns are involved.

Related Items: `addReadRequest()`, `doReadRequest()`, and `getAttribute()` methods.

doReadRequest(*reasonCode*, *identification*[, *domain*[, *path*[, *expiration*]]])

Returns: Nothing.
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

Once the names of the desired vCard attributes are stacked in the queue (via the `addReadRequest()` method), invoke the `doReadRequest()` method to prompt the user for the permission that your scripts need to gain access to the data. The user sees a detailed dialog box that lists the vCard fields you are requesting, as well as a description about your reason for wanting the data and who you are.

The first required parameter is an integer representing one of the standard descriptions as defined by the Internet Privacy Working Group. Associated text is displayed in the permission request dialog box that the user sees. The codes and their strings are as follows:

Code	*Description String*
0	Used for system administration.
1	Used for research and/or product development.
2	Used for completion and support of current transaction.
3	Used to customize the content and design of a site.
4	Used to improve the content of the site, including advertisements.
5	Used for notifying visitors about updates to the site.
6	Used for contacting visitors for marketing of services or products.
7	Used for linking other collected information.
8	Used by site for other purposes.
9	Disclosed to others for customization or improvement of the content and design of the site.
10	Disclosed to others, who may contact you, for marketing of services and/or products.
11	Disclosed to others, who may contact you, for marketing of services and/or products; you have the opportunity to ask a site not to do this.
12	Disclosed to others for any other purpose.

Although these description strings are fixed, you do have an opportunity to include some customized information in the second parameter. The parameter is intended to enable you to identify the Web site or organization requesting the information. Standards recommendations suggest you include a URL to the site, as well. In any case, the second parameter can be any string. But it is not treated like HTML, so do not attempt to include a clickable link here.

Two optional parameters enable you to specify a domain and path within that domain for which the user permissions are to apply. Both of these parameters closely mirror their usage in cookies, but they also depend on the capability to set an expiration date via the fifth parameter. Through IE6, however, the expiration date parameter is ignored. Therefore, permissions expire when the user quits the browser (just like temporary cookies do).

Example

See Listing 38-4 for an example of the `doReadRequest()` method in action. If you entered the `addReadRequest()` example for The Evaluator earlier in this chapter, you can now bring up the permissions dialog box (if you have a user profile for your version of Windows) by entering the following statement into the top text box:

```
navigator.userProfile.doReadRequest(1, "Just me!")
```

Related Items: `addReadRequest()`, `clearRequest()`, and `getAttribute()` methods.

`getAttribute("attributeName")`

Returns: String.
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `getAttribute()` method attempts to retrieve the vCard data based on the items queued via the `addReadRequest()` method. A permission dialog box provides the user an opportunity to choose which of the requested items to reveal or to deny all access to the information. Only one attribute name is permitted as a parameter to the `getAttribute()` method, requiring that you invoke the method for each attribute you want to fetch.

Example

See Listing 38-4 for an example of the `getAttribute()` method in action. Also, if you followed The Evaluator examples for this object, you can now extract the desired information (provided it is in your user profile). Enter the following statement into the top text box:

```
navigator.userProfile.getAttribute("vCard.LastName")
```

Related Items: `addReadRequest()`, `clearRequest()`, and `doReadRequest()` methods.

✦ ✦ ✦

CHAPTER 39

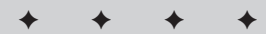
Positioned Objects

In This Chapter

Layer concepts

How to move, hide, and show content

The end of the `layer` element



This chapter tackles positioned objects and layers, and investigates the history of Dynamic HTML — the capability to alter content on the fly in response to user interaction — particularly with respect to Netscape Navigator 4, chronologically the first browser to offer these features. The original impetus for this chapter was the NN4 `layer` element and its associated object. What makes this discussion awkward is that the `layer` element and object became dead-end entities that never made it into the W3C standards process. NN6+/Moz instead have adopted the W3C standards for dynamic content, which more closely mimic the way Microsoft implemented its DHTML features starting with IE4. NN6+/Moz explicitly do not provide backward compatibility with scripted `layer` element objects, which also means that you must rewrite legacy Netscape Navigator applications to work in NN6+/Moz, if you haven't been done so already.

That leaves an ungainly task in this chapter to create a bridge between the old `layer` element and the more modern way of working with elements that can be positioned on the page, flown across the page, stacked in front of other elements, or hidden from view. The IE4+ and NN6+/W3C DOM way to accomplish all of this is through Cascading Style Sheets and the scripting thereof.

The NN4 `layer` element is well on its way to becoming a distant memory. But for the sake of historical completeness, the first half of this chapter covers the details of the now defunct `layer` and `ilayer` objects. The second half demonstrates how to apply CSS and modern DOM techniques to manage positioned elements in Mozilla-based browsers, Safari, and others.

What Is a Layer?

Terminology in the area of positioned elements has become a bit confusing over time. Because NN4 was the earliest browser to be released with positioned elements (the `layer` element), the term *layer* became synonymous with any positioned element. When IE4 came on the scene, it was convenient to call a stylesheet-positioned element (in other words, an element governed by a stylesheet rule with the `position` attribute) a *layer* as a generic term for any positioned element. In fact, NN4 even treated an element that was positioned through stylesheets as if it were a genuine layer object (although with some minor differences).

In the end, the layer term made good sense because no matter how it was achieved, a positioned element acted like a layer in front of the body content of a page. Perhaps you have seen how animated cartoons were created before computer animation changed the art.

Layers of clear acetate sheets were assembled atop a static background. Each sheet contained one character or portion of a character. When all the sheets were carefully positioned atop each other, the view (as captured by a still camera) formed a composite frame of the cartoon. To create the next frame of the cartoon, the artist moved one of the layers a fraction of an inch along its intended path and then took another picture.

If you can visualize how that operation works, you have a good starting point for understanding how layers work. Each layer contains some kind of HTML content that exists in its own plane above the main document that loads in a window. You can change or replace the content of an individual layer on the fly without affecting the layout of the other layers; you can also reposition, resize, or hide the entire layer under script control.

One aspect of layers that goes beyond the cartoon analogy is that a layer can contain other layers. When that happens, any change that affects the primary layer — such as moving the layer 10 pixels downward — also affects the layers nested inside. It's as if the nested layers are passengers of the outer layer. When the outer layer goes somewhere, the passengers do, too. And yet, within the "vehicle," the passengers may change seats by moving around without regard for what's going on outside.

With this analogy in mind, many commercial DHTML development tools and content authors refer to positioned elements as layers, which you can move, resize, stack, and hide independently of the body background. Therefore, references throughout this book to layers may mean anything from the NN4 layer object to an element positioned by way of stylesheets.

NN4 layer Object

Properties	*Methods*	*Event Handlers*
above	captureEvents()	onblur
background	handleEvent()	onfocus
below	load()	onload
bgcolor	moveAbove()	onmouseout
clip.bottom	moveBelow()	onmouseover
clip.left	moveBy()	
clip.right	moveTo()	
clip.top	moveToAbsolute()	
document	releaseEvents()	
left	resizeBy()	
name	resizeTo()	
pageX	routeEvent()	
pageY		
parentLayer		
siblingAbove		
siblingBelow		
src		
top		
visibility		
zIndex		

Syntax

Accessing layer object properties or methods:

```
[window.]document.layerName.[document.layerName. ...] property |
method([parameters])
[window.]document.layers[index].[document.layerName. ...]property |
method([parameters])
```

Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

About this object

You can create a `layer` object in NN4 in one of three ways. The first two ways use NN4-only syntax: the `<layer>` tag in HTML and the `new Layer()` constructor in JavaScript. The tag offers numerous attributes that establish the location, stacking order, and visibility. These attributes, in turn, become scriptable properties. If you create the layer through the constructor, you then use JavaScript to assign values to the object's properties.

The third way to create an NN4 `layer` object is to assign an absolute-positioned stylesheet rule to a block-level element — most typically a `div` element. This is the way that IE4+ and NN6+ do it, too. In practice, however, a positioned `div` element is not as robust (from rendering and scriptability standpoints) in NN4 as a genuine `layer` element. Therefore, it is sometimes necessary to branch a page's code to use `document.write()` for a `<layer>` tag in NN4 and a `<div>` tag in IE4+ and NN6+.

Layer references

The task of assembling JavaScript references to NN4 layers and the objects they contain resembles the same process for framesets (in fact, conceptually, a layer is like a dynamically movable and resizable free-floating frame). Therefore, before you start writing the reference, you must know the relationship between the document *containing* the script and the target of the reference.

To demonstrate how this works, I start with a script in the base document loaded into a window that needs to change the background color (`bgColor` property) of a layer defined in the document. The skeletal HTML is as follows:

```
<html>
   <head>
   </head>
   <body>
      <layer name="Flintstones" src="flintstonesFamily.html"></layer>
   </body>
</html>
```

From a script in the Head section, the statement that changes the layer's `bgColor` property is this:

```
document.Flintstones.bgColor = "yellow";
```

This syntax looks like the way you address any object in a document, such as a link or image. However, things get tricky in that each layer automatically contains a `document` object of its own. That `document` object is what holds the content of the layer. Therefore, if you want to inspect the `lastModified` property of the HTML document loaded into the layer, use this statement:

```
var modDate = document.Flintstones.document.lastModified;
```

The situation gets more complex if the layer has another layer nested inside it (one of those "passengers" that goes along for the ride). If the structure changes to

```
<html>
   <head>
   </head>
   <body>
      <layer name="Flintstones" src="flintstonesFamily.html">
         <layer name="Fred" src="fredFlintstone.html"></layer>
         <layer name="Wilma" src="wilmaFlintstone.html"></layer>
      </layer>
   </body>
</html>
```

references to items in the second level of layers get even longer. For example, to get the `lastModified` property of the `fredFlintstone.html` file loaded into the nested `Fred` layer, use this reference from the Head script:

```
document.Flintstones.document.Fred.document.lastModified
```

The reason for this is that NN4 does not have a shortcut access to every layer defined in a top-level document. As stated in the description of the `document.layers` property in Chapter 18, the property reflects only the first level of layers defined in a document. You must know the way to San Jose if you want to get its `lastModified` property.

Layers and forms

Because each layer has its own document, you cannot spread a form across multiple layers. Each layer's document must define its own `<form>` tags. If you need to submit one form from content located in multiple layers, one of the forms should have an `onsubmit` event handler to harvest all the related form values and place them in hidden input fields in the document containing the submitted form. In this case, you need to know how to devise references from a nested layer outward.

As a demonstration of reverse-direction references, I start with the following skeletal structure that contains multiple nested layers:

```
<html>
   <head>
   </head>
   <body>
      <form name="personal">
         <input type="text" name="emailAddr" />
      </form>
      <layer name="product" src="ultraGizmoLine.html">
         <layer name="color" src="colorChoice.html"></layer>
         <layer name="size" src="sizeChoice.html"></layer>
         <layer name="sendIt" src="submission.html"></layer>
      </layer>
   </body>
</html>
```

Each of the HTML files loaded into the layers also has a `<form>` tag defining some fields or select lists for relevant user choices, such as which specific model of the `UltraGizmo` line is selected, what color, and in what size. (These last two are defined as separate layers because their positions are animated when they are displayed.) The assumption here is that the Submit button is in the `sendIt` layer. That layer's document also includes hidden input fields for data to be pulled from the main document's form and three other layer forms. Two of those layers

are at the same nested level as `sendIt`, one is above it, and the main document's form is at the highest level.

To reach the `value` property of a field named `theColor` in the `color` layer, a script in the `sendIt` layer uses this reference:

```
parentLayer.document.color.document.forms[0].theColor.value
```

Analogous to working with frames, the reference starts with a reference to the next higher level (`parentLayer`) and then starts working its way down through the parent layer's document, the `color` layer, the `color` layer's document, and finally the form therein.

To reach the `value` property of a field named `modelNum` in the `product` layer, the reference starts the same way; but because the form is at the parent layer level, the reference goes immediately to that layer's document and form:

```
parentLayer.document.forms[0].modelNum.value
```

It may seem odd that a reference to an object at a different layer level is shorter than one at the same level (for example, the `color` layer), but the route to the parent layer is shorter than going via the parent layer to a sibling. Finally, to reach the value of the `emailAddr` field in the base document, the reference must ratchet out one more layer as follows:

```
parentLayer.parentLayer.document.forms[0].emailAddr.value
```

The two `parentLayer` entries step the reference out two levels, at which point the scope is in the base layer containing the main document and its form.

Layers and tables

The document-centered nature of NN4 layers also makes it difficult — if not impossible at times — to incorporate them inside tables. Even defining a layer that is contained by a `td` table cell can cause countless problems.

If you need to have absolute-positioned elements that look as though they are part of a table, I suggest you define the layers as freestanding elements outside of the table. After that, you can position the layers to make them look like they live in the table. You may also need to create empty placeholders in your table to make room for the overlaid layer. You can do this by way of a relative-positioned element inside the table cell whose visibility is hidden. This allows the element to flow as the page loads to accommodate the current browser window dimensions. Scripts can then read the location of the relative-positioned element and use those coordinates to move the absolute-positioned elements that are to overlay the hidden elements.

Properties

`above`
`below`
`siblingAbove`
`siblingBelow`

Value: Layer object. Read-Only
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

Each layer object is its own physical layer. Given that the variables x and y traditionally represent width and height, the third dimension — the position of a layer relative to the stack of

layers — is called the *z-order*. Layer orders are assigned automatically according to the loading order, with the highest number being the topmost layer. That topmost layer is the one closest to you as you view the page on the monitor.

If two layers are on a page, one layer must always be in front of the other even if they both appear to be transparent and visually overlap each other. Knowing which layer is above the other is important for scripting purposes, especially if your script needs to reorder the layering in response to user action. Layer objects have four properties to help you determine the layers adjacent to a particular layer.

The first pair of properties, *layerObject*.above and *layerObject*.below, takes a global look at all layers defined on the page regardless of the fact that one layer may contain any number of nested layers separate from other batches on the screen. If a layer lies above the one in question, the property contains a reference to that other layer; if no layer exists in that direction, the value is null. Attempts to retrieve properties of a nonexistent layer result in runtime scripting errors indicating that the object does not have properties (of course not — an object must exist before it can have properties).

To understand these two properties better, consider a document that contains three layers (in any nesting arrangement you like). The first layer to be defined is on the bottom of the stack. It has a layer above it, but none below it. The second layer in the middle has a layer both above and below it. And the topmost layer has a layer only below it, with no more layers above it (that is, coming toward your eye).

Another pair of properties, *layerObject*.siblingAbove and *layerObject*.siblingBelow, confines itself to the group of layers inside a parent layer container. Just as in real family life, siblings are descended from (teens might say "contained by") the same parent. An only child layer has no siblings, so both the *layerObject*.siblingAbove and *layerObject*.siblingBelow values are null. For two layers from the same parent, the first one to be defined has a sibling layer above it; the other has a sibling layer below it.

It is important to understand the difference between absolute layering and sibling layering to use these properties correctly. A nested layer might be the fifth layer from the bottom among all layers on the page but at the same time be the first layer among siblings within its family group. As you can see, these two sets of properties enable your script to be very specific about the relationships under examination.

Positioned objects in IE4+ and NN6+ have no comparable properties to the four described in this section.

Example

Listing 39-1 enables you to experiment with just one set of these properties: *layerObject*.above and *layerObject*.below. The page is almost in the form of a laboratory/quiz that enables you to query yourself about the values of these properties for two swappable layers.

Listing 39-1: **A Layer Quiz**

```
<html>
   <head>
      <title></title>
      <script type="text/javascript">
      function checkAbove(oneLayer) {
         document.forms[0].errors.value = "";
         document.forms[0].output.value = oneLayer.above.name;
      }
```

```
      function checkBelow(oneLayer) {
         document.forms[0].errors.value = "";
         document.forms[0].output.value = oneLayer.below.name;
      }
      function swapLayers() {
         if (document.yeller.above) {
            document.yeller.moveAbove(document.greeny);
         } else {
            document.greeny.moveAbove(document.yeller);
         }
      }
      function onerror(msg) {
         document.forms[0].output.value = "";
         document.forms[0].errors.value = msg;
         return true;
      }
      </script>
   </head>
   <body>
      <h1>Layer Ordering</h1>
      <hr />
      <form>
         Results:<input type="text" name="output" />
         <p><input type="button" value="Who's ABOVE the Yellow layer?"
            onclick="checkAbove(document.yeller)" /><br />
            <input type="button" value="Who's BELOW the Yellow layer?"
            onclick="checkBelow(document.yeller)" /></p>
         <p><input type="button" value="Who's ABOVE the Green layer?"
            onclick="checkAbove(document.greeny)" /><br />
            <input type="button" value="Who's BELOW the Green layer?"
            onclick="checkBelow(document.greeny)" /></p>
         <p><input type="button" value="Swap Layers" onclick="swapLayers()" />
         </p>
         <p>If there are any errors caused by missing<br />
            properties, they will appear below:<br />
            <textarea name="errors" cols="30" rows="3" wrap="virtual">
            </textarea></p>
      </form>
      <layer name="yeller" bgcolor="yellow" top="110" left="300" width="200"
      height="200">
         <b>This is just a yellow layer.</b>
      </layer>
      <layer name="greeny" bgcolor="lightgreen" top="150" left="340"
      width="200" height="200">
         <b>This is just a green layer.</b>
      </layer>
   </body>
</html>
```

The page contains two layers: one colored yellow and the other light green. Legends on four buttons ask you to guess whether one layer is above or below the other. For example, if you click the button labeled "Who's ABOVE the Yellow layer?" and the green layer is above it, the name of that green layer appears in the Results field. But if layers are oriented such that the returned value is `null`, the error message (indicating that the nonexistent object doesn't have a `name` property) appears in the error field at the bottom. Another button enables you to swap the order of the layers so you can try your hand at predicting the results based on your knowledge of layers and the `above` and `below` properties.

Positioned objects in IE4+ and NN6 have no comparable properties to the four described in this section.

Related Items: `layer.parentLayer` property; `layer.moveAbove()`, `layer.moveBelow()` methods.

background

Value: Image object. Read/Write
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

You can assign a background image to a layer. The `background` attribute of the `<layer>` tag usually sets the initial image, but you can assign a new image whenever you like via the *`layerObject`*`.background` property.

Layer background images are typically like those used for entire Web pages. They tend to be subtle — or at least of such a design and color scheme as not to distract from the primary content of the layer. On the other hand, the background image may in fact be the content. If so, then have a blast with whatever images suit you.

The value of the *`layerObject`*`.background` property is an image object (see Chapter 20). To change the image in that property on the fly, you must set the *`layerObject`*`.background.src` property to the URL of the desired image (just like changing `document.`*`imageName`*`.src` on the fly). You can remove the background image by setting the *`layerObject`*`.background.src` property to `null`. Background images smaller than the rectangle of the layer repeat themselves, just like document background pictures; images larger than the rectangle clip themselves to the rectangle of the layer rather than scaling to fit.

The IE4+ and NN6+ way of handling background images is through the `style.backgroundImage` property.

Example

A simple example (see Listing 39-2) defines one layer that features five buttons to change the background image of a second layer. I put the buttons in a layer because I want to make sure the buttons and background layer rectangles align themselves along their top edges on all platforms.

As the second layer loads, I merely assign a gray background color to it and write some reverse (white) text. Most of the images are of the small variety that repeats in the layer. One is a large photograph to demonstrate how images are clipped to the layer's rectangle. Along the way, I hope you also heed the lesson of readability demonstrated by the difficulty of reading text on a wild-looking background. For an example compatible with IE5+ and NN6+, see Listing 39-13.

Listing 39-2: Setting Layer Backgrounds

```
<html>
   <head>
      <title>Layer Backgrounds</title>
      <script type="text/javascript">
      function setBg(URL) {
         document.bgExpo.background.src = URL;
      }
      </script>
   </head>
   <body>
```

```
      <h1>Layer Backgrounds</h1>
      <hr />
      <layer name="buttons" top="100">
         <form>
            <input type="button" value="The Usual"
            onclick="setBg('cr_kraft.gif')" /><br />
            <input type="button" value="A Big One"
            onclick="setBg('arch.gif')" /><br />
            <input type="button" value="Not So Usual"
            onclick="setBg('wh86.gif')" /><br />
            <input type="button" value="Decidedly Unusual"
            onclick="setBg('sb23.gif')" /><br />
            <input type="button" value="Quick as..."
            onclick="setBg('lightnin.gif')" /><br />
         </form>
      </layer>
      <layer name="bgExpo" bgcolor="gray" top="100" left="250" width="300"
      height="260">
         <b><font color="white">Some text, which may or may not read well with
         the various backgrounds.</font></b>
      </layer>
   </body>
</html>
```

Related Items: `layer.bgColor` property; `image` object.

bgColor

Value: String. Read/Write

Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

A layer's background color fills the entire rectangle with the color set in the `<layer>` tag or from a script at a later time. Color values are the same as for document-related values; they may be in the hexadecimal triplet format or in one of the plain-language color names. You can turn a layer transparent by setting its `bgColor` property to `null`.

You control the corresponding behavior in IE4+ and NN6+ via the `style.backgroundColor` property.

Example

You can have some fun with Listing 39-3, which uses a number of layer scripting techniques. The page presents a kind of palette of eight colors, each one created as a small layer. Another, larger layer's `bgColor` property changes as you roll the mouse over any color in the palette.

To save HTML lines to create those eight color palette layers, I use a script to establish an array of colors and then `document.write()` the `<layer>` tags with appropriate attribute settings so the layers all line up in a contiguous row. By predefining a number of variable values for the size of the color layers, I can make all of them larger or smaller with the change of only a few script characters.

The `document` object handles the job of capturing the `mouseOver` events. I turn on the document's `captureEvents()` method such that it traps all `mouseOver` events and hands them to the `setColor()` function. The `setColor()` function reads the target object's `bgColor` and sets the larger layer's `bgColor` property to the same. If this page had other objects that could receive `mouseOver` events for other purposes, I would use `routeEvents()` to let those events pass on to their intended targets. For the purposes of this example, however, the events need to go no further. Listing 39-14 shows the same functionality working in IE5+ and NN6+.

Listing 39-3: **Layer Background Colors**

```
<html>
   <head>
      <title>Layer Background Colors</title>
      <script type="text/javascript">
      function setColor(e) {
         document.display.bgColor = e.target.bgColor;
      }
      document.captureEvents(Event.MOUSEOVER);
      document.onmouseover = setColor;
      </script>
   </head>
   <body>
      <h1>Layer Background Colors</h1>
      <hr />
      <script type="text/javascript">
      var oneLayer;
      var colorTop = 100;
      var colorLeft = 20;
      var colorWidth = 40;
      var colorHeight = 40;
      var colorPalette = new Array("aquamarine","coral","forestgreen",
         "goldenrod","red","magenta","navy","teal");
      for (var i = 0; i < colorPalette.length; i++) {
         oneLayer = "<layer name=swatch" + i + " top=" + colorTop;
         oneLayer += " left=" + ((colorWidth * i) + colorLeft);
         oneLayer += " width=" + colorWidth + " height=" + colorHeight;
         oneLayer += " bgcolor=" + colorPalette[i] + "><\/layer>\n";
         document.write(oneLayer);
      }
      </script>
      <layer name="display" bgcolor="gray" top="150" left="80" width="200"
      height="200">
         <center><b><font color="white">Some reversed text to test against
            background colors.</font></b></center>
      </layer>
   </body>
</html>
```

Related Items: `layer.background` property; `layer.onmouseover` event handler.

clip

Value: String. Read/Write
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

The *`layerObject`*`.clip` property is an object (the only one in NN4's document object model that exposes itself as a `rectangle` object) with six geographical properties defining the position and size of a rectangular area of a layer visible to the user. Those six properties are

- ✦ clip.top
- ✦ clip.left
- ✦ clip.bottom
- ✦ clip.right
- ✦ clip.width
- ✦ clip.height

The unit of measure is pixels, and the values are relative to the top-left corner of the layer object.

A clip region can be the same size as or smaller than the layer object. If the clip attribute is not defined in the <layer> tag, the clipping region is the same size as the layer. In this case, the clip.left and clip.top values are automatically zero because the clip region starts at the very top-left corner of the layer's rectangle (measurement is relative to the layer object whose clip property you're dealing with). The height and width of the layer object are not available properties in NN4. Therefore, you may have to use other means to get that information into your scripts if you need it. (I do it in Listing 39-4.) Also be aware that even if you set the height and width attributes of a layer tag, the content rules the initial size of the visible layer unless the tag also includes specific clipping instructions. Images, for example, expand a layer to fit the height and width attributes of the <img> tag; text (either from an external HTML file or inline in the current file) adheres to the <layer> tag's width attribute but flows down as far as necessary to display every character.

Setting a clip property does not move the layer or the content of the layer — only the visible area of the layer. Each adjustment has a unique impact on the apparent motion of the visible region. For example, if you increase the clip.left value from its original position of 0 to 20, the entire left edge of the rectangle shifts to the right by 20 pixels. No other edge moves. Changes to the clip.width property affect only the right edge; changes to the clip.height property affect only the bottom edge. Unfortunately, no shortcuts exist to adjust multiple edges at once. JavaScript is fast enough on most client machines to give the impression that multiple sides are moving if you issue assignment statements to different edges in sequence.

IE4+ and NN6+ have the style.clip property to assist in adjusting the clipping rectangle of a layer. But the W3C DOM's style.clip object does not offer additional subproperties to access individual edges or dimensions of the clipping rectangle. IE5's read-only currentStyle object does provide properties for the four edge dimensions. Listing 39-15 demonstrates how to adjust clipping in IE5+ and NN6+ syntax.

Example

Because of the edge movement behavior of adjustments to *layerObject*.clip properties, Listing 39-4 enables you to experiment with adjustments to each of the six properties. The document loads one layer that you can adjust by entering alternative values into six text fields — one per property.

As you enter values, all properties are updated to show their current values (via the showValues() function). Pay particular attention to the apparent motion of the edge and the effect the change has on at least one other property. For example, a change to the *layerObject*.clip.left value also affects the *layerObject*.clip.width property value.

Listing 39-4: **Adjusting layer.clip Properties**

```
<html>
   <head>
      <title>Layer Clip</title>
      <script type="text/javascript">
      var origLayerWidth = 0;
      var origLayerHeight = 0;
      function initializeXY() {
         origLayerWidth = document.display.clip.width;
         origLayerHeight = document.display.clip.height;
         showValues();
      }

      function setClip(field) {
         var clipVal = parseInt(field.value);
         document.display.clip[field.name] = clipVal;
         showValues();
      }
      function showValues() {
         var form = document.layers[0].document.forms[0];
         var propName;
         for (var i = 0; i < form.elements.length; i++) {
            propName = form.elements[i].name;
            if (form.elements[i].type == "text") {
               form.elements[i].value = document.display.clip[propName];
            }
         }
      }
      var intervalID;
      function revealClip() {
         var midWidth = Math.round(origLayerWidth /2);
         var midHeight = Math.round(origLayerHeight /2);
         document.display.clip.left = midWidth;
         document.display.clip.top = midHeight;
         document.display.clip.right = midWidth;
         document.display.clip.bottom = midHeight;
         intervalID = setInterval("stepClip()",1);
      }
      function stepClip() {
         var widthDone = false;
         var heightDone = false;
         if (document.display.clip.left > 0) {
            document.display.clip.left += -2;
            document.display.clip.right += 2;
         } else {
            widthDone = true;
         }
         if (document.display.clip.top > 0) {
            document.display.clip.top += -1;
            document.display.clip.bottom += 1;
         } else {
            heightDone = true;
         }
         showValues();
         if (widthDone && heightDone) {
            clearInterval(intervalID);
         }
```

```
      }
      </script>
   </head>
   <body onload="initializeXY()">
      <h1>Layer Clipping Properties</h1>
      <hr />
      Enter new clipping values to adjust the visible area of the layer.
      <layer top="130" left="5">
         <form>
            <table>
               <tr>
                  <td align="right">layer.clip.left:</td>
                  <td><input type="text" name="left" size="3"
                     onchange="setClip(this)" /></td>
               </tr>
               <tr>
                  <td align="right">layer.clip.top:</td>
                  <td><input type="text" name="top" size="3"
                     onchange="setClip(this)" /></td>
               </tr>
               <tr>
                  <td align="right">layer.clip.right:</td>
                  <td><input type="text" name="right" size="3"
                     onchange="setClip(this)" /></td>
               </tr>
               <tr>
                  <td align="right">layer.clip.bottom:</td>
                  <td><input type="text" name="bottom" size="3"
                     onchange="setClip(this)" /></td>
               </tr>
               <tr>
                  <td align="right">layer.clip.width:</td>
                  <td><input type="text" name="width" size="3"
                     onchange="setClip(this)" /></td>
               </tr>
               <tr>
                  <td align="right">layer.clip.height:</td>
                  <td><input type="text" name="height" size="3"
                     onchange="setClip(this)" /></td>
               </tr>
            </table>
            <input type="button" value="Reveal Original Layer"
            onclick="revealClip()" />
         </form>
      </layer>
      <layer name="display" bgcolor="coral" top="130" left="200" width="360"
      height="180">
         <h2>ARTICLE I</h2>
         <p>Congress shall make no law respecting an establishment of religion,
            or prohibiting the free exercise thereof; or abridging the freedom
            of speech, or of the press; or the right of the people peaceably to
            assemble, and to petition the government for a redress of
            grievances.</p>
      </layer>
   </body>
</html>
```

Listing 39-4 has a lot of other scripting in it to demonstrate a couple of other clip area techniques. After the document loads, the `onload` event handler initializes two global variables that represent the starting height and width of the layer as determined by the `clip.height` and `clip.width` properties. Because the `<layer>` tag does not specify any `clip` attributes, the *`layerObject`*`.clip` region is ensured of being the same as the layer's dimensions at load time.

I preserve the initial values for a somewhat advanced set of functions that act in response to the Reveal Original Layer button. The goal of this button is to temporarily shrink the clipping area to nothing and then expand the clip rectangle gradually from the very center of the layer. The effect is analogous to a zoom-out visual effect.

The clip region shrinks to practically nothing by setting all four edges to the same point midway along the height and width of the layer. The script then uses `setInterval()` to control the animation in `setClip()`. To make the zoom even on both axes, I first make sure that the initial size of the layer is an even ratio: twice as wide as it is tall. Each time through the `setClip()` function, the `clip.left` and `clip.right` values are adjusted in their respective directions by 2 pixels and `clip.top` and `clip.bottom` are adjusted by 1 pixel.

To make sure the animation stops when the layer is at its original size, I check whether the `clip.top` and `clip.left` values are their original zero values. If they are, I set a Boolean variable for each side. When both variables indicate that the clip rectangle is its original size, the script cancels the `setInterval()` action.

Listing 39-15 demonstrates how to adjust clipping in IE5+ and NN6+ syntax.

Related Items: `layer.pageX`, `layer.pageY` properties; `layer.resizeTo()` method.

document

Value: `document` object. Read-Only
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

Your scripts practically never have to retrieve the `document` property of a layer. But it is important to remember that it is always there as the actual container of content in the layer. As described at length in the opening section about the layer object, the `document` object reference plays a large role in assembling addresses to content items and properties in other layers. A document inside a layer has the same powers, properties, and methods of the main document in the browser window or in a frame.

Related Items: `document` object.

left
top

Value: Integer. Read/Write
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

The *`layerObject`*`.left` and *`layerObject`*`.top` properties correspond to the `left` and `top` attributes of the `<layer>` tag. These integer values determine the horizontal and vertical pixel coordinate point of the top-left corner of the layer relative to the browser window, frame, or parent layer in which it lives. The coordinate system of the layer's most immediate container is the one that these properties reflect.

Adjustments to these properties reposition the layer without adjusting its size. Clipping area values are untouched by changes in these properties. Thus, if you create a draggable layer object that needs to follow a dragged mouse pointer in a straight line along the x or y axis, it is more convenient to adjust one of these properties than to use the *`layerObject`*`.moveTo()` method.

IE4+ and NN6+ provide various properties to determine the coordinate location of a positioned element — all through the `style` object.

Example

To enable you to experiment with manually setting *`layerObject`*`.top` and *`layerObject`*`.left` properties, Listing 39-5 is a modified version of the `layer.clip` example (Listing 39-4). The current example again has the one modifiable layer, but it has only four text fields in which you can enter values. Two fields are for the *`layerObject`*`.left` and *`layerObject`*`.top` properties; the other two are for the *`layerObject`*`.clip.left` and *`layerObject`*`.clip.top` properties. I present both sets of values here to help reinforce the lack of connection between layer and clip location properties in the same layer object.

You can find the corresponding syntax for IE5+ and NN6+ in Listing 39-16.

Listing 39-5: **Comparison of Layer and Clip Location Properties**

```
<html>
   <head>
      <title>Layer vs. Clip</title>
      <script type="text/javascript">
      function setClip(field) {
         var clipVal = parseInt(field.value);
         document.display.clip[field.name] = clipVal;
         showValues();
      }
      function setLayer(field) {
         var layerVal = parseInt(field.value);
         document.display[field.name] = layerVal;
         showValues();
      }
      function showValues() {
         var form = document.layers[0].document.forms[0];
         form.elements[0].value = document.display.left;
         form.elements[1].value = document.display.top;
         form.elements[2].value = document.display.clip.left;
         form.elements[3].value = document.display.clip.top;
      }
      </script>
   </head>
   <body onload="showValues()">
      <b>Layer vs. Clip Location Properties</b>
      <hr />
      Enter new layer and clipping values to adjust the layer.
      <layer top="80" left="5">
         <form>
            <table>
               <tr>
                  <td align="right">layer.left:</td>
                  <td><input type="text" name="left" size="3"
                     onchange="setLayer(this)" /></td>
               </tr>
               <tr>
                  <td align="right">layer.top:</td>
                  <td><input type="text" name="top" size="3"
                     onchange="setLayer(this)" /></td>
```

Continued

Listing 39-5 *(continued)*

```
            </tr>
            <tr>
               <td align="right">layer.clip.left:</td>
               <td><input type="text" name="left" size="3"
                  onchange="setClip(this)" /></td>
            </tr>
            <tr>
               <td align="right">layer.clip.top:</td>
               <td><input type="text" name="top" size="3"
                  onchange="setClip(this)" /></td>
            </tr>
         </table>
      </form>
   </layer>
   <layer name="display" bgcolor="coral" top="80" left="200" width="360"
   height="180">
      <h2>ARTICLE I</h2>
      <p>Congress shall make no law respecting an establishment of religion,
         or prohibiting the free exercise thereof; or abridging the freedom
         of speech, or of the press; or the right of the people peaceably to
         assemble, and to petition the government for a redress of
         grievances.</p>
   </layer>
</body>
</html>
```

Related Items: `layer.clip`, `layer.parentLayer` properties.

name

Value: String. Read-Only
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

The *`layerObject`*`.name` property reflects the `name` attribute of the `<layer>` tag or name you assign to a positioned `div` or `span` element. This property is read-only. If you don't assign a name to a layer when you create it, Navigator generates a name for the layer in this format:

`js_layer_`*nn*

Here, *nn* is a serial number. That serial number is not the same every time the page loads, so you cannot rely on the automatically generated name to help you script an absolute reference to the layer.

pageX
pageY

Value: Integer. Read/Write
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

In Netscape's coordinate terminology, the *page* is the content area of a document. The top-left corner of the page space is point 0,0, and you can position any layer (including a nested layer) on the page relative to this corner. In the `<layer>` tag, the attributes that enable authors to set the position are `pagex` and `pagey`. These values are retrievable and modifiable as the *`layerObject`*`.pageX` and *`layerObject`*`.pageY` properties, respectively. Note the capitalization of the final letters of these property names.

The *`layerObject`*`.pageX` and *`layerObject`*`.pageY` values are identical to *`layerObject`*`.left` and `layer`*`Object`*`.top` only when the layer in question is at the main document level. That's because the *`layerObject`*`.left` and *`layerObject`*`.top` values are measured by the next higher container's coordinate system — which, in this case, is the same as the page.

The situation gets more interesting when you're dealing with nested layers. For a nested layer, the *`layerObject`*`.pageX` and *`layerObject`*`.pageY` values are still measured relative to the page, while *`layerObject`*`.left` and *`layerObject`*`.top` are measured relative to the next higher layer. If trying to conceive of these differences makes your head hurt, the example in Listing 39-6 should help clear things up for you.

Adjusting the *`layerObject`*`.pageX` and *`layerObject`*`.pageY` values of any layer has the same effect as using the *`layerObject`*`.moveToAbsolute()` method, which measures its coordinate system based on the page. If you create flying layers on your page, you can't go wrong by setting the *`layerObject`*`.pageX` and *`layerObject`*`.pageY` properties (or using the `moveToAbsolute()` method) in your script. That way, should you add another layer in the hierarchy between the base document and the flying layer, the animation is in the same coordinate system as before the new layer was added.

IE4+ does not provide a pair of properties to determine the location of a positioned element relative to the page, but the `offsetLeft` and `offsetTop` properties provide coordinates within the element's next outermost positioning context. Thus, you may have to "walk" the `offsetParent` trail to accumulate complete coordinate values. In NN6+/W3C, the `offsetLeft` and `offsetTop` properties use the page as the positioning context.

Example

Listing 39-6 defines one outer layer and one nested inner layer of different colors. The inner layer contains some text content; the outer layer is sized initially to present a colorful border by being below the inner layer and 10 pixels wider and taller.

Two sets of fields display (and enable you to change) the *`layerObject`*`.pageX`, *`layerObject`*`.pageY`, *`layerObject`*`.left`, and *`layerObject`*`.top` properties for each of the nested layers. Each set of fields is color-coded to its corresponding layer.

When you change any value, all values are recalculated and displayed in the other fields. For example, the initial `pageX` position for the outer layer is 200 pixels; for the inner layer, the `pageX` value is 205 pixels (accounting for the 5-pixel "border" around the inner layer). If you change the outer layer's `pageX` value to 220, the outer layer moves to the right by 20 pixels, taking the inner layer along for the ride. The `layer.pageX` value for the inner layer after the move is 225 pixels.

The outer layer values for the pairs of values are always the same no matter what. But for the inner layer, the page values are significantly different from the `layer.left` and `layer.top` values because these latter values are measured relative to their containing layer — the outer layer. If you move the outer layer, the inner layer values for *`layerObject`*`.left` and *`layerObject`*`.top` don't change one iota. Listing 39-17 shows the comparable syntax for IE5+ and NN6+/W3C.

Listing 39-6: **Testing Nested Layer Coordinate Systems**

```
<html>
   <head>
      <title>Nested Layer PageX/PageY</title>
      <script type="text/javascript">
      function setOuterPage(field) {
         var layerVal = parseInt(field.value);
         document.outerDisplay[field.name] = layerVal;
         showValues();
      }
      function setOuterLayer(field) {
         var layerVal = parseInt(field.value);
         document.outerDisplay[field.name] = layerVal;
         showValues();
      }
      function setInnerPage(field) {
         var layerVal = parseInt(field.value);
         document.outerDisplay.document.innerDisplay[field.name] = layerVal;
         showValues();
      }
      function setInnerLayer(field) {
         var layerVal = parseInt(field.value);
         document.outerDisplay.document.innerDisplay[field.name] = layerVal;
         showValues();
      }
      function showValues() {
         var form = document.layers[0].document.forms[0];
         form.elements[0].value = document.outerDisplay.pageX;
         form.elements[1].value = document.outerDisplay.pageY;
         form.elements[2].value = document.outerDisplay.left;
         form.elements[3].value = document.outerDisplay.top;
         form.elements[4].value =
            document.outerDisplay.document.innerDisplay.pageX;
         form.elements[5].value =
            document.outerDisplay.document.innerDisplay.pageY;
         form.elements[6].value =
            document.outerDisplay.document.innerDisplay.left;
         form.elements[7].value =
            document.outerDisplay.document.innerDisplay.top;
      }
      </script>
   </head>
   <body onload="showValues()">
      <b>Coordinate Systems for Nested Layers</b>
      <hr />
      Enter new page and layer coordinates for the <font color="coral">outer
      layer</font> and <font color="aquamarine">inner layer</font> objects.
      <layer top="80" left="5">
         <form>
            <table>
               <tr>
                  <td align="right" bgcolor="coral">
                     layer.pageX:
                  </td>
                  <td bgcolor="coral">
                     <input type="text" name="pageX" size="3"
                     onchange="setOuterPage(this)" />
                  </td>
```

```
            </tr>
            <tr>
               <td align="right" bgcolor="coral">
                  layer.pageY:
               </td>
               <td bgcolor="coral">
                  <input type="text" name="pageY" size="3"
                  onchange="setOuterPage(this)" />
               </td>
            </tr>
            <tr>
               <td align="right" bgcolor="coral">
                  layer.left:
               </td>
               <td bgcolor="coral">
                  <input type="text" name="left" size="3"
                  onchange="setOuterLayer(this)" /></td>
            </tr>
            <tr>
               <td align="right" bgcolor="coral">layer.top:</td>
               <td bgcolor="coral"><input type="text" name="top" size="3"
                  onchange="setOuterLayer(this)" /></td>
            </tr>
            <tr>
               <td align="right" bgcolor="aquamarine">layer.pageX:</td>
               <td bgcolor="aquamarine"><input type="text" name="pageX"
                  size="3" onchange="setInnerPage(this)" /></td>
            </tr>
            <tr>
               <td align="right" bgcolor="aquamarine">layer.pageY:</td>
               <td bgcolor="aquamarine"><input type="text" name="pageY"
                  size="3" onchange="setInnerPage(this)" /></td>
            </tr>
            <tr>
               <td align="right" bgcolor="aquamarine">layer.left:</td>
               <td bgcolor="aquamarine"><input type="text" name="left"
                  size="3" onchange="setInnerLayer(this)" /></td>
            </tr>
            <tr>
               <td align="right" bgcolor="aquamarine">layer.top:</td>
               <td bgcolor="aquamarine"><input type="text" name="top"
                  size="3" onchange="setInnerLayer(this)" /></td>
            </tr>
         </table>
      </form>
   </layer>
   <layer name="outerDisplay" bgcolor="coral" top="80" left="200"
   width="370" height="190">
      <layer name="innerDisplay" bgcolor="aquamarine" top="5" left="5"
      width="360" height="180">
         <h2>ARTICLE I</h2>
         <p>Congress shall make no law respecting an establishment of
            religion, or prohibiting the free exercise thereof; or abridging
            the freedom of speech, or of the press; or the right of the
            people peaceably to assemble, and to petition the government for
            a redress of grievances.</p>
      </layer>
   </layer>
</body>
</html>
```

Related Items: `layer.left, layer.top`, `window.innerHeight`, `window.innerWidth` properties; `layer.moveToAbsolute()` method.

parentLayer

Value: Object. Read-Only
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

Every layer has a parent that contains that layer. In the case of a layer defined at the main document level, its parent layer is the window or frame containing that document (the "page"). For this kind of layer, the *`layerObject`*`.parentLayer` property object is a `window` object. But for any nested layer contained by a layer, the `parentLayer` property is a layer object.

Be aware of the important distinction between *`layerObject`*`.parentLayer` and *`layerObject`*`.below`. As a parent layer can contain multiple layers in the next containment level, each of those layers' `parentLayer` properties evaluate to that same parent layer. But because each layer object is its own physical layer among the stack of layers on a page, the `layer.below` property in each layer points to a different object — the layer next lower in z-order.

Keeping the direction of things straight can get confusing. On the one hand, you have a layer's parent, which, by connotation, is higher up the hierarchical chain of layers. On the other hand, the order of physical layers is such that a parent more than likely has a lower z-order than its children because it is defined earlier in the document.

Use the *`layerObject`*`.parentLayer` property to assemble references to other nested layers. See the discussion about layer references at the beginning of this chapter for several syntax examples.

IE4+ offers an `offsetParent` property, which comes close to the functionality of the *`layerObject`*`.parentLayer` property.

Related Items: `layer.above`, `layer.below` properties.

siblingAbove
siblingBelow

(See `layer.above` and `layer.below` properties earlier in this chapter)

src

Value: String. Read/Write
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

Content for a layer may come from within the document that defines the layer or from an external source, such as an HTML or image file. If defined by a `<layer>` tag, an external file is specified by the `src` attribute. This attribute is reflected by the *`layerObject`*`.src` property.

The value of this property is a string of the URL of the external file. If you do not specify an `src` attribute in the `<layer>` tag, the value returns `null`. Do not set this property to an empty string in an effort to clear the layer of content: `document.write()` or load an empty page instead. Otherwise, the empty string is treated like a URL, and it loads the current client directory.

You can, however, change the content of a layer by loading a new source file into the layer. Simply assign a new URL to the *`layerObject`*`.src` property. Again, if a layer has nested layers inside it, those nested layers are blown away by the content that loads into the layer whose `src` property you change. The new file, of course, can be an HTML file that defines its own nested layers, which then become part of the page's object model.

Netscape also provides the *layerObject*.load() method to insert new content into a layer. One advantage of this method is that an optional second parameter enables you to redefine the width of the layer at the same time you specify a new document. But if you are simply replacing the content in the same width layer, you can use either way of loading new content.

Be aware that the height and width of a replacement layer are governed as much by their hard-coded content size as by the initial loading of any layer. For example, if your layer is initially sized at a width of 200 pixels and your replacement layer document includes an image whose width is set to 500 pixels, the layer expands its width to accommodate the larger content — unless you also restrict the view of the layer via the *layerObject*.clip properties. Similarly, longer text content flows beyond the bottom of the previously sized layer unless restricted by clipping properties.

Positioned elements in IE4+ and NN6+/W3C do not provide a direct way to load external content into them. That's what the W3C sees as the purpose of the iframe element. Even so, as Listing 39-18 shows, you can script your way around this limitation if it's absolutely necessary.

Example

Setting the *layerObject*.src property of a layer that is a member of a layer family (that is, a family with at least one parent and one child) can be tricky business if you're not careful. Listing 39-7 presents a workspace for you to see how changing the src property of outer and inner layers affects the scenery.

When you first load the document, one outer layer contains one inner layer (each with a different background color). Control buttons on the page enable you to set the *layerObject*.src property of each layer independently. Changes to the inner layer content affect only that layer. Long content forces the inner layer to expand its depth, but the inner layer's view is automatically clipped by its parent layer.

Changing the outer layer content, however, removes the inner layer completely. Code in the following listing shows one way to examine for the presence of a particular layer before attempting to load new content in it. If the inner layer doesn't exist, the script creates a new layer on the fly to replace the original inner layer.

Listing 39-7: **Setting Nested Layer Source Content**

```
<html>
   <head>
      <title>Layer Source</title>
      <script type="text/javascript">
      function loadOuter(doc) {
         document.outerDisplay.src = doc;
      }
      function loadInner(doc) {
         var nested = document.outerDisplay.document.layers;
         if (nested.length > 0) {
            // inner layer exists, so load content or restore
            if (doc) {
               nested[0].src = doc;
            } else {
               restoreInner(nested[0]);
            }
         } else {
            // prompt user about restoring inner layer
```

Continued

Listing 39-7 *(continued)*

```
            if (confirm("The inner layer has been removed by loading an outer
               document. Restore the original layers?")) {
               restoreLayers(doc);
            }
         }
      }
      function restoreLayers(doc) {
         // reset appearance of outer layer
         document.outerDisplay.bgColor = "coral";
         document.outerDisplay.resizeTo(370,190); // sets clip
         document.outerDisplay.document.write("");
         document.outerDisplay.document.close();
         // generate new inner layer
         var newInner = new Layer(360, document.layers["outerDisplay"]);
         newInner.bgColor = "aquamarine";
         newInner.moveTo(5,5);
         if (doc) {
            // user clicked an inner content button
            newInner.src = doc;
         } else {
            // return to pristine look
            restoreInner(newInner);
         }
         newInner.visibility = "show";
      }
      function restoreInner(inner) {
         inner.document.write("<html><body><p><b>Placeholder text for raw inner
            layer.<\/b><\/p><\/body><\/html>");
         inner.document.close();
         inner.resizeTo(360,180); // sets clip
      }
      </script>
   </head>
   <body>
      <b>Setting the <tt>layer.src</tt> Property of Nested Layers</b>
      <hr />
      Click the buttons to see what happens when you load new source documents
      into the <font color="coral">outer layer</font> and <font
      color="aquamarine">inner layer</font> objects.
      <layer top="100" bgcolor="coral">
         <form>
            Load into outer layer:<br />
            <input type="button" value="Article I"
            onclick="loadOuter('article1.htm')" /><br />
            <input type="button" value="Entire Bill of Rights"
            onclick="loadOuter('bofright.htm')" /><br />
         </form>
      </layer>
      <layer top="220" bgcolor="aquamarine">
         <form>
            Load into inner layer:<br />
            <input type="button" value="Article I"
            onclick="loadInner('article1.htm')" /><br />
            <input type="button" value="Entire Bill of Rights"
```

```
            onclick="loadInner('bofright.htm')" /><br />
            <input type="button" value="Restore Original"
            onclick="loadInner()" /><br />
         </form>
      </layer>
      <layer name="outerDisplay" bgcolor="coral" top="100" left="200"
      width="370" height="190">
         <layer name="innerDisplay" bgcolor="aquamarine" top="5" left="5"
         width="360" height="180">
            <p><b>Placeholder text for raw inner layer.</b></p>
         </layer>
      </layer>
   </body>
</html>
```

Restoring the original layers via script (as opposed to reloading the document) does not perform a perfect restoration. The key difference is that the scripts use the *layerObject*.resizeTo() method to set the layers to the height and width established by the <layer> tags that create the layers in the first place. This method, however, sets the clipping rectangle of the layer — not the layer's size. Therefore, if you use the script to restore the layers, loading the longer text file into either layer does not force the layer to expand to display all the content; the clipping region governs the view.

Related Items: layer.load(), layer.resizeTo() methods.

visibility

Value: String. Read/Write
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

A layer's visibility property can use one of three settings: show, hide, or inherit — the same values you can assign to the visibility attribute of the <layer> tag. But NN4 also enables you to set the property to hidden and visible, which are the values for the style.visibility property used in IE4+ and NN6+/W3C.

Unlike many other layer properties, you can set the visibility property such that a layer can either follow the behavior of its parent or strike out on its own. By default, a layer's visibility property is set to inherit, which means the layer's visibility is governed solely by that of its parent (and of *its* parent, if the nesting includes many layers). When the governing parent's property is, say, hide, the child's property remains inherit. Thus, you cannot tell whether an inheriting layer is presently visible or not without checking up the hierarchy (with the help of the *layerObject*.parentLayer property). However, you can override the parent's behavior by setting the current layer's property explicitly to show or hide. This action does not alter in any way other parent-child relationships between layers.

Example

Use the page in Listing 39-8 to see how the *layerObject*.visibility property settings affect a pair of nested layers. When the page first loads, the default inherit setting is in effect. Changes you make to the outer layer by clicking the outer layer buttons affect the inner layer, but setting the inner layer's properties to hide or show severs the visibility relationship between parent and child. Listing 39-19 shows this example with IE5+ and NN6+/W3C DOM syntax.

Listing 39-8: **Nested Layer Visibility Relationships**

```
<html>
   <head>
      <title>Layer Source</title>
      <script type="text/javascript">
      function setOuterVis(type) {
         document.outerDisplay.visibility = type;
      }
      function setInnerVis(type) {
         document.outerDisplay.document.innerDisplay.visibility = type;
      }
      </script>
   </head>
   <body>
      <b>Setting the <tt>layer.visibility</tt> Property of Nested Layers</b>
      <hr />
      Click the buttons to see what happens when you change the visibility of
      the <font color="coral">outer layer</font> and <font
      color="aquamarine">inner layer</font> objects.
      <layer top="100" bgcolor="coral">
         <form>
            Control outer layer property:<br />
            <input type="button" value="Hide Outer Layer"
            onclick="setOuterVis('hide')" /><br />
            <input type="button" value="Show Outer Layer"
            onclick="setOuterVis('show')" /><br />
         </form>
      </layer>
      <layer top="220" bgcolor="aquamarine">
         <form>
            Control inner layer property:<br />
            <input type="button" value="Hide Inner Layer"
            onclick="setInnerVis('hide')" /><br />
            <input type="button" value="Show Inner Layer"
            onclick="setInnerVis('show')" /><br />
            <input type="button" value="Inherit Outer Layer"
            onclick="setInnerVis('inherit')" /><br />
         </form>
      </layer>
      <layer name="outerDisplay" bgcolor="coral" top="100" left="200"
      width="370" height="190">
         <layer name="innerDisplay" bgcolor="aquamarine" top="5" left="5"
         width="360" height="180">
            <p><b>Placeholder text for raw inner layer.</b></p>
         </layer>
      </layer>
   </body>
</html>
```

zIndex

Value: Integer. Read/Write
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

Close relationships exist among the *layerObject*.above, *layerObject*.below, and *layerObject*.zIndex properties. When you define a layer in a document with the <layer> tag, you can supply only one of the three attributes (above, below, and z-index). After the layer is generated with any one of those attributes, the document object model automatically assigns values to at least two of those properties (*layerObject*.above and *layerObject*.below) unless you specify the z-index attribute; in this case, all three properties are assigned to the layer. If you don't specify any of these properties, the physical stacking order of the layers is the same as in the HTML document. The *layerObject*.above and *layerObject*.below properties are set as described in their discussion earlier in this chapter. But the *layerObject*.zIndex properties for all layers are zero.

The CSS attribute is spelled with a hyphen after the "z." Because a JavaScript property name cannot contain a hyphen, the character was removed for the property name. The capital "I" is important because JavaScript properties are case-sensitive.

Changes to layer*Object*.zIndex values affect the stacking order only of sibling layers. You can assign the same value to two layers, but the last layer to have its *layerObject*.zIndex property set lies physically above the other one. Therefore, if you want to ensure a stacking order, set the zIndex values for all layers within a container. Each value should be a unique number.

Stacking order is determined simply by the value of the integer assigned to the property. If you want to stack three sibling layers, the order is the same if you assign them the values of 1, 2, 3 or 10, 13, 50. As you modify a *layerObject*.zIndex value, the *layerObject*.above and *layerObject*.below properties for all affected layers change as a result.

Avoid setting zIndex property values to negative numbers in NN4. Negative values are treated as their absolute (positive) values for ordering.

For IE4+ and NN6+/W3C, the style.zIndex property controls z-order.

Example

The relationships among the three stacking property values can be difficult to visualize. Listing 39-9 offers a way to see the results of changing the *layerObject*.zIndex properties of three overlapping sibling layers.

The sequence of the <layer> tags in the document governs the original stacking order. Because the attribute is not set in the HTML, the initial values appear as zero for all three layers. But, as the page reveals, the *layerObject*.above and *layerObject*.below properties are automatically established. When a layer has no other layer object above it, the page shows (none). Also, if the layer below the bottom of the stack is the main window, a strange inner layer name is assigned (something like _js_layer_21).

To experiment with this page, first make sure you understand the *layerObject*.above and *layerObject*.below readings for the default order of the layers. Then, assign different orders to the layers with value sequences such as 3-2-1, 1-3-2, 2-2-2, and so on. Each time you enter one new value, check the actual layers to see if their stacking order changed and how that affected the other properties of all layers.

Listing 39-20 shows how to achieve the same action with IE5+ and NN6+/W3C syntax.

Listing 39-9: **Relationships Among zIndex, above, and below**

```
<html>
   <head>
      <title>Layer zIndex</title>
      <script type="text/javascript">
      function setZ(field) {
         switch (field.name) {
         case "top" :
            document.top.zIndex = parseInt(field.value);
            break;
         case "mid" :
            document.middle.zIndex = parseInt(field.value);
            break;
         case "bot" :
            document.bottom.zIndex = parseInt(field.value);
         }
         showValues();
      }
      function showValues() {
         document.layers[0].document.forms[0].bot.value =
            document.bottom.zIndex;
         document.layers[1].document.forms[0].mid.value =
            document.middle.zIndex;
         document.layers[2].document.forms[0].top.value = document.top.zIndex;

         document.layers[0].document.forms[0].above.value =
            (document.bottom.above) ? document.bottom.above.name : "(none)";
         document.layers[1].document.forms[0].above.value =
            (document.middle.above) ? document.middle.above.name : "(none)";
         document.layers[2].document.forms[0].above.value =
            (document.top.above) ? document.top.above.name : "(none)";

         document.layers[0].document.forms[0].below.value =
            (document.bottom.below) ? document.bottom.below.name : "(none)";
         document.layers[1].document.forms[0].below.value =
            (document.middle.below) ? document.middle.below.name : "(none)";
         document.layers[2].document.forms[0].below.value =
            (document.top.below) ? document.top.below.name : "(none)";
      }
      </script>
   </head>
   <body onload="showValues()">
      <b>Setting the <tt>layer.zIndex</tt> Property of Sibling Layers</b>
      <hr />
      Enter new zIndex values to see the effect on three layers.
      <layer top="90" left="5" width="240" bgcolor="coral">
         <form>
            Control Original Bottom Layer:<br />
            <table>
               <tr>
                  <td align="right">Layer zIndex:</td>
                  <td><input type="text" name="bot" size="3"
                     onchange="setZ(this)" /></td>
               </tr>
```

```
         <tr>
            <td align="right">Layer above:</td>
            <td><input type="text" name="above" size="13" /></td>
         </tr>
         <tr>
            <td align="right">Layer below:</td>
            <td><input type="text" name="below" size="13" /></td>
         </tr>
      </table>
   </form>
</layer>
<layer top="220" left="5" width="240" bgcolor="aquamarine">
   <form>
      Control Original Middle Layer:<br />
      <table>
         <tr>
            <td align="right">Layer zIndex:</td>
            <td><input type="text" name="mid" size="3"
               onchange="setZ(this)" /></td>
         </tr>
         <tr>
            <td align="right">Layer above:</td>
            <td><input type="text" name="above" size="13" /></td>
         </tr>
         <tr>
            <td align="right">Layer below:</td>
            <td><input type="text" name="below" size="13" /></td>
         </tr>
      </table>
   </form>
</layer>
<layer top="350" left="5" width="240" bgcolor="yellow">
   <form>
      Control Original Top Layer:<br />
      <table>
         <tr>
            <td align="right">Layer zIndex:</td>
            <td><input type="text" name="top" size="3"
               onchange="setZ(this)" /></td>
         </tr>
         <tr>
            <td align="right">Layer above:</td>
            <td><input type="text" name="above" size="13" /></td>
         </tr>
         <tr>
            <td align="right">Layer below:</td>
            <td><input type="text" name="below" size="13" /></td>
         </tr>
      </table>
   </form>
</layer>
<layer name="bottom" bgcolor="coral" top="90" left="260" width="300"
height="190">
   <p><b>Original Bottom Layer</b></p>
</layer>
<layer name="middle" bgcolor="aquamarine" top="110" left="280"
```

Continued

Listing 39-9 *(continued)*

```
      width="300" height="190">
         <p><b>Original Middle Layer</b></p>
      </layer>
      <layer name="top" bgcolor="yellow" top="130" left="300" width="300"
      height="190">
         <p><b>Original Top Layer</b></p>
      </layer>
   </body>
</html>
```

Related Items: `layer.above`, `layer.below` properties; `layer.moveAbove()`, `layer.moveBelow()` methods.

Methods

`load("URL", newLayerWidth)`

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

One way to change the content of an NN4 layer after it loads is to use the *`layerObject`*`.load()` method. This method has an advantage over setting the *`layerObject`*`.src` property because the second parameter is a new layer width for the content if one is desired. If you don't specify the second parameter, a small default value is substituted for you (unless the new document hard-wires widths to its elements that must expand the current width). If you are concerned about a new document being too long for the existing height of the layer, use the *`layerObject`*`.resizeTo()` method or set the individual *`layerObject`*`.clip` properties before loading the new document. This keeps the viewable area of the layer at a fixed size.

IE4+ and NN6+/W3C object models don't have a method like this, but you can work around the situation (as shown in Listing 39-18) and then adjust the `style.width` property of the positioned element.

Example

Buttons in Listing 39-10 enable you to load short and long documents into a layer. The first two buttons don't change the width (in fact, the second parameter to *`layerObject`*`.load()` is the *`layerObject`*`.clip.left` value). For the second two buttons, a narrower width than the original is specified. Click the Restore button frequently to return to a known state.

Listing 39-10: **Loading Documents into Layers**

```
<html>
   <head>
      <title>Layer Loading</title>
      <script type="text/javascript">
      function loadDoc(URL,width) {
         if (!width) {
            width = document.myLayer.clip.width;
         }
```

```
            document.myLayer.load(URL, width);
         }
         </script>
      </head>
      <body>
         <b>Loading New Documents</b>
         <hr />
         <layer top="90" width="240" bgcolor="yellow">
            <form>
               Loading new documents:<br />
               <input type="button" value="Small Doc/Existing Width"
               onclick="loadDoc('article1.htm')" /><br />
               <input type="button" value="Large Doc/Existing Width"
               onclick="loadDoc('bofright.htm')" />
               <p><input type="button" value="Small Doc/Narrower Width"
                  onclick="loadDoc('article1.htm',200)" /><br />
                  <input type="button" value="Large Doc/Narrower Width"
                  onclick="loadDoc('bofright.htm',200)" /></p>
               <p><input type="button" value="Restore"
                  onclick="location.reload()" /></p>
            </form>
         </layer>
         <layer name="myLayer" bgcolor="yellow" top="90" left="300" width="300"
         height="190">
            <p><b>Text loaded in original document.</b></p>
         </layer>
      </body>
   </html>
```

Related Item: layer.src property.

moveAbove(*layerObject*)
moveBelow(*layerObject*)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

With the exception of the *layerObject*.zIndex property, the layer object does not let you adjust properties that affect the global stacking order of layers. The *layerObject*.moveAbove() and *layerObject*.moveBelow() methods enable you to adjust a layer in relation to another layer object. Both layers in the transaction must be siblings — they must be in the same container, whether it be the base document window or some other layer. You cannot move existing layers from one container to another; you must delete the layer from the source and create a new layer in the destination. Neither of these methods affects the viewable size or coordinate system location of the layer.

The syntax for these methods is a little strange at first because the statement that makes these work has two layer object references in it. Named first in the statement (to the left of the method name, separated by a period) is the layer object you want to move. The sole parameter for each method is a reference to the layer object that is the physical reference point for the trip. For example, in this statement,

```
document.fred.moveAbove(document.ginger);
```

the instruction moves the fred layer above the ginger layer. The fred layer can be in any stacking relation to ginger; but, again, both layers must be in the same container.

Obviously, after one of these moves, the *layerObject*.above and *layerObject*.below properties of some or all layers in the container feel the ripple effects of the shift in order. If you have several layers that are out of order because of user interaction with your scripts, you can reorder them using these methods — or, more practically, by setting their *layerObject*.zIndex properties. In the latter case, it is easier to visualize through your code how the ordering is handled with increasing zIndex values for each layer.

There is no comparable method for IE4+ or NN6+/W3C.

Example

You can see the *layerObject*.moveAbove() method at work in Listing 39-1.

Related Items: layer.above, layer.below, layer.zIndex properties.

moveBy(*deltaX,deltaY*)
moveTo(*x,y*)
moveToAbsolute(*x,y*)

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

Much of what CSS-Positioning is all about is being able to precisely plant an element on a Web page. The unit of measure is the pixel, with the coordinate space starting at an upper-left corner at location 0,0. That coordinate space for a layer is typically the container (parent layer) for that layer. The *layerObject*.moveTo() and *layerObject*.moveBy() methods let scripts adjust the location of a layer inside that coordinate space — very much the way window.moveTo() and window.moveBy() work for window objects.

Moving a layer entails moving it (and its nested layers) without adjusting its size or stacking order. You can accomplish animation of a layer by issuing a series of *layerObject*.moveTo() methods if you know the precise points along the path. Or you can nudge the layer by increments in one or both axes with the *layerObject*.moveBy() method.

In case you need to position a layer with respect to the page's coordinate system (for example, you are moving items from multiple containers to a common point), the *layerObject*.moveToAbsolute() method bypasses the layer's immediate container. The 0,0 point for this method is the top-left corner of the document. Be aware, however, that you can move a layer to a position such that some or all of it lies out of range of the container's clip rectangle.

Moving positioned layers in IE4+ and NN6+/W3C requires adjusting the style.left and style.top properties (or the style.pixelLeft, style.pixelTop, style.posLeft, and style.posTop properties in IE4+).

Example

Listing 39-11 shows a demonstration of the *layerObject*.moveTo() method. It is a simple script that enables you to click and drag a layer around the screen. The script employs the coordinate values of the mousemove event; after compensating for the offset within the layer at which the click occurs, the script moves the layer to track the mouse action.

I want to present this example for an additional reason: to explain an important user interface difference between Windows and Macintosh versions of NN4. In Windows versions, you can click and hold the mouse button down on an object and let the object receive all the mousemove events as you drag the cursor around the screen. On the Macintosh, however, NN4 tries to

compensate for the lack of a second mouse button by popping up a context-sensitive menu at the cursor position when the user holds the mouse button down for more than just a click. To prevent the pop-up menu from appearing, the `engage()` method invoked by the `onmousedown` event handler ends with `return false`.

Notice in the following listing how the layer captures a number of mouse events. Each one plays an important role in creating a mode that is essentially like a `mouseStillDown` event (which doesn't exist in NN4's event model). The `mousedown` event sets a Boolean flag (`engaged`) indicating that the user clicked down in the layer. At the same time, the script records how far away from the layer's top-left corner the `mousedown` event occurred. This offset information is needed so that any setting of the layer's location takes this offset into account (otherwise, the top-left corner of the layer would jump to the cursor position and be dragged from there).

During the drag (`mousedown` events firing with each mouse movement), the `dragIt()` function checks whether the drag mode is engaged. If so, the layer is moved to the page location calculated by subtracting the original downstroke offset from the `mousemove` event location on the page. When the user releases the mouse button, the `mouseup` event turns off the drag mode Boolean value.

Listing 39-21 shows a version of this example for IE5+ and NN6+/W3C.

Listing 39-11: **Dragging a Layer**

```
<html>
   <head>
      <title>Layer Dragging</title>
      <script type="text/javascript">
      var engaged = false;
      var offsetX = 0;
      var offsetY = 0;
      function dragIt(e) {
         if (engaged) {
            document.myLayer.moveTo(e.pageX - offsetX, e.pageY - offsetY);
         }
      }
      function engage(e) {
         engaged = true;
         offsetX = e.pageX - document.myLayer.left;
         offsetY = e.pageY - document.myLayer.top;
         return false;
      }
      function release() {
         engaged = false;
      }
      </script>
   </head>
   <body>
      <b>Dragging a Layer</b>
      <hr />
      <layer name="myLayer" bgcolor="lightgreen" top="90" left="100"
      width="300" height="190">
         <p><b>Drag me around the window.</b></p>
      </layer>
      <script type="text/javascript">
```

Continued

Listing 39-11 *(continued)*

```
      document.myLayer.captureEvents(Event.MOUSEDOWN | Event.MOUSEUP |
         Event.MOUSEMOVE);
      document.myLayer.onmousedown = engage;
      document.myLayer.onmouseup = release;
      document.myLayer.onmousemove = dragIt;
      </script>
   </body>
</html>
```

Related Items: `layer.resizeBy()`, `layer.resizeTo()`, `window.moveBy()`, `window.moveTo()` methods.

`resizeBy(deltaX,deltaY)`
`resizeTo(width,height)`

FTR:document.*layerObject*.resizeBy()

Returns: Nothing.
Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

The basic functionality and parameter requirements of the *`layerObject`*`.resizeBy()` and *`layerObject`*`.resizeTo()` methods are similar to the identically named methods of the `window` object. You should, however, be cognizant of some considerations unique to layers.

Unlike resizing a window, which causes all content to reflow to fit the new size, the layer sizing methods don't adjust the size of the layer. Instead, these methods control the size of the clipping region of the layer. Therefore, the content of the layer does not reflow automatically when you use these methods any more than it does when you change individual *`layerObject`*`.clip` values.

Another impact of this clipping region relationship deals with content that extends beyond the bounds of the layer. For example, if you provide `height` and `width` attributes to a `<layer>` tag, content that requires more space to display itself than those attribute settings afford automatically expands the viewable area of the layer. To rein in such runaway content, you can set the `clip` attribute. But because the layer resize methods adjust the clipping rectangle, outsized content doesn't overflow the `<layer>` tag's height and width settings. This may or may not be beneficial for you, depending on your design intentions. Adjusting the size of a layer with either method affects only the position of the right and bottom edges of the layer. The top-left location of the layer does not move.

Neither IE4+ nor NN6+/W3C provides a similar method, but you can accomplish the same effects by adjusting the `style` properties of a positioned element.

Example

It is important to understand the ramifications of content flow when these two methods resize a layer. Listing 39-12a (and the companion document Listing 39-12b) shows you how to set the lower-right corner of a layer to be dragged by a user for resizing the layer (much like grabbing the resize corner of a document window). Three radio buttons enable you to choose whether and when the content should be redrawn to the layer — never, after resizing, or during resizing.

Event capture is very much like that in Listing 39-11 for layer dragging. The primary difference is that drag mode is engaged only when the mouse event takes place in the region of the lower-right corner. A different kind of offset value is saved here because, for resizing, the script needs to know the mouse event offset from the right and bottom edges of the layer.

Condition statements in the `resizeIt()` and `release()` functions verify whether a specific radio button is checked to determine when (or if) the content should be redrawn. I designed this page with the knowledge that its content might be redrawn. Therefore, I built the content of the layer as a separate HTML document that loads in the `<layer>` tag.

Redrawing the content requires reloading the document into the layer. I use the *`layerObject`*`.load()` method because I want to send the current *`layerObject`*`.clip.width` as a parameter for the width of the clip region to accommodate the content as it loads.

An important point to know about reloading content into a layer is that all property settings for the layer's event capture are erased when the document loads. Overcoming this behavior requires setting the layer's `onload` event handler to set the layer's event capture mechanism. If the layer event capturing is specified as part of the statements at the end of the document, the layer ignores some important events needed for the dynamic resizing after the document reloads the first time.

As you experiment with the different ways to resize and redraw, you see that redrawing during resizing is a slow process because of the repetitive loading (from cache) needed each time. On slower client machines, it is easy for the cursor to outrun the layer region, causing the layer to not get `mouseOver` events at all. It may not be the best-looking solution, but I prefer to redraw after resizing the layer.

Listing 39-22 shows a version designed for the IE5+ and NN6+/W3C object models. Because content automatically reflows in those browsers, you do not have to load the content of the positioned element from an external document.

Listing 39-12a: **Resizing a Layer**

```
<html>
   <head>
      <title>Layer Resizing</title>
      <script type="text/javascript">
      var engaged = false;
      var offsetX = 0;
      var offsetY = 0;
      function resizeIt(e) {
         if (engaged) {
            document.myLayer.resizeTo(e.pageX + offsetX, e.pageY + offsetY);
            if (document.forms[0].redraw[2].checked) {
               document.myLayer.load("lst39-12b.htm",
                  document.myLayer.clip.width);
            }
         }
      }
      function engage(e) {
         if (e.pageX > (document.myLayer.clip.right - 10) &&
            e.pageY > (document.myLayer.clip.bottom - 10)) {
            engaged = true;
            offsetX = document.myLayer.clip.right - e.pageX;
            offsetY = document.myLayer.clip.bottom - e.pageY;
```

Continued

Listing 39-12a *(continued)*

```
         }
      }
      function release() {
         if (engaged && document.forms[0].redraw[1].checked) {
            document.myLayer.load("lst39-12b.htm",
               document.myLayer.clip.width);
         }
         engaged = false;
      }
      function grabEvents() {
         document.myLayer.captureEvents(Event.MOUSEDOWN | Event.MOUSEUP |
            Event.MOUSEMOVE);
      }
      </script>
   </head>
   <body>
      <b>Resizing a Layer</b>
      <hr />
      <form>
         Redraw layer content:<br />
         <input type="radio" name="redraw" checked="checked" />Never <input
         type="radio" name="redraw" />After resize <input type="radio"
         name="redraw" />During resize
      </form>
      <layer name="myLayer" src="lst39-12b.htm" bgcolor="lightblue" top="120"
      left="100" width="300" height="190" onload="grabEvents()">
      </layer>
      <script type="text/javascript">
      document.myLayer.onmousedown = engage;
      document.myLayer.onmouseup = release;
      document.myLayer.onmousemove = resizeIt;
      </script>
   </body>
</html>
```

Listing 39-12b: **Content for the Resizable Layer**

```
<html>
   <head>
      <title></title>
   </head>
   <body>
      <p><b>Resize me by dragging the lower right corner.</b></p>
      <script type="text/javascript">
      if (navigator.userAgent.indexOf("Mac") != -1) {
         document.write("(Mac users: Ctrl-Click me first; then Click to stop
            dragging.)");
      }
      </script>
   </body>
</html>
```

Related Items: `layer.moveBy()`, `layer.moveTo()`, `window.resizeBy()`, `window.resizeTo()` methods.

Event handlers

onblur
onfocus

Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

A user gets no visual cue when a layer receives focus. But a click on the clipping region of a layer triggers a `focus` event that can be handled with an `onFocus` event handler. Clicking anywhere on the page outside of that layer area fires a `blur` event. Changing the stacking order of sibling layers does not fire either event unless mouse activity occurs in one of the layers.

If your layer contains elements that have their own `focus` and `blur` events (such as text fields), those objects' event handlers still fire even if you also have the same event handlers defined for the layer. The layer's events fire after the text field's events.

Unlike comparable event handlers in windows, layer events for `blur` and `focus` do not have companion methods to bring a layer into focus or to blur it. However, if you use the `focus()` and/or `blur()` methods on applicable form elements in a layer, the layer's corresponding event handlers are triggered as a result.

Related Items: `textbox.blur()`, `textbox.focus()` methods.

onload

Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

Scripting layers can sometimes lead to instances of unfortunate sequences of loading. For example, if you want to set some layer object properties via a script (that is, not in the `<layer>` tag), you can do so only after the layer object exists in the document object model. One way to make sure the object exists is to place the scripting in `<script>` tags at the end of the document. Another way is to specify an `onload` event handler in the tag, as shown in Listing 39-12a.

Each time you load a document into the layer — either via the `src` attribute in the `<layer>` tag or by invoking the *`layerObject`*`.load()` method — the `onload` event handler runs. But also be aware that an interaction occurs between a layer's `onload` event handler and an `onload` event handler in the `<body>` tag of a document loaded into a layer. If the document body has an `onload` event handler, the layer's `onload` event handler does not fire. You get one or the other, but not both.

Related Item: `window.onload` event handler.

onmouseout
onmouseover

Compatibility: WinIE-, MacIE-, NN4, Moz-, Safari-

A layer knows when the cursor rolls into and out of its clipping region. Like several other objects in the document object model, the layer object has `onmouseover` and `onmouseout` event handlers that enable you to perform any number of actions in response to those user activities. Typically, a layer's `onmouseover` event handler changes colors, hides, or shows pseudo-borders devised of colored layers behind the primary layer; sometimes, it even changes the text or image content. The status bar is also available to plant plain-language legends about the purpose of the layer or offer other relevant help.

Both events occur only once per entrance to, and egress from, a layer's region by the cursor. If you want to script actions dependent upon the location of the cursor in the layer, you can use *layerObject*.captureEvents() to grab mousemove and all types of mouse button events. This kind of event capture generates an event object (see Chapter 25) that includes information about the coordinate position of the cursor at the time of the event.

Related Items: link.onmouseout, link.onmouseover, area.onmouseout, area.onmouseover event handlers.

Positioned Elements in the Modern DOM

With the dwindling NN4 installed base, you can focus on applying "layer" techniques in browsers whose object models expose every element of an object and whose rendering engines automatically reflow content in response to changes. This section follows the sequence of examples in the discussion about NN4's layer object but shows you how to accomplish the same operations and learn the behavior of positioned elements in IE4+, NN6+, and W3C-compatible browsers.

An important facet that these newer browsers have in common is the style property of every renderable element object. Most adjustments to the location, layering, size, and visibility of positioned elements use the style object associated with each element. Cross-browser complications ensue, however, with some aspects of nested layers. Plus, there is the ever-present difference between the IE- and NN-class browsers with respect to the event objects — how to reference the event object and the names of its properties. Some of the examples that follow have more code in them than their corresponding NN4 layer version shown earlier in this chapter. Most of the additional code concerns itself with accommodating the different event object models.

One more point about the following examples: The syntax adopted for references to element objects uses the W3C DOM document.getElementById() method, which is supported in IE5+, NN6+/Mozilla, and Safari. If you intend to apply any of the techniques in these examples to applications that run exclusively in an IE environment (and must be compatible with IE4), you can substitute the document.all referencing syntax.

Changing element backgrounds

Listing 39-13 demonstrates the syntax and behavior of setting background images via the style.backgroundImage property. Note the CSS-style syntax for the URL value assigned to the style.backgroundImage property. It's a good lesson to learn that most style properties are strings, and their values are in the same format as the values normally assigned in a stylesheet definition.

Removing a background image requires setting the URL to null. Also, a background image overlays whatever color (if any) you assign to the element. If the background image has transparent regions, the background color shows through.

Listing 39-13: **Setting Layer Backgrounds (W3C)**

```
<html>
  <head>
     <title>Layer Backgrounds (W3C)</title>
     <script type="text/javascript">
     function setBg(URL) {
        document.getElementById("bgExpo").style.backgroundImage =
```

```
            "url(" + URL + ")";
      }
      </script>
   </head>
   <body>
      <h1>Layer Backgrounds (W3C)</h1>
      <hr />
      <div id="buttons" style="position:absolute; top:100">
         <form>
            <input type="button" value="The Usual"
            onclick="setBg('cr_kraft.gif')" /><br />
            <input type="button" value="A Big One"
            onclick="setBg('arch.gif')" /><br />
            <input type="button" value="Not So Usual"
            onclick="setBg('wh86.gif')" /><br />
            <input type="button" value="Decidedly Unusual"
            onclick="setBg('sb23.gif')" /><br />
            <input type="button" value="Quick as..."
            onclick="setBg('lightnin.gif')" />
            <p><input type="button" value="Remove Image"
               onclick="setBg(null)" /><br /></p>
         </form>
      </div>
      <div id="bgExpo"
      style="position:absolute; top:100; left:250; width:300; height:260;
         background-color:gray">
         <span style="font-weight:bold; color:white">Some text, which may or
         may not read well with the various backgrounds.</span>
      </div>
   </body>
</html>
```

Listing 39-14 focuses on background color. A color palette is laid out as a series of rectangles. As the user rolls atop a color in the palette, the color is assigned to the background of the layer. Because of the regularity of the `div` elements generated for the palette, this example uses scripts to write them dynamically to the page as the page loads. This lets the `for` loop handle all the positioning math based on initial values set as global variables.

Perhaps of more interest here than the background color setting is the event handling. First of all, because the target browsers all employ event bubbling, the page lets a single event handler at the document level wait for `mouseover` events to bubble up to the document level. But because the `mouseover` event of every element on the page bubbles there, the event handler must filter the events and process only those on the palette elements.

The `setColor()` method begins by equalizing the IE4+ and NN6+/W3C DOM event object models. If an object is assigned to the `evt` parameter variable, that means the W3C DOM browser is processing the event; otherwise, it's IE4+ — meaning that the `window.event` object contains the event information. Whichever browser performs the processing, the event object is assigned to the `evt` variable. After verifying that a valid event triggered the function, the next step is to equalize the different, event-model–specific property names for the event's target element. For W3C DOM browsers, the property is `target`, while IE4+ uses `srcElement`. The final validation is to check the `className` property of the event's target element. Because all elements acting as palette colors share the same `class` attribute, the `className` property is examined. If the value is `palette`, the `mouseover` event has occurred on one of the colors. Now it's time to extract the target element's `style.backgroundColor` property and assign that color to the same property of the main positioned element.

Listing 39-14: **Layer Background Colors (W3C)**

```
<html>
  <head>
     <title>Layer Background Colors (W3C)</title>
     <script type="text/javascript">
     function setColor(evt) {
        evt = (evt) ? evt : (window.event) ? window.event : "";
        if (evt) {
           var elem = (evt.target) ? evt.target : evt.srcElement;
           if (elem.className == "palette") {
              document.getElementById("display").style.backgroundColor =
                 elem.style.backgroundColor;
           }
        }
     }
     document.onmouseover = setColor;
     </script>
  </head>
  <body>
     <h1>Layer Background Colors (W3C)</h1>
     <hr />
     <script type="text/javascript">
     var oneLayer;
     var colorTop = 100;
     var colorLeft = 20;
     var colorWidth = 40;
     var colorHeight = 40;
     var colorPalette = new Array("aquamarine","coral","forestgreen",
        "goldenrod","red","magenta","navy","teal");
     for (var i = 0; i < colorPalette.length; i++) {
        oneLayer = "<div id='swatch" + i + "' class='palette'";
        oneLayer += "style='position:absolute; top:" + colorTop + ";";
        oneLayer += "left:" + ((colorWidth * i) + colorLeft) + ";";
        oneLayer += "width:" + colorWidth + "; height:" + colorHeight + ";";
        oneLayer += "background-color:" + colorPalette[i] + "'><\/div>\n";
        document.write(oneLayer);
     }
     </script>
     <div id="display"
     style="position:absolute; top:150; left:80; width:200; height:200;
        background-color:gray">
        <span style="font-weight:bold; color:white; text-align:center">Some
        reversed text to test against background colors.</span>
     </div>
  </body>
</html>
```

Layer clipping

Working with clipping rectangles is a bit cumbersome using CSS syntax because the object model standard does not provide separate readouts or controls over individual edges of a clipping rectangle. IE5+ enables you to read individual edge dimensions via the `currentStyle` object (for example, `currentStyle.clipTop`), but these properties are read-only.

Based on these limitations, Listing 39-15 is implemented in a way that, for the sake of convenience, preserves the current clipping rectangle edge values as global variables. Any adjustments to individual edge values are first recorded in those variables (in the `setClip()` function), and then the `style.clip` property is assigned the long string of values in the required format (in the `adjustClip()` function). The `showValues()` function reads the variable values and displays updated values after making the necessary calculations for the width and height of the clipping rectangle.

As a demonstration of a "reveal" visual effect (which you can carry out more simply in WinIE4+ via a transition filter), the `revealClip()` function establishes beginning clip values at the midpoints of the width and height of the layer. Then the `setInterval()` method loops through `stepClip()` until the clipping rectangle dimensions match those of the layer.

Listing 39-15: **Adjusting Layer clip Properties (W3C)**

```
<html>
   <head>
      <title>Layer Clip</title>
      <script type="text/javascript">
      var origLayerWidth = 0
      var origLayerHeight = 0
      var currTop, currRight, currBottom, currLeft
      function init() {
         origLayerWidth =
            parseInt(document.getElementById("display").style.width);
         origLayerHeight =
            parseInt(document.getElementById("display").style.height);
         currTop = 0;
         currRight = origLayerWidth;
         currBottom = origLayerHeight;
         currLeft = 0;
         showValues();
      }

      function setClip(field) {
         var val = parseInt(field.value);
         switch (field.name) {
         case "top" :
            currTop = val;
            break;
         case "right" :
            currRight = val;
            break;
         case "bottom" :
            currBottom = val;
            break;
         case "left" :
            currLeft = val;
            break;
         case "width" :
            currRight = currLeft + val;
            break;
         case "height" :
            currBottom = currTop + val;
            break;
         }
```

Continued

Listing 39-15 *(continued)*

```
        adjustClip();
        showValues();
    }

    function adjustClip() {
        document.getElementById("display").style.clip = "rect(" + currTop +
            "px " + currRight + "px " + currBottom + "px " + currLeft + "px)";
    }

    function showValues() {
        var form = document.forms[0];
        form.top.value = currTop;
        form.right.value = currRight;
        form.bottom.value = currBottom;
        form.left.value = currLeft;
        form.width.value = currRight - currLeft;
        form.height.value = currBottom - currTop;
    }
    var intervalID;
    function revealClip() {
        var midWidth = Math.round(origLayerWidth /2);
        var midHeight = Math.round(origLayerHeight /2);
        currTop = midHeight;
        currBottom = midHeight;
        currRight = midWidth;
        currLeft = midWidth;
        intervalID = setInterval("stepClip()",1);
    }
    function stepClip() {
        var widthDone = false;
        var heightDone = false;
        if (currLeft > 0) {
            currLeft += -2;
            currRight += 2;
        } else {
            widthDone = true;
        }
        if (currTop > 0) {
            currTop += -1;
            currBottom += 1;
        } else {
            heightDone = true;
        }
        adjustClip();
        showValues();
        if (widthDone && heightDone) {
            clearInterval(intervalID);
        }
    }
    </script>
</head>
<body onload="init()">
    <h1>Layer Clipping Properties (W3C)</h1>
    <hr />
    Enter new clipping values to adjust the visible area of the layer.
    <div style="position:absolute; left:10; top:130">
        <form>
```

```
            <table>
               <tr>
                  <td align="right">layer.style.clip (left):</td>
                  <td><input type="text" name="left" size="3"
                     onchange="setClip(this)" /></td>
               </tr>
               <tr>
                  <td align="right">layer.style.clip (top):</td>
                  <td><input type="text" name="top" size="3"
                     onchange="setClip(this)" /></td>
               </tr>
               <tr>
                  <td align="right">layer.style.clip (right):</td>
                  <td><input type="text" name="right" size="3"
                     onchange="setClip(this)" /></td>
               </tr>
               <tr>
                  <td align="right">layer.style.clip (bottom):</td>
                  <td><input type="text" name="bottom" size="3"
                     onchange="setClip(this)" /></td>
               </tr>
               <tr>
                  <td align="right">layer.style.clip (width):</td>
                  <td><input type="text" name="width" size="3"
                     onchange="setClip(this)" /></td>
               </tr>
               <tr>
                  <td align="right">layer.style.clip (height):</td>
                  <td><input type="text" name="height" size="3"
                     onchange="setClip(this)" /></td>
               </tr>
            </table>
            <input type="button" value="Reveal Original Layer"
            onclick="revealClip()" />
         </form>
      </div>
      <div id="display"
      style="position:absolute; top:130; left:220; width:360; height:180;
         clip:rect(0px 360px 180px 0px); background-color:coral">
         <h2>ARTICLE I</h2>
         <p>Congress shall make no law respecting an establishment of religion,
            or prohibiting the free exercise thereof; or abridging the freedom
            of speech, or of the press; or the right of the people peaceably to
            assemble, and to petition the government for a redress of
            grievances.</p>
      </div>
   </body>
</html>
```

Listing 39-16 enables you to compare the results of adjusting a clipping rectangle versus the size of a positioned element. This example goes a bit further than the corresponding NN4 layer version (Listing 39-5) in that it enables you to adjust the dimensions of the entire layer (via the `style.left` and `style.right` properties) as well as the right and bottom edges of the clipping rectangle associated with the layer. As a bonus, the code includes a function that converts the `style.clip` string into an object representing the rectangle of the clipping rectangle (in other words, with four properties, one for each edge). Values from that `rectangle` object populate two of the fields on the page, providing dynamic readouts of the clipping rectangle's right and bottom edges.

Global variables still temporarily store the clipping rectangle values so that the `adjustClip()` function can operate just as it does in Listing 39-15. Note that the clipping rectangle is explicitly defined in the stylesheet rule for the positioned element. This is necessary for the element's `style.clip` property to have some values with which to start.

Listing 39-16: Comparison of Layer and Clip Location Properties (W3C)

```
<html>
   <head>
      <title>Layer vs. Clip</title>
      <script type="text/javascript">
      var currClipTop = 0;
      var currClipLeft = 0;
      var currClipRight = 360;
      var currClipBottom = 180;
      function setClip(field) {
         var val = parseInt(field.value);
         switch (field.name) {
         case "clipBottom" :
            currClipBottom = val;
            break;
         case "clipRight" :
            currClipRight = val;
            break;
         }
         adjustClip();
         showValues();
      }
      function adjustClip() {
         document.getElementById("display").style.clip =
            "rect(" + currClipTop + "px " + currClipRight + "px " +
            currClipBottom + "px " + currClipLeft + "px)";
      }

      function setLayer(field) {
         var val = parseInt(field.value);
         switch (field.name) {
         case "width" :
            document.getElementById("display").style.width = val + "px";
            break;
         case "height" :
            document.getElementById("display").style.height = val + "px";
            break;
         }
         showValues();
      }
      function showValues() {
         var form = document.forms[0];
         var elem = document.getElementById("display");
         var clipRect = getClipRect(elem);
         form.width.value = parseInt(elem.style.width);
         form.height.value = parseInt(elem.style.height);
         form.clipRight.value = clipRect.right;
         form.clipBottom.value = clipRect.bottom;
      }
      // convert clip property string to an object
      function getClipRect(elem) {
         var clipString = elem.style.clip;
```

```
         // assumes "rect(npx, npx, npx, npx)" form
         // get rid of "rect("
         clipString = clipString.replace(/rect\(/,"");
         // get rid of "px)"
         clipString = clipString.replace(/px\)/,"");
         // get rid of remaining "px" strings
         clipString = clipString.replace(/px/g,",");
         // turn remaining string into an array
         clipArray = clipString.split(",");
         // make object out of array values
         var clipRect = {top:parseInt(clipArray[0]),
            right:parseInt(clipArray[1]), bottom:parseInt(clipArray[2]),
            left:parseInt(clipArray[3])};
         return clipRect;
      }
      </script>
   </head>
   <body onload="showValues()">
      <h1>Layer vs. Clip Dimension Properties (W3C)</h1>
      <hr />
      Enter new layer and clipping values to adjust the layer.
      <div style="position:absolute; left:10; top:130">
         <form>
            <table>
               <tr>
                  <td align="right">layer.style.width:</td>
                  <td><input type="text" name="width" size="3"
                     onchange="setLayer(this)" /></td>
               </tr>
               <tr>
                  <td align="right">layer.style.height:</td>
                  <td><input type="text" name="height" size="3"
                     onchange="setLayer(this)" /></td>
               </tr>
               <tr>
                  <td align="right">layer.style.clip (right):</td>
                  <td><input type="text" name="clipRight" size="3"
                     onchange="setClip(this)" /></td>
               </tr>
               <tr>
                  <td align="right">layer.style.clip (bottom):</td>
                  <td><input type="text" name="clipBottom" size="3"
                     onchange="setClip(this)" /></td>
               </tr>
            </table>
         </form>
      </div>
      <div id="display"
      style="position:absolute; top:130; left:250; width:360; height:180;
         clip:rect(0px, 360px, 180px, 0px); background-color:coral">
         <h2>ARTICLE I</h2>
         <p>Congress shall make no law respecting an establishment of religion,
            or prohibiting the free exercise thereof; or abridging the freedom
            of speech, or of the press; or the right of the people peaceably to
            assemble, and to petition the government for a redress of
            grievances.</p>
      </div>
   </body>
</html>
```

Scripting nested layers

Working with nested layer locations, especially in a cross-browser manner, presents numerous browser-specific syntax problems that need equalization to behave the same to all users. Some discrepancies even appear between Windows and Macintosh versions of IE.

The scenario for Listing 39-17 consists of one positioned layer (greenish) nested inside another (reddish). The inner layer is initially sized and positioned so that the outer layer extends 5 pixels in each direction. Text boxes enable you to adjust the coordinates for either layer relative to the entire page as well as the layer's positioning context. If you make a change to any one value, all the others are recalculated and displayed to show you the effect the change has on other coordinate values.

As you see when you load the page, the outer element's positioning context is the page, so the "page" and "container" coordinates are the same (although the calculations to achieve this equality are not so simple across all browsers). The inner layer's initial page coordinates are to the right and down 5 pixels in each direction, and the coordinates within the container show those 5 pixels.

Because of browser idiosyncrasies, calculating the coordinates within the page takes the most work. The `getGrossOffsetLeft()` and `getGrossOffsetTop()` functions perform those calculations in the page. Passed a reference to the positioned element to be measured, the first number to grab is whatever the browser returns as the `offsetLeft` or `offsetTop` value of the element (see Chapter 15). These values are independent of the `style` property, and they can report different values for different browsers. IE, for example, measures the offset with respect to whatever it determines as the next outermost positioning context. NN6+, on the other hand, treats the page as the positioning context regardless of nesting. So, as long as there is an `offsetParent` element, a `while` loop starts accumulating the `offsetLeft` measures of each succeeding offset parent element going outward from the element. But even before that happens, a correction for MacIE must be accounted for. If there is a difference between the `style.left` and `offsetLeft` property values of an element, that difference is added to the offset. In MacIE5, for example, failure to correct this results in the "page" and "container" values of the outer layer being 10 pixels different in each direction. Values returned from these two gross measures are inserted in the readouts for the "page" measures of both inner and outer elements.

Reading the coordinates relative to each element's "container" is easy: The `style.left` and `style.top` properties have the correct values for all browsers. Moving a layer with respect to its positioning context (the "container" values) is equally easy: assign the entered values to the same `style.left` and `style.top` properties.

Moving the layers with respect to the page coordinate planes (via the `setOuterPage()` and `setInnerPage()` functions) involves going the long way to assign values that take each browser's positioning idiosyncrasies into account. The way you move a positioned element (cross-browser, anyway) is to assign a value to the `style.left` and `style.top` properties. These values are relative to their positioning context, but NN6+ doesn't offer any shortcuts to reveal what element is the positioning context for a nested element. Calls to the `getNetOffsetLeft()` and `getNetOffsetTop()` functions do the inverse of the `getGrossOffsetLeft()` and `getGrossOffsetTop()` functions. Because the values received from the text box are relative to the entire page, the values must have any intervening positioning contexts subtracted from that value in order to achieve the net positioning values that can be applied to the `style.left` and `style.top` properties. To get there, however, a call to the `getParentLayer()` function cuts through the browser-specific implementations of container references to locate the positioning context so that its coordinate values can be subtracted properly. The same kind of

correction for MacIE is required here as in the gross offset calculations; but here, the correction is subtracted from the value that eventually is returned as the value for either the `style.left` or `style.top` of the layer.

Let me add one quick word about the condition statements of the `while` constructions in the `getNetOffsetLeft()` and `getNetOffsetTop()` functions. You see here a construction not used frequently in this book, but one that is perfectly legal. When the conditional expression evaluates, the `getParentLayer()` method is invoked, and its returned value is assigned to the `elem` variable. That expression evaluates to the value returned by the function. As you can see from the `getParentLayer()` function definition, a value is returned as either an element reference or `null`. The `while` condition treats a value of `null` as `false`; any reference to an object is treated as `true`. Thus, the conditional expression does not use a comparison operator but rather executes some code and branches based on the value returned by that code. NN6+ reports JavaScript warnings (not errors) for this construction because it tries to alert you to a common scripting bug that occurs when you use the = operator when you really mean the == operator. But an NN warning is not the same as a script error, so don't be concerned when you see these messages in the JavaScript Console window during your debugging.

Listing 39-17: **Testing Nested Layer Coordinate Systems (W3C)**

```
<html>
   <head>
      <title>Nested Layer Coordinates (W3C)</title>
      <script type="text/javascript">
      // offsets within page
      function getGrossOffsetLeft(elem) {
         var offset = 0;
         while (elem.offsetParent) {
            // correct for MacIE discrepancy between offset and style
            // coordinates, but not if the parent is HTML element (NN6+)
            offset += (elem.offsetParent.tagName != "HTML") ?
               parseInt(elem.style.left) - parseInt(elem.offsetLeft) : 0;
            elem = elem.offsetParent;
            offset += elem.offsetLeft;
         }
         return offset;
      }
      function getGrossOffsetTop(elem) {
         var offset = 0;
         while (elem.offsetParent) {
            // correct for MacIE discrepancy between offset and style
            // coordinates, but not if the parent is HTML element (NN6+)
            offset += (elem.offsetParent.tagName != "HTML") ?
               parseInt(elem.style.top) - parseInt(elem.offsetTop) : 0;
            elem = elem.offsetParent;
            offset += elem.offsetTop;
         }
         return offset;
      }

      // offsets within element's positioning context
```

Continued

Listing 39-17 *(continued)*

```
function getNetOffsetLeft(offset, elem) {
   while (elem = getParentLayer(elem)) {
      // correct for MacIE discrepancy between offset and style
      // coordinates, but not if the parent is HTML element (NN6+)
      offset -= (elem.offsetParent.tagName != "HTML") ?
         parseInt(elem.style.left) - parseInt(elem.offsetLeft) : 0;
      offset -= elem.offsetLeft;
   }
   return offset;
}
function getNetOffsetTop(offset, elem) {
   while (elem = getParentLayer(elem)) {
      // correct for MacIE discrepancy between offset and style
      // coordinates, but not if the parent is HTML element (NN6+)
      offset -= (elem.offsetParent.tagName != "HTML") ?
         parseInt(elem.style.top) - parseInt(elem.offsetTop) : 0;
      offset -= elem.offsetTop;
   }
   return offset;
}
// find positioning context parent element
function getParentLayer(elem) {
   if (elem.parentNode) {
      while (elem.parentNode != document.body) {
         elem = elem.parentNode;
         while (elem.nodeType != 1) {
            elem = elem.parentNode;
         }
         if (elem.style.position == "absolute" || elem.style.position ==
            "relative") {
            return elem;
         }
         elem = elem.parentNode;
      }
      return null;
   } else if (elem.offsetParent && elem.offsetParent.tagName != "HTML") {
      return elem.offsetParent;
   } else {
      return null;
   }
}

// functions that respond to changes in text boxes
function setOuterPage(field) {
   var val = parseInt(field.value);
   var elem = document.getElementById("outerDisplay");
   switch (field.name) {
   case "pageX" :
      elem.style.left = ((elem.offsetParent) ?
         getNetOffsetLeft(val, elem) : val) + "px";
      break;
   case "pageY" :
      elem.style.top = ((elem.offsetParent) ?
         getNetOffsetTop(val, elem) : val) + "px";
      break;
   }
```

```
   showValues();
}
function setOuterLayer(field) {
   var val = parseInt(field.value);
   switch (field.name) {
   case "left" :
      document.getElementById("outerDisplay").style.left = val + "px";
      break;
   case "top" :
      document.getElementById("outerDisplay").style.top = val + "px";
      break;
   }
   showValues();
}
function setInnerPage(field) {
   var val = parseInt(field.value);
   var elem = document.getElementById("innerDisplay");
   switch (field.name) {
   case "pageX" :
      elem.style.left = ((elem.offsetParent) ?
         getNetOffsetLeft(val, elem) : val) + "px";
      break;
   case "pageY" :
      elem.style.top = ((elem.offsetParent) ?
         getNetOffsetTop(val, elem) : val) + "px";
      break;
   }
   showValues();
}
function setInnerLayer(field) {
   var val = parseInt(field.value);
   switch (field.name) {
   case "left" :
      document.getElementById("innerDisplay").style.left = val + "px";
      break;
   case "top" :
      document.getElementById("innerDisplay").style.top = val + "px";
      break;
   }
   showValues();
}
function showValues() {
   var form = document.forms[0];
   var outer = document.getElementById("outerDisplay");
   var inner = document.getElementById("innerDisplay");
   form.elements[0].value = outer.offsetLeft +
      ((outer.offsetParent) ? getGrossOffsetLeft(outer) : 0);
   form.elements[1].value = outer.offsetTop +
      ((outer.offsetParent) ? getGrossOffsetTop(outer) : 0);
   form.elements[2].value = parseInt(outer.style.left);
   form.elements[3].value = parseInt(outer.style.top);
   form.elements[4].value = inner.offsetLeft +
      ((inner.offsetParent) ? getGrossOffsetLeft(inner) : 0);
   form.elements[5].value = inner.offsetTop +
      ((inner.offsetParent) ? getGrossOffsetTop(inner) : 0);
   form.elements[6].value = parseInt(inner.style.left);
   form.elements[7].value = parseInt(inner.style.top);
}
```

Continued

Listing 39-17 *(continued)*

```
   </script>
</head>
<body onload="showValues()">
   <h1>Nested Layer Coordinates (W3C)</h1>
   <hr />
   Enter new page and layer coordinates for the <font color="coral">outer
   layer</font> and <font color="aquamarine">inner layer</font> objects.
   <div style="position:absolute; left:10; top:130">
      <form>
         <table>
            <tr>
               <td align="right" bgcolor="coral">Page X:</td>
               <td bgcolor="coral"><input type="text" name="pageX" size="3"
                  onchange="setOuterPage(this)" /></td>
            </tr>
            <tr>
               <td align="right" bgcolor="coral">Page Y:</td>
               <td bgcolor="coral"><input type="text" name="pageY" size="3"
                  onchange="setOuterPage(this)" /></td>
            </tr>
            <tr>
               <td align="right" bgcolor="coral">Container X:</td>
               <td bgcolor="coral"><input type="text" name="left" size="3"
                  onchange="setOuterLayer(this)" /></td>
            </tr>
            <tr>
               <td align="right" bgcolor="coral">Container Y:</td>
               <td bgcolor="coral"><input type="text" name="top" size="3"
                  onchange="setOuterLayer(this)" /></td>
            </tr>
            <tr>
               <td align="right" bgcolor="aquamarine">Page X:</td>
               <td bgcolor="aquamarine"><input type="text" name="pageX"
                  size="3" onchange="setInnerPage(this)" /></td>
            </tr>
            <tr>
               <td align="right" bgcolor="aquamarine">Page Y:</td>
               <td bgcolor="aquamarine"><input type="text" name="pageY"
                  size="3" onchange="setInnerPage(this)" /></td>
            </tr>
            <tr>
               <td align="right" bgcolor="aquamarine">Container X:</td>
               <td bgcolor="aquamarine"><input type="text" name="left"
                  size="3" onchange="setInnerLayer(this)" /></td>
            </tr>
            <tr>
               <td align="right" bgcolor="aquamarine">Container Y:</td>
               <td bgcolor="aquamarine"><input type="text" name="top"
                  size="3" onchange="setInnerLayer(this)" /></td>
            </tr>
         </table>
      </form>
   </div>
   <div id="outerDisplay"
   style="position:absolute; top:130; left:200; width:370; height:190;
```

```
      background-color:coral">
         <div id="innerDisplay"
         style="position:absolute; top:5; left:5; width:360; height:180;
         background-color:aquamarine">
            <h2>ARTICLE I</h2>
            <p>Congress shall make no law respecting an establishment of
               religion, or prohibiting the free exercise thereof; or abridging
               the freedom of speech, or of the press; or the right of the
               people peaceably to assemble, and to petition the government for
               a redress of grievances.</p>
         </div>
      </div>
   </body>
</html>
```

Try entering a variety of values in all text boxes to see what happens. Here is one possible sequence of tests and explanations:

1. Increase the red Page X value to 250. This moves the outer layer to the right by 50 pixels. Because the green layer is nested inside, it moves along with it. The green's Page X value also increases by 50, but its Container X value remains the same because the inner layer maintains the same relationship with the outer layer as before.
2. Increase the green Page X value to 300. This action shifts the position of the green inner layer by 45 pixels, making it a total of 50 pixels inset within its positioning context. Because the outer layer does not have its clipping rectangle set, the inner layer's content bleeds beyond the width of the red layer.
3. Set the Container Y value to –50. This action moves the green inner layer upward so that its top is 50 pixels above the top of its red container. As a result, the Page Y value of the inner layer is 80, while the Page Y value of the red outer layer remains at 130 (thus, the 50-pixel difference).

As you experiment with moving the layers around, you may encounter some screen refresh problems where traces of the inner layer remain when moved beyond the outer layer's rectangle. Take these bugs into account when you design the actions of your script-controlled positioning.

Loading external HTML into a layer

The NN4 `layer` object had an unfair advantage when it came to loading external content into it: the element was designed to do just that, acting in some ways like the W3C-endorsed `iframe` element.

Because the IE4+ and W3C DOM object models embrace the `iframe` element, using that element may be the easy way for you to designate a space within a page for external content. In fact, you can even assign a stylesheet rule that absolute-positions the `iframe` precisely on the page where you want it. Be sure to set the `frameborder` attribute to `0` unless you want the border to be visible to the user (and then watch out for content that may overrun the rectangle and cause scrollbars to appear). In this case, you must then leave all the formatting and stylesheet control of that content to the HTML loaded into the `iframe`, just as if it were in a separate window or frame. To load different content into the element, assign a different URL to the `src` property of the `iframe` element object.

As one more example that more closely simulates the loading of external content into a layer, Listing 39-18 demonstrates a somewhat ugly workaround that lets a layer's background color or image show through some kinds of HTML content. The technique works only in IE5.5+ and NN6+/W3C because these browser generations are the first to offer scripted access to the HTML you need to load into an intermediate (and hidden) `iframe` before stuffing the content into the layer.

A hidden `iframe` element is the initial recipient of the external HTML file, as loaded by the `loadOuter()` method. When that file loads, the `transferHTML()` method is invoked to copy the `innerHTML` of just the `body` element of the content window of the `iframe` (note the different syntax for NN6+ — the `contentDocument` property — and IE5.5+ — the `contentWindow` property). By eliminating the `body` element and any tags in the `head`, you prevent the tags in the layer from conflicting with the tags for the main document. As a result, however, notice how the background color set for the layer shows through the HTML plugged into the layer.

HTML element objects (other than `iframe`) were not designed to get their content from external files. But, as Listing 39-18 shows, where there is a will there is a way — even if the workaround isn't pretty.

Listing 39-18: **Setting Layer Source Content (W3C)**

```
<html>
   <head>
      <title>Loading External Content into a Layer (W3C)</title>
      <script type="text/javascript">
      function loadOuter(doc) {
         document.getElementById("hiddenContent").src = doc;
         // workaround for missing onload event in iframe for NN6+
         if (!document.getElementById("hiddenContent").onload) {
            setTimeout("transferHTML()", 1000);
         }
      }
      function transferHTML() {
         var srcFrame = document.getElementById("hiddenContent");
         var srcContent = (srcFrame.contentDocument) ?
            srcFrame.contentDocument.getElementsByTagName("BODY")[0].innerHTML
            : (srcFrame.contentWindow) ?
            srcFrame.contentWindow.document.body.innerHTML : "";
         document.getElementById("outerDisplay").innerHTML = srcContent;
      }
      </script>
   </head>
   <body>
      <h1>Loading External Content into a Layer (W3C)</h1>
      <hr />
      <p>Click the buttons to see what happens when you load new source
         documents into the <font color="coral">layer</font> object.</p>
      <div
      style="position:absolute; top:150; width:200; background-color:coral">
         <form>
            Load into outer layer:<br />
            <input type="button" value="Article I"
            onclick="loadOuter('article1.htm')" /><br />
            <input type="button" value="Entire Bill of Rights"
            onclick="loadOuter('bofright.htm')" /><br />
         </form>
      </div>
```

```
      <div id="outerDisplay"
      style="position:absolute; top:150; left:250; width:370; height:190;
         background-color:coral">
         <p><b>Placeholder text for layer.</b></p>
      </div>
      <iframe id="hiddenContent" style="visibility:hidden"
      onload="transferHTML()"></iframe>
   </body>
</html>
```

Positioned element visibility behavior

There is very little code in Listing 39-19 because it simply adjusts the `style.visibility` property of an outer layer and a nested, inner layer. You can see that when the page loads, the green inner layer's `visibility` is automatically set to inherit the `visibility` of its containing outer layer. When you click the outer layer buttons, the inner layer blindly follows the settings.

Things change, however, once you start adjusting the properties of the inner layer independently of the outer layer. With the outer layer hidden, you can show the inner layer (except in Safari 1.0). Only by setting the `visibility` property of the inner layer to `inherit` can you make it rejoin the outer layer in its behavior.

Listing 39-19: **Nested Layer Visibility Relationships (W3C)**

```
<html>
   <head>
      <title>layer.style.visibility (W3C)</title>
      <script type="text/javascript">
      function setOuterVis(type) {
         document.getElementById("outerDisplay").style.visibility = type;
      }
      function setInnerVis(type) {
         document.getElementById("innerDisplay").style.visibility = type;
      }
      </script>
   </head>
   <body>
      <h1>Setting the <tt>layer.style.visibility</tt> Property of Nested Layers
         (W3C)</h1>
      <hr />
      Click the buttons to see what happens when you change the visibility of
      the <font color="coral">outer layer</font> and <font
      color="aquamarine">inner layer</font> objects.
      <div
      style="position:absolute; top:150; width:180; background-color:coral">
         <form>
            Control outer layer property:<br />
            <input type="button" value="Hide Outer Layer"
            onclick="setOuterVis('hidden')" /><br />
            <input type="button" value="Show Outer Layer"
            onclick="setOuterVis('visible')" /><br />
         </form>
```

Continued

Listing 39-19 *(continued)*

```
      </div>
      <div
      style="position:absolute; top:270; width:180;
         background-color:aquamarine">
         <form>
            Control inner layer property:<br />
            <input type="button" value="Hide Inner Layer"
            onclick="setInnerVis('hidden')" /><br />
            <input type="button" value="Show Inner Layer"
            onclick="setInnerVis('visible')" /><br />
            <input type="button" value="Inherit Outer Layer"
            onclick="setInnerVis('inherit')" /><br />
         </form>
      </div>
      <div id="outerDisplay"
      style="position:absolute; top:150; left:200; width:370; height:190;
         background-color:coral">
         <div id="innerDisplay"
         style="position:absolute; top:5; left:5; width:360; height:180;
            background-color:aquamarine">
            <p><b>Placeholder text for raw inner layer.</b></p>
         </div>
      </div>
   </body>
</html>
```

Scripting layer stacking order

Listing 39-20 is simpler than its NN4 layer-specific version (Listing 39-9) because the W3C DOM, as implemented in IE4+ and NN6+/Mozilla/Safari, does not have properties that reveal the equivalent of the *layerObject*.above or *layerObject*.below properties. Therefore, Listing 39-20 confines itself to enabling you to adjust the style.zIndex property values of three overlapping layers. All three layers (none of which are nested inside another) initially set their zIndex values to 0, meaning that the source code order rules the stacking order.

Listing 39-20: **Relationships Among zIndex Values (W3C)**

```
<html>
   <head>
      <title>layer.style.zIndex</title>
      <script type="text/javascript">
      function setZ(field) {
         switch (field.name) {
         case "top" :
            document.getElementById("topLayer").style.zIndex =
               parseInt(field.value);
            break;
         case "mid" :
            document.getElementById("middleLayer").style.zIndex =
               parseInt(field.value);
```

```
            break;
         case "bot" :
            document.getElementById("bottomLayer").style.zIndex =
               parseInt(field.value);
         }
         showValues();
      }
      function showValues() {
         var botLayer = document.getElementById("bottomLayer");
         var midLayer = document.getElementById("middleLayer");
         var topLayer = document.getElementById("topLayer");

         document.forms[0].bot.value = botLayer.style.zIndex;
         document.forms[1].mid.value = midLayer.style.zIndex;
         document.forms[2].top.value = topLayer.style.zIndex;
      }
      </script>
   </head>
   <body onload="showValues()">
      <h1><tt>layer.style.zIndex</tt> Property of Sibling Layers</h1>
      <hr />
      Enter new zIndex values to see the effect on three layers.
      <div
      style="position:absolute; top:140; left:10; width:240;
         background-color:coral">
         <form>
            Control Original Bottom Layer:<br />
            <table>
               <tr>
                  <td align="right">Layer zIndex:</td>
                  <td><input type="text" name="bot" size="3"
                     onchange="setZ(this)" /></td>
               </tr>
            </table>
         </form>
      </div>
      <div
      style="position:absolute; top:220; left:10; width:240;
         background-color:aquamarine">
         <form>
            Control Original Middle Layer:<br />
            <table>
               <tr>
                  <td align="right">Layer zIndex:</td>
                  <td><input type="text" name="mid" size="3"
                     onchange="setZ(this)" /></td>
               </tr>
            </table>
         </form>
      </div>
      <div
      style="position:absolute; top:300; left:10; width:240;
         background-color:yellow">
         <form>
            Control Original Top Layer:<br />
            <table>
```

Continued

Listing 39-20 *(continued)*

```
            <tr>
               <td align="right">Layer zIndex:</td>
               <td><input type="text" name="top" size="3"
                  onchange="setZ(this)" /></td>
            </tr>
         </table>
      </form>
   </div>
   <div id="bottomLayer"
   style="position:absolute; top:140; left:260; width:300; height:190;
      z-Index:0; background-color:coral">
      <span><b>Original Bottom Layer</b></span>
   </div>
   <div id="middleLayer"
   style="position:absolute; top:160; left:280; width:300; height:190;
      z-Index:0; background-color:aquamarine">
      <span><b>Original Middle div</b></span>
   </div>
   <div id="topLayer"
   style="position:absolute; top:180; left:300; width:300; height:190;
      z-Index:0; background-color:yellow">
      <span><b>Original Top Layer</b></span>
   </div>
</body>
</html>
```

Dragging and resizing a layer

Listing 39-21 is an IE4+- and W3C-compatible version of the layer dragging example shown earlier in Listing 39-11. The basic structure is the same, with event handler functions for engaging the drag mode, handling the mouse movement while in drag mode, and releasing the element at the end of the journey.

There is a lot more code in this version for several reasons. The main reason is to accommodate the two event object models in IE and NN6+/W3C browsers. First of all, event bubbling is used so that all mouse events are handled at the document level. Thus, all of the event handlers need to equalize the `event` object and event target element, as well as filter events so that the action occurs only when a draggable element (as identified by its `className` property) is the target of the event action.

The toughest job involves the `engage()` function because it must use the two different event and element object models to establish the offset of the `mousedown` event within the draggable element. For WinIE, this also means taking the scrolling of the body into account. To get the element to reposition itself with mouse motion, the `dragIt()` function applies browser-specific coordinate values to the `style.left` and `style.top` properties of the draggable element. This function is invoked very frequently in response to the `mousemove` event.

Nothing in this example, however, treats the `zIndex` stacking order, which must be addressed if the page contains multiple, draggable items. See the map puzzle game in Chapter 56 on the CD-ROM for an example of processing multiple, draggable items.

Listing 39-21: **Dragging a Layer (W3C)**

```
<html>
   <head>
      <title>Layer Dragging</title>
      <style type="text/css">
      .draggable {cursor:hand}
      </style>
      <script type="text/javascript">
      var draggedElem;
      var offsetX = 0;
      var offsetY = 0;
      function dragIt(evt) {
         evt = (evt) ? evt : (window.event) ? window.event : "";
         var targElem = (evt.target) ? evt.target : evt.srcElement;
         if (draggedElem) {
            targElem = draggedElem;
            if (targElem.className == "draggable") {
               while (targElem.id != "myLayer" && targElem.parentNode) {
                  targElem = targElem.parentNode;
               }
               if (evt.pageX) {
                  targElem.style.left = evt.pageX - offsetX + "px";
                  targElem.style.top = evt.pageY - offsetY + "px";
               } else {
                  targElem.style.left = evt.clientX - offsetX + "px";
                  targElem.style.top = evt.clientY - offsetY + "px";
               }
               return false;
            }
         }
      }
      function engage(evt) {
         evt = (evt) ? evt : (window.event) ? window.event : "";
         var targElem = (evt.target) ? evt.target : evt.srcElement;
         if (targElem.className == "draggable") {
            while (targElem.id != "myLayer" && targElem.parentNode) {
               targElem = targElem.parentNode;
            }
            if (targElem.id == "myLayer") {
               draggedElem = targElem;
               if (evt.pageX) {
                  offsetX = evt.pageX - targElem.offsetLeft;
                  offsetY = evt.pageY - targElem.offsetTop;
               } else {
                  offsetX = evt.offsetX - document.body.scrollLeft;
                  offsetY = evt.offsetY - document.body.scrollTop;
                  if (navigator.userAgent.indexOf("Win") == -1) {
                     offsetX += document.body.scrollLeft;
                     offsetY += document.body.scrollTop;
                  }
               }
               return false;
```

Continued

Listing 39-21 *(continued)*

```
            }
         }
      }
      function release(evt) {
         evt = (evt) ? evt : (window.event) ? window.event : "";
         var targElem = (evt.target) ? evt.target : evt.srcElement;
         if (targElem.className == "draggable") {
            while (targElem.id != "myLayer" && targElem.parentNode) {
               targElem = targElem.parentNode;
            }
            if (draggedElem && targElem.id == "myLayer") {
               draggedElem = null;
            }
         }
      }
      </script>
   </head>
   <body>
      <h1>Dragging a Layer</h1>
      <hr />
      <div id="myLayer" class="draggable"
      style="position:absolute; top:90; left:100; width:300; height:190;
         background-color:lightgreen">
         <span class="draggable"><b>Drag me around the window.</b></span>
         <script type="text/javascript">
         document.onmousedown = engage;
         document.onmouseup = release;
         document.onmousemove = dragIt;
         </script>
      </div>
   </body>
</html>
```

The final listing in this section, Listing 39-22, applies many example components used thus far to let scripts control the resizing of a positionable element by dragging the lower-right, 20-pixel region. A lot of the hairy code in the `engage()` function is for determining if the `onmousedown` event occurs in the invisible 20-pixel square.

The `resizeIt()` function resembles the `dragIt()` function of Listing 39-21, but the adjustments are made to the width and height of the positionable element. A fair amount of math determines the width of the element in response to the cursor's instantaneous location and sets the `style.width` and `style.height` properties accordingly.

Listing 39-22: **Resizing a Layer (W3C)**

```
<html>
   <head>
      <title>Layer Resizing</title>
      <script type="text/javascript">
      var draggedElem;
      var offsetX = 0;
      var offsetY = 0;
```

```
function resizeIt(evt) {
   evt = (evt) ? evt : (window.event) ? window.event : "";
   var targElem = (evt.target) ? evt.target : evt.srcElement;
   if (draggedElem) {
      if (evt.pageX) {
         targElem.style.width = (evt.pageX - targElem.offsetLeft -
            offsetX) + "px";
         targElem.style.height = (evt.pageY - targElem.offsetTop -
            offsetY) + "px";
      } else {
         var elemWidth = evt.clientX - targElem.offsetLeft - offsetX -
            (parseInt(targElem.style.left) -
            parseInt(targElem.offsetLeft));
         var elemHeight = evt.clientY - targElem.offsetTop - offsetY -
            (parseInt(targElem.style.top) -
            parseInt(targElem.offsetTop));
         targElem.style.width = elemWidth + "px";
         targElem.style.height = elemHeight + "px";
      }
   }
}

function engage(evt) {
   evt = (evt) ? evt : (window.event) ? window.event : "";
   var targElem = (evt.target) ? evt.target : evt.srcElement;
   if (targElem.className == "draggable") {
      while (targElem.id != "myLayer" && targElem.parentNode) {
         targElem = targElem.parentNode;
      }
      if (targElem.id == "myLayer") {
         if (evt.pageX && (evt.pageX > ((parseInt(targElem.style.width) -
            20) + targElem.offsetLeft)) && (evt.pageY >
            ((parseInt(targElem.style.height) - 20) +
            targElem.offsetTop))) {
            offsetX = evt.pageX - parseInt(targElem.style.width) -
               targElem.offsetLeft;
            offsetY = evt.pageY - parseInt(targElem.style.height) -
               targElem.offsetTop;
            draggedElem = targElem;
         } else if ((evt.offsetX > parseInt(targElem.style.width) - 20)
            && (evt.offsetY > parseInt(targElem.style.height) - 20)) {
            offsetX = evt.offsetX - parseInt(targElem.style.width) -
               document.body.scrollLeft;
            offsetY = evt.offsetY - parseInt(targElem.style.height) -
               document.body.scrollTop;
            draggedElem = targElem;
            if (navigator.userAgent.indexOf("Win") == -1) {
               offsetX += document.body.scrollLeft;
               offsetY += document.body.scrollTop;
            }
         }
         return false;
      }
   }
}
function release(evt) {
   evt = (evt) ? evt : (window.event) ? window.event : "";
   var targElem = (evt.target) ? evt.target : evt.srcElement;
```

Continued

Listing 39-22 *(continued)*

```
         if (targElem.className == "draggable") {
            while (targElem.id != "myLayer" && targElem.parentNode) {
               targElem = targElem.parentNode;
            }
            if (draggedElem && targElem.id == "myLayer") {
               draggedElem = null;
            }
         }
      }
      </script>
   </head>
   <body>
      <h1>Resizing a Layer (W3C)</h1>
      <hr />
      <div id="myLayer" class="draggable"
      style="position:absolute; top:170; left:100; width:300; height:190;
         background-color:lightblue">
         <span>Here is some content inside the layer. See what happens to it as
         you resize the layer via the bottom-right 20-pixel handle.</span>
      </div>
      <script type="text/javascript">
      document.onmousedown = engage;
      document.onmouseup = release;
      document.onmousemove = resizeIt;
      </script>
   </body>
</html>
```

This chapter only scratches the surface in the kinds of positioned element actions you can control via scripts. You may have seen examples of positioned element scripting at sites around the Web. For example, some pages have subject headers fly into place — even "bounce" around until they settle into position. Or elements can go in circles or spirals to get your attention (or distract you, as the case may be). The authors of those tricks apply formulas from other disciplines (such as games programming) to the `style` object properties of a positioned element.

Sometimes the effects are there just for the sake of looking cool (at first, anyway) or because the page author knows how to script those effects. Your chief guide in implementing such features, however, should be whether the scripting genuinely adds value to the content offering. If you don't improve the content by adding a flying doo-dad or pulsating images, then leave them out. A greater challenge is finding meaningful ways to apply positioning techniques. Done the right way and for the right reason, they can significantly enhance the visitor's enjoyment of your application.

✦ ✦ ✦

CHAPTER 40

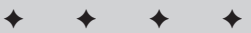
Embedded Objects

In This Chapter

Using `embed` element objects

Exploring the `object` element object

Understanding the unusual `param` element



In addition to the typical content that you see in Web pages — primarily text and images — you can embed other kinds of content into the page. Such embedded content usually requires the powers of additional software, such as plug-in players or other external code processors, to load and display the content. All of this external content is added to a page by one of three HTML elements: `applet`, `embed`, or `object`. In the HTML 4.0 standard, the `applet` element, which was intended originally for loading Java applets, is deprecated in favor of the newer `object` element. An `object` element is intended to be more extensible, meaning that it has enough attributes and power to summon the Java virtual machine if the incoming code is a Java applet, or run an ActiveX program (in IE for Windows, that is). The `embed` element is commonly used to display a plug-in control panel directly in the document, rather than having the panel appear in a separate window, but it, too, is gradually giving way to the `object` element.

In all cases, when a visual element is embedded via any of these elements, the control panel or applet occupies a segregated rectangular space on the page and generally confines its activities to that rectangle. But in some browsers and plug-in player types, JavaScript can also interact with the content or the player, allowing your scripts to extend themselves with powers for actions, such as controlling audio playback or the operation of a Java applet. Don't expect universal browser support for controlling plug-ins.

This chapter's primary focus is not on the content and players that you can control as it is on the HTML element objects that load the content or players into the page in the first place. Most of the properties represent nothing more than scriptable access to the element HTML attributes. The property descriptions in this chapter are therefore not extensive. Online HTML references (including the W3C HTML 4.0 specification and the Microsoft Developer Network documentation) should fill in the attribute value information quite well. In practice, scripts have very little interaction with these element objects, but if you ever need to know what's scriptable, you'll find that information here. As for controlling applets and plug-ins, you can find information about that in Chapter 44.

applet Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

applet

Properties	*Methods*	*Event Handlers*
align	(Applet methods)	oncellchange
alt		ondataavailable
altHTML		ondatasetchanged
archive		ondatasetcomplete
code		onload
codeBase		onrowenter
height		onrowexit
hspace		onrowsdelete
name		onrowsinserted
object		onscroll
vspace		
width		
(Applet variables)		

Syntax

Accessing `applet` element object properties or methods:

```
(NN3+/IE4+)    [window.]document.appletName.property | method([parameters])
(NN3+/IE4+)    [window.]document.applets[index].property | method([parameters])
(IE4+)         [window.]document.all.appletID.property | method([parameters])
(IE5+/W3C)     [window.]document.getElementById("appletID").property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari1+

About this object

The fact that the applet, itself, can expose public instance variables and public methods as properties and methods of the `applet` object means that the scriptable characteristics of an `applet` object are highly dependent upon the way the applet was written. You can learn more about how to compose an applet that exposes its innards to JavaScript in Chapter 44.

Perhaps the most important point to remember about accessing applets is that you must have them loaded and running before you can address them as objects. Although you cannot query an applet to find out whether it's loaded (as you can with an image), you can rely on the `onload` event handler of a window to fire only when all applets in the window are loaded and running (with the occasional version- or platform-specific bug in frames, as described in the `window.onload` event handler discussion in Chapter 16). IE4+ also features an `onload` event handler for the `applet` element directly, but applets tend to be the last things to load on a page. Therefore, you won't be able to use an applet embedded in a document to help you create the HTML content of that page as it loads, but an applet can provide content for new documents or for modifiable elements of a page. With the highly dynamic object models of IE4+ and W3C, this can lead to all kinds of possibilities.

Java applets have also been used to maintain contact with a server after the page has loaded by way of a servlet running on the server. A servlet allows the applet to query or be refreshed with instantaneously updated information without having to reload the page. Of course, getting a sophisticated applet to run in a wide range of browsers and operating systems is a challenge unto itself.

A large set of event handlers for this element (all but `onload` and `onscroll`) is related to the application of WinIE data binding for `param` elements nested inside an `applet` element. These events fire when a variety of actions occur to the data source or recordset associated with the applet. For more about applying data binding to an `applet` element, see `http://msdn.microsoft.com/workshop/author/databind/dataconsumer.asp`.

Properties

align

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `align` property controls either the horizontal or vertical alignment of the element with regard to surrounding content. String values of `left` or `right` cause the applet rectangle to cling to the left or right edges of its next outermost positioning context. String values of `absbottom`, `absmiddle`, `baseline`, `bottom`, `middle`, `texttop`, or `top` influence the vertical alignment with respect to adjacent text, with the same kind of results as corresponding values of the `style.verticalAlign` property.

Related Items: `style.verticalAlign` property.

alt

Value: String. Read/Write
Compatibility: WinIE6+, MacIE6+, NN6+, Moz1+, Safari1+

The `alt` property represents the `alt` attribute, which should contain text that displays in the browser in the event that the applet does not load or the user has Java turned off in the browser preferences. This information should be set as the `applet` element's attribute, because assigning text to the property after the applet attempts to load does not insert the text into the page.

Related Items: `altHTML` property.

altHTML

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `altHTML` property is supposed to provide an `applet` element with HTML content to render if the applet doesn't load. In practice, assigning an HTML string to this property has no effect on an `applet` element.

Related Items: `alt` property.

archive

Value: String. Read/Write
Compatibility: WinIE6+, MacIE-, NN6+, Moz1+, Safari+

The `archive` property represents the `archive` attribute, which points to the URL of a compressed (`.zip`) file containing Java class files needed for the applet. The archive must include the class file that is assigned to the `code` attribute to get the applet loaded and started.

Related Items: `code` property.

code

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `code` property is the URL string of the Java class file that is to begin loading the applet (or the property may be the entire applet if it consists of a single class file). You cannot change the code assigned to an applet after the element has loaded (even if the applet code did not load successfully).

Related Items: `codeBase` property.

codeBase

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `codeBase` property is the string of the path on the server to the Java class file that is to begin loading the applet (or the property may be the entire applet if it consists of a single class file). The actual Java class filename is not part of the `codeBase` property.

Related Items: `code` property.

height
width

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `height` and `width` properties represent the `height` and `width` attributes of the `applet` element. Although these values should be set via attributes in the tag, these properties can adjust the size of the applet after the fact in IE5+.

Related Items: `hspace`, `vspace` properties.

hspace
vspace

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `hspace` and `vspace` properties represent the `hspace` and `vspace` attributes of the `applet` element, which control the number of pixels of transparent padding around the `applet` element on the page. Although these values should be set via attributes in the tag, these properties can adjust the size of the applet padding after the fact in IE5+.

Related Items: `height`, `width` properties.

name

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `name` property represents the `name` attribute, a holdover from the early implementations of the `applet` element before `id` attributes were used to identify elements. The value assigned to the `name` attribute is the name you can use to reference applets in all browsers that support accessing applets: `document.appletName`.

object

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `object` property represents the `object` attribute, which, according to the W3C HTML standard, points to the URL of a serialized (that is, "saved") version of the applet's current state.

Related Items: `code` property.

vspace

(See `hspace`)

width

(See `height`)

object Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`align`	(Object methods)	`oncellchange`
`alt`		`ondataavailable`
`altHTML`		`ondatasetchanged`
`archive*`		`ondatasetcomplete`
`BaseHref`		`onload`
`baseURI`		`onrowenter`
`border`		`onrowexit`
`classid`		`onrowsdelete`
`code`		`onrowsinserted`
`codeBase`		`onscroll`
`codeType`		
`contentDocument`		
`data`		
`declare*`		
`form`		

Continued

Properties	Methods	Event Handlers
height		
hspace		
name		
object		
standby*		
type		
useMap		
vspace		
width		
(Object variables)		

*See text.

Syntax

Accessing `object` element object properties or methods:

```
(IE4+)         [window.]document.all.objectID.property | method([parameters])
(IE5+/W3C)     [window.]document.getElementById("objectID").property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

About this object

The `object` element is intended to be the primary way to add external content (that is, content that the browser itself does not render) to a page. For example, WinIE uses it to load ActiveX controls (whether from the server or locally). The `object` element is also destined to replace usage of the `applet` and `embed` elements.

As with the `applet` element object, scripts can frequently control the programs and plug-ins that get loaded into the browser through the `object` tag. Chapter 44 shows you how to do that for common objects. The property listings here are merely for the properties of the element, most of which mimic the attributes available for the `object` element. Even though the properties are exposed, they are very rarely scripted, except perhaps to adjust the size of the space occupied by a media controller. Most properties are read-only after their values are set by attributes in the element's tag. But if your scripts are creating the `object` element anew, scripts can set the property values the first time to initialize the object.

In the list of properties that begins this object's coverage, several are marked with an asterisk (*). These properties are defined in the W3C DOM Level 2 specification, and placeholders are included in version 6 browsers (and NN7/Mozilla1.4). But as of this writing, there is no indication that these properties are "connected."

A large set of event handlers for this element (all but `onload` and `onscroll`) is related to the application of WinIE data binding for `param` elements nested inside an `object` element. These events fire when a variety of actions occur to the data source or recordset associated with the program associated with the element. For more about applying data binding to an `object` element, see `http://msdn.microsoft.com/workshop/author/databind/dataconsumer.asp`.

Properties

align

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari1+

The `align` property controls either the horizontal or vertical alignment of the element with regard to surrounding content. String values of `left` or `right` cause the object rectangle to cling to the left or right edges of its next outermost positioning context. String values of `absbottom`, `absmiddle`, `baseline`, `bottom`, `middle`, `texttop`, or `top` influence the vertical alignment with respect to adjacent text, with the same kind of results as corresponding values of the `style.verticalAlign` property.

Related Items: `style.verticalAlign` property.

alt

Value: String. Read/Write
Compatibility: WinIE6+, MacIE-, NN-, Moz-, Safari-

The `alt` property represents the `alt` attribute, which should contain text that displays in the browser in the event that the object or its data do not load. This information should be set as the `object` element's attribute, because assigning text to the property after the object attempts to load does not insert the text into the page.

Related Items: `altHTML` property.

altHTML

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `altHTML` property is supposed to provide an `object` element with HTML content to render if the object doesn't load. In practice, assigning an HTML string to this property has no effect on an `object` element.

Related Items: `alt` property.

BaseHref

Value: String. Read-Only
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The `BaseHref` property returns the full URL path to the current document.

Related Items: `baseURI` property.

baseURI

Value: String. Read-Only
Compatibility: WinIE-, MacIE-, NN7+, Moz1+, Safari-

The `baseURI` property returns the full URL (URI) path to the current document.

Related Items: `BaseHref` property.

border

Value: Number as string. Read/Write
Compatibility: WinIE6+, MacIE-, NN6+, Moz1+, Safari1+

The `border` property controls the thickness of the border around the object, in pixels. You must specify the border thickness as a string, as in `"6"` for 6 pixels.

classid

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `classid` property represents the `classid` attribute of the `object` element. IE for Windows uses this attribute to assign the Globally Unique ID (GUID) of an ActiveX control. For example, to load a (nearly) invisible Windows Media Player object into a page, the HTML is as follows:

```
<object id="medPlayer" width="1" height="1"
   classid="CLSID:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
   codebase="#Version=1,0,0,0">
```

If your script then accesses the `classid` property of the `medPlayer` object, the value returned is the complete string as assigned to the attribute:

```
CLSID:22d6f312-b0f6-11d0-94ab-0080c74c7e95
```

Note that the `CLSID:` prefix is also part of the string value. Even if the object does not load (for example, because the object is missing or an error is in the long `classid` string), the property value reports the value as assigned to the attribute.

The HTML 4.0 specification indicates that the `classid` attribute be used for any kind of external class files, including Java applets. But in practice, IE wants applet URLs supplied to the `code` attribute (a non-HTML 4.0 attribute).

Related Items: `code` property.

code

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `code` property is the URL string of a Java class file that is to begin loading the applet (or the property may be the entire applet if it consists of a single class file). You cannot change the code assigned to an applet after the element has loaded (even if the applet code did not load successfully).

Related Items: `codeBase` property.

codeBase

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `codeBase` property is the string of the path on the server to the source of the applet or ActiveX control referenced by the `classid` or `code` attributes. IE4+ also uses the `codebase` attribute to specify a minimum version of control that is to load, if the attribute is available. This facet is discussed in Chapter 38's coverage of plug-in detection for WinIE.

Related Items: `code` property.

codeType

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `codeType` property is a string of the `mime` type of whatever object is pointed to by the `code` attribute value.

Related Items: `type` property.

contentDocument

Value: Document node reference. Read-Only
Compatibility: WinIE-, MacIE-, NN6+, Moz1+, Safari-

The `contentDocument` property returns the document node created by the object, if it exists.

data

Value: URL as string. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `data` property provides access to the URL of a file containing data for the object, as opposed to the object itself.

form

Value: Object reference. Read-Only
Compatibility: WinIE4+, MacIE5, NN6+, Moz1+, Safari1+

The `form` property returns a reference to the `form` element that contains the object, if there is one. This property only applies if the object is acting as a control within a form.

height
width

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `height` and `width` properties represent the `height` and `width` attributes of the `object` element. Although these values should be set via attributes in the tag, these properties can adjust the size of the embedded element after the fact in IE5+.

Related Items: `hspace`, `vspace` properties.

hspace
vspace

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `hspace` and `vspace` properties represent the `hspace` and `vspace` attributes of the `object` element, which control the number of pixels of transparent padding around the `object` element on the page. Although these values should be set via attributes in the tag, these properties can adjust the size of the padding around the element after the fact in IE5+.

Related Items: `height`, `width` properties.

name

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `name` property represents the `name` attribute of the `object` element. The better form is to assign an `id` to the `object` element and use accepted reference syntax for element IDs.

object

Value: External object. Read-Only
Compatibility: WinIE4+, MacIE4+, NN-, Moz-, Safari-

The `object` property returns a reference to the object contained by the `object` element. This property is essential if the program running inside the `object` element has the same property or method names as the `object` element itself. For example, consider a Java applet loaded into the `object` element as follows:

```
<object code="coolApplet" id="myAPPLET" ... >
```

If the applet code contained a public variable called `height`, an attempt to read or write that property through the `object` element will cause the element's properties to be read, and not the applet's properties. Therefore, if you insert the `object` property in the reference, the script reaches into the `applet` object for the property:

```
document.getElementById("myAPPLET").object.height = 40;
```

If there is no ambiguity between element and object property and method names, the browser looks first at the element and then the object to find a match.

type

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari1+

The `type` property represents the `type` attribute of the `object` element, which is intended to warn the browser about the `mime` type of data that is to be loaded into the object's process.

Related Items: `codeType` property.

useMap

Value: String. Read/Write
Compatibility: WinIE6+, MacIE-, NN6+, Moz1+, Safari1+

The `useMap` property identifies the URL of an image map; the image map is described by the `map` element in the same document. The value of the `useMap` property includes a hash mark followed by the name of the image map, as specified in the `usemap` attribute of the `object` element.

vspace

(See `hspace`)

width

(See `height`)

embed Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`align`	(Object methods)	`onload`
`height`		`onscroll`
`hidden`		
`name`		
`pluginspage`		
`src`		
`units`		
`width`		
(Object variables)		

Syntax

Accessing `embed` element object properties or methods:

```
(IE4+)         [window.]document.all.objectID.property | method([parameters])
(IE5+/W3C)     [window.]document.getElementById("objectID").property |
method([parameters])
```

Compatibility: WinIE4+, MacIE4+, NN3+, Moz1+, Safari-

About this object

An `embed` element is a carryover from the early browser days. Although never adopted by the W3C HTML standard, the `embed` element has been used in NN and IE as a way to embed non-native content (for example, sounds, video clips, and custom `mime` types for plug-ins, such as Shockwave) into a page. What gets embedded into the page is the controller or viewer for whatever kind of data the `embed` element points to (via the `src` attribute).

The `embed` element is far less sophisticated than the `object` element, but current browsers continue to support it. If you have been using the `embed` element in previous applications, it may be a good idea to start gravitating toward the `object` element. For backward-compatibility purposes, nesting an `embed` element inside an `object` element is not uncommon, both of which attempt to load the same content and plug-in. Browsers that know about the `object` element will load the content that way; older browsers will use the `embed` element and its attributes and parameters.

Because an `embed` element loads a plug-in (including ActiveX control types of plug-ins in WinIE), you can reference the plug-in's properties and methods through the `embed` object's reference.

Properties

align

Value: String. Read/Write
Compatibility: WinIE-, MacIE5+, NN6+, Moz1+, Safari-

The `align` property controls either the horizontal or vertical alignment of the element with regard to surrounding content. String values of `left` or `right` cause the object rectangle to cling to the left or right edges of its next outermost positioning context. String values of `absbottom`, `absmiddle`, `baseline`, `bottom`, `middle`, `texttop`, or `top` influence the vertical alignment with respect to adjacent text, with the same kind of results as corresponding values of the `style.verticalAlign` property.

Related Items: `style.verticalAlign` property.

height
width

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari-

The `height` and `width` properties represent the `height` and `width` attributes of the `embed` element. Although these values should be set via attributes in the tag, these properties can adjust the size of the element after the fact in IE5+.

hidden

Value: Boolean. Read/Write
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The `hidden` property represents the `hidden` attribute of the `embed` element. When an `embed` element is hidden, neither controller nor the content is shown. Application of this element in modern browsers should use stylesheets to hide and show the element.

Related Items: `style.visibility` property.

name

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN6+, Moz1+, Safari-

The `name` property represents the `name` attribute of the `embed` element. The better form is to assign an `id` to the `embed` element and use accepted reference syntax for element IDs.

pluginspage

Value: String. Read-Only
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The `pluginspage` property represents the `pluginspage` attribute of the `embed` element. This attribute is a URL that gets applied to a link in the browser if the plug-in associated with the external file's `mime` type cannot be found on the client.

src

Value: String. Read/Write
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The src property represents the src attribute of the embed element. This attribute points to the external file that is to be loaded into the browser via the associated plug-in. Scripts can assign a new URL string to this property to load a different file into the current plug-in.

units

Value: String. Read-Only
Compatibility: WinIE4+, MacIE-, NN-, Moz-, Safari-

The units property returns the unit of measure assigned with the length value of the height and width properties, in pixels.

Related Items: height, width properties.

The Odd Case of the param Element

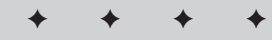
HTML pages pass parameters to Java applets, plug-ins, and ActiveX controls by way of param elements that are nested inside applet, embed, and object elements. Although a param element object is defined by the W3C DOM Level 2 specification, it does not show up on some browsers' radar when you try to reference the param element by itself. Even assigning an id to a param element or using document.getElementsByTagName("param") fail to allow references to access an individual param element object. At most, you can retrieve the innerHTML property of the surrounding element. But even here, the values returned may not necessarily be precisely the HTML you specify in the document.

In practice, this limitation is not particularly important. For one thing, even if you could access the param elements of an embedded object or program, attempts to modify the values would be wasted: Those values are read at load time only. Secondly, a well-designed plug-in, applet, or ActiveX control will provide its own properties or methods to retrieve the current settings of whatever properties are initialized via the param elements.

✦ ✦ ✦

CHAPTER 41

XML Objects



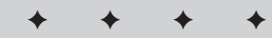
In This Chapter

Treating XML elements as objects

Creating IE XML data islands

Accessing XML element attributes

Using the `XMLHttpRequest` object



XML (eXtensible Markup Language) is an undeniably hot topic in the Internet world, and has been for the past few years. Not only has the W3C organization formed multiple working groups and recommendations for XML and its offshoots, but the W3C DOM recommendation also has XML in mind when it comes to defining how elements, attributes, and data of any kind — not just the HTML vocabulary — are exposed to browsers as an object model. Most of the arcana of the W3C DOM Core specification — especially the structure based on the node — are in direct response to the XML possibilities of documents that are beginning to travel the Internet.

During its early explorations into XML and browsers, Microsoft devised a custom HTML element — the `<xml>` tag — that allowed authors to embed XML data into an HTML document. These tags created what were called XML *data islands*. A more practical solution came slightly later with the creation of an ActiveX control that could retrieve XML data (from either a static `.xml` file or a Web service that returns XML-structured data) into a Web page without disturbing the HTML portion. Scripts could then use W3C DOM methods and properties to read the node tree as needed. Mozilla browsers emulate the behavior of this `XMLHttpRequest` control so that both WinIE and Mozilla-based browsers (all OSes) can load external XML data into a page for script inspection and manipulation.

This chapter covers both WinIE XML data islands and the more flexible `XMLHttpRequest` object. Out of necessity, this book assumes that you are already familiar with XML such that your server-based applications serve up XML data exclusively, embed XML islands into HTML documents, or convert database data into XML. The focus of this chapter, and application examples of Chapters 52 and 57, is how to access XML data and apply that data to rendered HTML content.

Elements and Nodes

Once you leave the specialized DOM vocabulary of HTML elements, the world can appear rather primitive — a highly granular world of node hierarchies, elements, element attributes, and node data. This granularity is a necessity in an environment in which the elements are far from generic and the structure of data in a document does not have to follow a format handed down from above. One Web application can describe an individual's contact information with one set of elements, while another application uses a completely different approach to element names, element nesting, and their sequence.

Fortunately, most, if not all, scripting you do on XML data is on data served up by your own applications. Therefore, you know what the structure of the data is — or you know enough of it to let your scripts access the data.

The discussion of the W3C DOM in Chapter 14 should serve as a good introduction to the way you need to think about elements and their content. All relevant properties and methods are listed among the items shared by all elements in Chapter 15.

XML data, whether delivered "raw" or embedded in a WinIE HTML document as a *data island* is a hierarchy of nodes. Typically, the outermost nodes are elements. Some elements have attributes, each of which is a typical name/value pair. Some elements have data that goes between the start and end tags of the element (such data is a text node nested inside the element node). And some elements can have both attributes and data. When an XML data collection contains the equivalent of multiple database records, an element container whose tag name is the same as each of the other records surrounds each record. Thus, the `getElementsByTagName()` method frequently accesses a collection of like-named elements.

Once you have a reference to an element node, you can reference that element's attributes as properties; however, a more formal access route is via the `getAttribute()` method of the element. If the element has text data between its start and end tags, you can access that data from the element's reference by calling the `firstChild.data` property (although you may want to verify that the element has a child node of the text type before committing to retrieving the data).

Of course, your specific approach to `xml` elements and their data varies with what you intend to script with the data. For example, you may wish to do nothing more with scripting than enable a different stylesheet for the data based on a user choice. The XSL (eXtensible Stylesheet Language) standard is a kind of (non-JavaScript) scripting language for transforming raw `xml` data into a variety of presentations. But you can still use JavaScript to connect user-interface elements that control which of several stylesheets renders the data. Or, as demonstrated in Chapters 52 and 57, you may wish to use JavaScript for more explicit control over the data and its rendering, taking advantage of JavaScript sorting and data manipulation facilities along the way.

Table 41-1 is a summary of W3C DOM Core objects, properties, and methods that you are most likely to use in extracting data from `xml` elements. You can find details of all of these items in Chapter 15.

Table 41-1: Properties and Methods for XML Element Reading

Property or Method	*Description*
`Node.nodeValue`	Data of a text node
`Node.nodeType`	Which node type
`Node.parentNode`	Reference to parent node
`Node.childNodes`	Array of child nodes
`Node.firstChild`	First of all child nodes
`Node.lastChild`	Last of all child nodes
`Node.previousSibling`	Previous node at same level
`Node.nextSibling`	Next node at same level
`Element.parentNode`	Reference to parent node
`Element.childNodes`	Array of child nodes

Property or Method	Description
`Element.firstChild`	First of all child nodes
`Element.lastChild`	Last of all child nodes
`Element.previousSibling`	Previous node at same level
`Element.nextSibling`	Next node at same level
`Element.tagName`	Tag name
`Element.getAttribute(`*`name`*`)`	Retrieves attribute (Attr) object
`Element.getElementsByTagName(`*`name`*`)`	Array of nested, named elements
`Attr.name`	Name part of attribute object's name/value pair
`Attr.value`	Value part of attribute object's name/value pair

xml Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	Methods	Event Handlers
`src`		
`XMLDocument`		

Syntax

Accessing `xml` element object properties or methods:

```
(IE5+)     [window.]document.all.elementID.property | method([parameters])
```

Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

About this object

The `xml` element object is the primary container of an `xml` data island within an HTML page. If your scripts intend to traverse the node hierarchy within the element, or simply access properties of nested elements, you should assign an identifier to the `id` attribute of the XML element. For example, if the XML data contains results from a database query for music recordings that match some user-entered criteria, each returned record might be denoted as a `recording` element as follows:

```
<xml id="results">
   <searchresults>
      <recording>
         ...elements with details...
      </recording>
      <recording>
         ...elements with details...
      </recording>
```

```
      <recording>
         ...elements with details...
      </recording>
   </searchresults>
</xml>
```

Your script can now obtain an array of references to `recording` elements as follows:

```
var recs =
   document.getElementById("results").getElementsByTagName("recording");
```

Although it is also true that there is no known HTML element with the tag name `recording` (which enables you to use `document.getElementsByTagName("recording")`), the unpredictability of `xml` data element names is reason enough to limit the scope of the `getElementsByTagName()` method to the `xml` data island.

The W3C DOM Level 2 does not define an `xml` element object within the HTML section. You cannot simply embed an XML document inside an HTML document: The standards clearly indicate that a document can be one or the other, but not both. Although the NN6/7 DOM can recognize custom elements, the browser understandably gets confused when custom elements have tag names that already belong to the HTML DTD. Therefore, I do not recommend attempting to embed custom elements into an HTML document for NN6/7 unless it someday implements a mechanism similar to IE's XML data islands.

Properties

src

Value: String. Read/Write
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `src` property represents the `src` attribute of the `xml` element. The attribute points to the URL of an external `xml` document whose data is embedded within the current HTML document.

XMLDocument

Value: Object reference. Read-Only
Compatibility: WinIE5+, MacIE-, NN-, Moz-, Safari-

The `XMLDocument` property returns a reference to Microsoft's proprietary XML `document` object and the object model associated with it (the so-called XML DOM). A lot of this object model is patterned after the W3C DOM model, but access to these properties is via a rather roundabout way. For more details, visit

`http://msdn.microsoft.com/xml/reference/xmldom/start.asp`

XMLHttpRequest Object

Properties	*Methods*	*Event Handlers*
`readyState`	`abort()`	
`responseText`	`getAllResponseHeaders()`	

Properties	Methods	Event Handlers
responseXML	getResponseHeader()	
status	open()	
statusText	send()	

Syntax

Accessing XMLHttpRequest object properties or methods:

```
(IE5+/Moz)     XMLHttpRequestObjectRef.property | method([parameters])
```

Compatibility: WinIE5+, MacIE-, NN7+, Moz1+, Safari-

About this object

The XMLHttpRequest object is an abstract object that lets your scripts retrieve XML data from, or send XML data to, any URL designed for that purpose. All of the action occurs invisibly to the user, and it is the responsibility of your scripts to make the connection with the server and process the XML data either after retrieval or prior to submission. This object was originally designed by Microsoft as part of its XML Core Services (MSXML), as first released as part of Internet Explorer 5 for Windows. Mozilla engineers have implemented much of the same functionality in Mozilla browsers, with almost identical syntax.

Where the IE and Mozilla variations differ is how you create the object to begin. Because the IE version is an ActiveX control, you create the object using the ActiveXObject constructor function. For Mozilla, the XMLHttpRequest object has its own constructor. Therefore, to equalize the creation of the two versions in a single document, you need to branch your code accordingly. Use object detection to handle the branching most effectively:

```
var req;
if (window.XMLHttpRequest) {
   req = new XMLHttpRequest();
} else if (window.ActiveXObject) {
   req = new ActiveXObject("Microsoft.XMLHTTP");
}
```

After the object is created, the basic syntax for opening a connection, sending the request, and retrieving the response data is the same for both WinIE and Mozilla. To retrieve an XML document (node tree) from a URL source, the basic sequence is as follows:

```
req.open("GET", "sourceURL", false);
req.send(null);
var xmlDocument = req.responseXML;
```

At this point, scripts can inspect the contents of the xmlDocument value by way of W3C DOM node properties and methods.

The XMLHttpRequest object in Mozilla-based browsers must reference pages served from a Web server, and not a local file. You can experiment successfully from a "personal Web server" running on your PC, but not with files accessed through the file: protocol.

Listing 41-1 shows a utility script that retrieves XML content from a URL (passed as a parameter to the loadXML() function) in a cross-browser manner. Additional error checking verifies

that the retrieval is successful, or reports the error to the user. If the browser running the script does not support the XML reading feature, the user is alerted of that, as well. The XML document object is preserved in a global variable called `xDoc`.

Listing 41-1: **Utility XML Data Reading Script**

```
// XML document
var xDoc;

// retrieve XML document as document object
function loadXMLDoc(url) {
   var req;
   if (window.XMLHttpRequest) {
      req = new XMLHttpRequest();
      req.open("GET", url, false);
      req.send(null);
   } else if (window.ActiveXObject) {
      req = new ActiveXObject("Microsoft.XMLHTTP");
      if (req) {
         req.open("GET", url, false);
         req.send();
      }
   }
   if (req) {
      if (req.status == 200) {
         xDoc = req.responseXML;
         if (xDoc && typeof xDoc.childNodes != "undefined" &&
            xDoc.childNodes.length == 0) {
            xDoc = null;
         }
      } else {
         alert("There was a problem retrieving the XML data:\n" +
         req.statusText);
      }
   } else {
      alert("Sorry, this browser isn\'t equipped to read XML data.");
   }
}
```

Properties and methods described in this chapter are those that the object has in common for both WinIE and Mozilla browsers. The WinIE object has some additional features, which you can study at Microsoft's MSXML 4.0 developer pages at `http://msdn.microsoft.com/library/en-us/xmlsdk/htm/dom_reference_2kdh.asp`. You can see examples of this object within the applications of Chapters 52 and 57.

Properties

`readyState`

Value: Integer. Read-Only
Compatibility: WinIE5+, MacIE-, NN7+, Moz1+, Safari-

Your scripts can read the value of the `readyState` property to determine the state of the `XMLHttpRequest` object, particularly while it is operating during its initialization or data

transfer. Values are the same as for other objects that offer this property. See Table 15-6 for integer values and their meanings. If you are scripting exclusively for WinIE, you can assign an `onreadystatechange` event handler to the `XMLHttpRequest` object; the event function can then inspect the `readyState` property for further processing.

Related Items: `status` property.

responseText

Value: String. Read-Only
Compatibility: WinIE5+, MacIE-, NN7+, Moz1+, Safari-

After the `send()` method executes, and if the server returns any data (as it will with a `GET` operation), you can access a string version of the returned data via the `responseText` property. If the returned data is an XML document, this property provides a string-only version of the entire content.

Related Items: `responseXML` property.

responseXML

Value: XML document object. Read-Only
Compatibility: WinIE5+, MacIE-, NN7+, Moz1+, Safari-

After the `send()` method executes, and if the server returns any data (as it will with a `GET` operation), you can access the returned W3C DOM-compliant document object via the `responseXML` property. The object to which this property points is a genuine document node (`nodeType` of 9), which gives your scripts the power to walk the node tree, and retrieve tags, attributes, and text nodes inside elements, as you would with any DOM document.

As the examples in Chapters 52 and 57 demonstrate, you can use the data from the XML document to build HTML that displays the XML content in the format of your choice (using JavaScript as a more flexible alternative to XSL). If your page is interactive to the extent that users can modify the content, you may then modify the document tree stored in your script variable and send the revised XML back to the server by opening a new `XMLHttpRequest` connection pointing to the URL that accepts the posted data.

Related Items: `responseText` property; `open()` method.

status

Value: Integer. Read-Only
Compatibility: WinIE5+, MacIE-, NN7+, Moz1+, Safari-

After the `send()` method executes, you can read the status of the transaction via the `status` property. The value is an integer corresponding to the response issued by the server at the end of the transaction. A successful transaction value is 200 (corresponding to the OK `statusText` property value). Perhaps the other most common status value is 404, which occurs if the URL you supply to the `open()` method points to a file or source not found on the server. As shown in Listing 41-1, you can use the 200 value as the key to determining if the transaction is a success, and then simply report any other value to the user (although inexperienced users may not understand the meaning of the status text). A complete list of status values can be found at `http://msdn.microsoft.com/library/en-us/xmlsdk/htm/xml_pro_sz_0h4k.asp`.

Related Items: `statusText` property.

statusText

Value: String. Read-Only
Compatibility: WinIE5+, MacIE-, NN7+, Moz1+, Safari-

After the `send()` method executes, you can read the plain-language status of the transaction via the `statusText` property. The value is a string corresponding to the response integer by the server at the end of the transaction. A successful transaction value is OK (corresponding to the 200 `status` property value). Use the `status` property for testing the results in your script, and the `statusText` property to report errors to users. See Listing 41-1.

Related Items: `status` property.

Methods

abort()

Returns: Nothing.
Compatibility: WinIE5+, MacIE-, NN7+, Moz1+, Safari-

The `abort()` method stops any transaction currently in progress. This method is the scripted equivalent of clicking a browser's Stop button while it retrieves contents of a Web page.

Related Items: `readyState` property; `send()` method.

getAllResponseHeaders()
getResponseHeader("*headerName*")

Returns: String.
Compatibility: WinIE5+, MacIE-, NN7+, Moz1+, Safari-

For each transaction, the server transmits a series of name/value pairs as a header to the actual data. The `getAllResponseHeaders()` method returns the complete set as received by the `XMLHttpRequest` object. Such a header set may look like the following:

```
Date: Mon, 06 Oct 2003 03:12:59 GMT
Server: Apache/1.3.27 (Darwin)
Last-Modified: Sun, 05 Oct 2003 22:13:04 GMT
Etag: "12babe-3a2-3f809770"
Accept-Ranges: bytes
Content-Length: 930
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/xml
```

If you want to retrieve the value of just one of the headers, use the `getResponseHeader()` method and pass as a parameter a string with only the name portion of one of the headers. For example:

```
var size = req.getResponseHeader("Content-Length");
```

The parameter is not case-sensitive, but the spelling (along with any hyphen in the name) is critical.

Related Items: `readyState` property; `send()` method.

open("*method*", "URL"[, *asyncFlag*[, "*userName*"[, "*password*"]]])

Returns: Nothing.
Compatibility: WinIE5+, MacIE-, NN7+, Moz1+, Safari-

Use the `open()` method to specify the transaction type and URL of the destination of the request. The *method* parameter may be either `GET` (for retrieving data from a server) or `POST` (for sending XML to a server). The *URL* may be either relative to the current page, or a complete `http:` URL.

Three additional parameters are optional. The first is a Boolean value for whether the request should be asynchronous. If `true` (the default), the `XMLHttpRequest` object does not wait for a response (after the `send()` method) before continuing with script processing. By setting this parameter to `false`, you ensure that processing continues only after the transaction has completed or timed out. If you'd rather not wait for the transaction to complete, but instead have some action occur when the returned data has arrived, you can assign an `onload` event handler to the `XMLHttpRequest` object, and let that event handler process the incoming data:

```
var req;
if (window.XMLHttpRequest) {
   req = new XMLHttpRequest();
} else if (window.ActiveXObject) {
   req = new ActiveXObject("Microsoft.XMLHTTP");
}
req.onload = processXML;  // event handler function reference
req.open("GET", "sourceURL", false);
req.send(null);
```

The other optional parameters are strings for a username and password if one is needed to access the URL.

Note that the `open()` method merely fills various properties of the request, and that the request does not occur until the `send()` method is invoked.

Related Items: `open()` method.

send(*content*)

Returns: Nothing.
Compatibility: WinIE5+, MacIE-, NN7+, Moz1+, Safari-

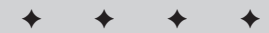
After setting the characteristics of the request via the `open()` method and its parameters, invoke the `send()` method to trigger the actual request over the network. For a GET operation, specify `null` as the parameter. But for a POST operation, the parameter should be a reference to a DOM document that has been assembled in script. You may also specify a string as the value being posted to the request's URL.

Related Items: `open()` method.

✦ ✦ ✦

CHAPTER 42

The Regular Expression and RegExp Objects

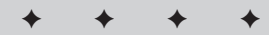


In This Chapter

What regular expressions are

How to use regular expressions for text search-and-replace

How to apply regular expressions to string object methods



Web programmers who have worked in Perl (and other Web application programming languages) know the power of regular expressions for processing incoming data and formatting data for readability in an HTML page or for accurate storage in a server database. Any task that requires extensive search and replacement of text can greatly benefit from the flexibility and conciseness of regular expressions. Version 4 browsers brought that power to JavaScript, although it is more fully fleshed out in IE5.5+.

Most of the benefit of JavaScript regular expressions accrues to those who script their CGI programs on servers that support a JavaScript version that contains regular expressions. But that's not to exclude the client-side from application of this "language within a language." If your scripts perform client-side data validations or any other extensive text entry parsing, consider using regular expressions rather than cobbling together comparatively complex JavaScript functions to perform the same tasks.

Regular Expressions and Patterns

In several chapters earlier in this book, I describe expressions as any sequence of identifiers, keywords, and/or operators that evaluate to some value. A regular expression follows that description, but has much more power behind it. In essence, a regular expression uses a sequence of characters and symbols to define a pattern of text. Such a pattern is used to locate a chunk of text in a string by matching up the pattern against the characters in the string.

An experienced JavaScript writer may point out the availability of the *string*`.indexOf()` and *string*`.lastIndexOf()` methods that can instantly reveal whether a string contains a substring and even where in the string that substring begins. These methods work perfectly well when the match is exact, character for character. But if you want to do more sophisticated matching (for example, does the string contain a five-digit ZIP code?), you'd have to cast aside those handy string methods and write some parsing functions. That's the beauty of a regular expression: It lets you define a matching substring that has some intelligence about it and can follow guidelines you set as to what should or should not match.

The simplest kind of regular expression pattern is the same kind you use in the *string*.indexOf() method. Such a pattern is nothing more than the text that you want to match. In JavaScript, one way to create a regular expression is to surround the expression by forward slashes. For example, consider the string

```
Oh, hello, do you want to play Othello in the school play?
```

This string and others may be examined by a script whose job it is to turn formal terms into informal ones. Therefore, one of its tasks is to replace the word "hello" with "hi." A typical brute force search-and-replace function starts with a simple pattern of the search string. In JavaScript, you define a pattern (a regular expression) by surrounding it with forward slashes. For convenience and readability, I usually assign the regular expression to a variable, as in the following example:

```
var myRegExpression = /hello/;
```

In concert with some regular expression or string object methods, this pattern matches the string "hello" wherever that series of letters appears. The problem is that this simple pattern causes problems during the loop that searches and replaces the strings in the example string: It finds not only the standalone word "hello," but also the "hello" in "Othello."

Trying to write another brute force routine for this search-and-replace operation that looks only for standalone words would be a nightmare. You can't merely extend the simple pattern to include spaces on either or both sides of "hello," because there could be punctuation — a comma, a dash, a colon, or whatever — before or after the letters. Fortunately, regular expressions provide a shortcut way to specify general characteristics, including a feature known as a word boundary. The symbol for a word boundary is `\b` (backslash, lowercase b). If you redefine the pattern to include these specifications on both ends of the text to match, the regular expression creation statement looks like this:

```
var myRegExpression = /\bhello\b/;
```

When JavaScript uses this regular expression as a parameter in a special string object method that performs search-and-replace operations, it changes only the standalone word "hello" to "hi," and passes over "Othello" entirely.

If you are still learning JavaScript and don't have experience with regular expressions in other languages, you have a price to pay for this power: Learning the regular expression lingo filled with so many symbols means that expressions sometimes look like cartoon substitutions for swear words. The goal of this chapter is to introduce you to regular expression syntax as implemented in JavaScript rather than engage in lengthy tutorials for this language. Of more importance in the long run is understanding how JavaScript treats regular expressions as objects and distinctions between instances of regular expression objects and the `RegExp` static object. I hope the examples in the following sections begin to reveal the powers of regular expressions.

Language Basics

To cover the depth of the regular expression syntax, I divide the subject into three sections. The first covers simple expressions (some of which you've already seen). Then I get into the wide range of special characters used to define specifications for search strings. Last comes an introduction to the usage of parentheses in the language, and how they not only help in grouping expressions for influencing calculation precedence (as they do for regular math expressions), but also how they temporarily store intermediate results of more complex expressions for use in reconstructing strings after their dissection by the regular expression.

Simple patterns

A simple regular expression uses no special characters for defining the string to be used in a search. Therefore, if you want to replace every space in a string with an underscore character, the simple pattern to match the space character is

```
var re = / /;
```

A space appears between the regular expression start-end forward slashes. The problem with this expression, however, is that it knows only how to find a single instance of a space in a long string. Regular expressions can be instructed to apply the matching string on a global basis by appending the `g` modifier:

```
var re = / /g;
```

When this `re` value is supplied as a parameter to the `replace()` method that uses regular expressions (described later in this chapter), the replacement is performed throughout the entire string, rather than just once on the first match found. Notice that the modifier appears after the final forward slash of the regular expression creation statement.

Regular expression matching — like a lot of other aspects of JavaScript — is case-sensitive. But you can override this behavior by using one other modifier that lets you specify a case-insensitive match. Therefore, the following expression

```
var re = /web/i;
```

finds a match for "web," "Web," or any combination of uppercase and lowercase letters in the word. You can combine the two modifiers together at the end of a regular expression. For example, the following expression is both case-insensitive and global in scope:

```
var re = /web/gi;
```

In compliance with the ECMA-262 Edition 3 standard, IE5.5+, NN6+/Moz, and Safari also allow a flag to force the regular expression to treat carriage-return-delimited lines of a multiline string as separate substrings with their own start and end boundaries. That modifier is the letter `m`.

Special characters

The regular expression in JavaScript borrows most of its vocabulary from the Perl regular expression. In a few instances, JavaScript offers alternatives to simplify the syntax, but also accepts the Perl version for those with experience in that arena.

Significant programming power comes from the way regular expressions allow you to include terse specifications about such facets as types of characters to accept in a match, how the characters are surrounded within a string, and how often a type of character can appear in the matching string. A series of escaped one-character commands (that is, letters preceded by the backslash) handle most of the character issues; punctuation and grouping symbols help define issues of frequency and range.

You saw an example earlier how `\b` specified a word boundary on one side of a search string. Table 42-1 lists the escaped character specifiers in JavaScript regular expressions. The vocabulary forms part of what are known as metacharacters — characters in expressions that are not matchable characters themselves, but act more as commands or guidelines of the regular expression language.

Table 42-1: JavaScript Regular Expression Matching Metacharacters

Character	*Matches*	*Example*
`\b`	Word boundary	`/\bor/` matches "origami" and "or" but not "normal" `/or\b/` matches "traitor" and "or" but not "perform" `/\bor\b/` matches full word "or" and nothing else
`\B`	Word non-boundary	`/\Bor/` matches "normal" but not "origami" `/or\B/` matches "normal" and "origami" but not "traitor" `/\Bor\B/` matches "normal" but not "origami" or "traitor"
`\d`	Numeral 0 through 9	`/\d\d\d/` matches "212" and "415" but not "B17"
`\D`	Non-numeral	`/\D\D\D/` matches "ABC" but not "212" or "B17"
`\s`	Single white space	`/over\sbite/` matches "over bite" but not "overbite" or "over bite"
`\S`	Single non-white space	`/over\Sbite/` matches "over-bite" but not "overbite" or "over bite"
`\w`	Letter, numeral, or underscore	`/A\w/` matches "A1" and "AA" but not "A+"
`\W`	Not letter, numeral, or underscore	`/A\W/` matches "A+" but not "A1" and "AA"
`.`	Any character except characters newline	`/.../` matches "ABC", "1+3", "A 3", or any three
`[...]`	Character set	`/[AN]BC/` matches "ABC" and "NBC" but not "BBC"
`[^...]`	Negated character set	`/[^AN]BC/` matches "BBC" and "CBC" but not "ABC" or "NBC"

Not to be confused with the metacharacters listed in Table 42-1 are the escaped string characters for tab (`\t`), newline (`\n`), carriage return (`\r`), formfeed (`\f`), and vertical tab (`\v`).

Let me further clarify about the `[...]` and `[^...]` metacharacters. You can specify either individual characters between the brackets (as shown in Table 42-1) or a contiguous range of characters or both. For example, the `\d` metacharacter can also be defined by `[0-9]`, meaning any numeral from zero through nine. If you only want to accept a value of 2 and a range from 6 through 8, the specification would be `[26-8]`. Similarly, the accommodating `\w` metacharacter is defined as `[A-Za-z0-9_]`, reminding you of the case-sensitivity of regular expression matches not otherwise modified.

All but the bracketed character set items listed in Table 42-1 apply to a single character in the regular expression. In most cases, however, you cannot predict how incoming data will be formatted — the length of a word or the number of digits in a number. A batch of extra metacharacters lets you set the frequency of the occurrence of either a specific character or a type of character (specified like the ones in Table 42-1). If you have experience in command-line operating systems, you can see some of the same ideas that apply to wildcards also apply to regular expressions. Table 42-2 lists the counting metacharacters in JavaScript regular expressions.

Table 42-2: JavaScript Regular Expression Counting Metacharacters

Character	*Matches Last Character*	*Example*
*	Zero or more times	`/Ja*vaScript/` matches "JvaScript", "JavaScript", and "JaaavaScript" but not "JovaScript"
?	Zero or one time	`/Ja?vaScript/` matches "JvaScript" or "JavaScript" but not "JaaavaScript"
+	One or more times	`/Ja+vaScript/` matches "JavaScript" or "JaavaScript" but not "JvaScript"
{*n*}	Exactly n times	`/Ja{2}vaScript/` matches "JaavaScript" but not "JvaScript" or "JavaScript"
{*n*,}	n or more times	`/Ja{2,}vaScript/` matches "JaavaScript" or "JaaavaScript" but not "JavaScript"
{*n*,*m*}	At least n, at most m times	`/Ja{2,3}vaScript/` matches "JaavaScript" or "JaaavaScript" but not "JavaScript"

Every metacharacter in Table 42-2 applies to the character immediately preceding it in the regular expression. Preceding characters may also be matching metacharacters from Table 42-1. For example, a match occurs for the following expression if the string contains two digits separated by one or more vowels:

```
/\d[aeiouy]+\d/
```

The last major contribution of metacharacters is helping the regular expression search a particular position in a string. By position, I don't mean something such as an offset — the matching functionality of regular expressions can tell me that. But, rather, whether the string to look for should be at the beginning or end of a larger string or (if the `m` modifier is added to the regular expression assignment) a line of that larger string. Table 42-3 shows the positional metacharacters for JavaScript's regular expressions.

Table 42-3: JavaScript Regular Expression Positional Metacharacters

Character	*Matches Located*	*Example*
^	At beginning of a string or line	`/^Fred/` matches "Fred is OK" but not "I'm with Fred" or "Is Fred here?"
$	At end of a string or line	`/Fred$/` matches "I'm with Fred" but not "Fred is OK" or "Is Fred here?"

For example, you may want to make sure that a match for a Roman numeral is found only when it is at the start of a string, rather than when it is used inline somewhere else. If the document contains Roman numerals in an outline, you can match all the top-level items that are flush left with the document with a regular expression, such as the following:

```
/^[IVXMDCL]+\./m
```

This expression matches any combination of Roman numeral characters at the start of a line followed by a period (the period is a special character in regular expressions, as shown in Table 42-1, so that you have to escape the period to offer it as a character), provided the Roman numeral is at the beginning of a line and has no tabs or spaces before it. There would also not be a match in a line that contains, for example, the phrase "see Part IV" because the Roman numeral is not at the beginning of a line.

Speaking of lines, a line of text is a contiguous string of characters delimited by a newline and/or carriage return (depending on the operating system platform). Word wrapping in `textarea` elements does not affect the starts and ends of true lines of text.

Grouping and backreferencing

Regular expressions obey most of the JavaScript operator precedence laws with regard to grouping by parentheses and the logical Or operator. One difference is that the regular expression Or operator is a single pipe character (`|`) rather than JavaScript's double pipe.

Parentheses have additional powers that go beyond influencing the precedence of calculation. Any set of parentheses (that is, a matched pair of left and right) stores the results of a found match of the expression within those parentheses. Parentheses can be nested inside one another. Storage is accomplished automatically, with the data stored in an indexed array accessible to your scripts and to your regular expressions (although through different syntax). Access to these storage bins is known as *backreferencing*, because a regular expression can point backward to the result of an expression component earlier in the overall expression. These stored subcomponents come in handy for replace operations, as demonstrated later in this chapter.

Object Relationships

JavaScript has a lot going on behind the scenes when you create a regular expression and perform the simplest operation with it. As important as the regular expression language described earlier in this chapter is to applying regular expressions in your scripts, the JavaScript object interrelationships are perhaps even more important if you want to exploit regular expressions to the fullest.

The first concept to master is that two entities are involved: a regular expression instance object and the `RegExp` static object. Both objects are core objects of JavaScript and are not part of the document object model. Both objects work together, but have entirely different sets of properties that may be useful to your application.

When you create a regular expression (even via the `/ . . . /` syntax), JavaScript invokes the `new RegExp()` constructor, much the way a `new Date()` constructor creates a date object around one specific date. The regular expression instance object returned by the constructor is endowed with several properties containing details of its data. At the same time, the single, static `RegExp` object maintains its own properties that monitor regular expression activity in the current window (or frame).

To help you see the typically unseen operations, I step you through the creation and application of a regular expression. In the process, I show you what happens to all of the related object properties when you use one of the regular expression methods to search for a match.

Several properties of both the regular expression instance object and the static `RegExp` object shown in the following "walk-through" are not available in IE until version 5.5. All are available in NN4+. See the individual property listings later in this chapter for compatibility ratings.

The starting text that I use to search through is the beginning of Hamlet's soliloquy (assigned to an arbitrary variable named `mainString`):

```
var mainString = "To be, or not to be: That is the question:";
```

If my ultimate goal is to locate each instance of the word "be," I must first create a regular expression that matches the word "be." I set the regular expression up to perform a global search when eventually called upon to replace itself (assigning the expression to an arbitrary variable named `re`):

```
var re = /\bbe\b/g;
```

To guarantee that only complete words "be" are matched, I surround the letters with the word boundary metacharacters. The final "g" is the global modifier. The variable to which the expression is assigned, `re`, represents a regular expression object whose properties and values are as follows:

Object.PropertyName	*Value*
re.source	"\bbe\bg"
re.global	true
re.ignoreCase	false
re.lastIndex	0

A regular expression's `source` property is the string consisting of the regular expression syntax (less the literal forward slashes). Each of the two possible modifiers, `g` and `i`, have their own properties, `global` and `ignoreCase`, whose values are Booleans indicating whether the modifiers are part of the source expression. The final property, `lastIndex`, indicates the index value within the main string at which the next search for a match should start. The default value for this property in a newly hatched regular expression is zero so that the search starts with the first character of the string. This property is read/write, so your scripts may want to adjust the value if they must have special control over the search process. As you see in a moment, JavaScript modifies this value over time if a global search is indicated for the object.

The `RegExp` constructor does more than just create regular expression objects. Like the `Math` object, the `RegExp` object is always "around" — one `RegExp` per window or frame — and tracks regular expression activity in a script. Its properties reveal what, if any, regular expression pattern matching has just taken place in the window. At this stage of the regular expression creation process, the `RegExp` object has only one of its properties set:

Object.PropertyName	*Value*
RegExp.input	
RegExp.multiline	false
RegExp.lastMatch	
RegExp.lastParen	
RegExp.leftContext	
RegExp.rightContext	
RegExp.$1	
...	
RegExp.$9	

The last group of properties (`$1` through `$9`) is for storage of backreferences. But because the regular expression I define above doesn't have any parentheses in it, these properties are empty for the duration of this examination and omitted from future listings in this "walk-through" section.

With the regular expression object ready to go, I invoke the `exec()` regular expression method, which looks through a string for a match defined by the regular expression. If the method is successful in finding a match, it returns a third object whose properties reveal a great deal about the item it found (I arbitrarily assign the variable `foundArray` to this returned object):

```
var foundArray = re.exec(mainString);
```

JavaScript includes a shortcut for the `exec()` method if you turn the regular expression object into a method:

```
var foundArray = re(mainString);
```

Normally, a script would check whether `foundArray` is `null` (meaning that there was no match) before proceeding to inspect the rest of the related objects. Because this is a controlled experiment, I know at least one match exists, so I first look into some other results. Running this simple method has not only generated the `foundArray` data, but also altered several properties of the `RegExp` and regular expression objects. The following shows you the current stage of the regular expression object:

Object.PropertyName	*Value*
`re.source`	`"\bbe\bg"`
`re.global`	`true`
`re.ignoreCase`	`false`
`re.lastIndex`	`5`

The only change is an important one: The `lastIndex` value has bumped up to `5`. In other words, this one invocation of the `exec()` method must have found a match whose offset plus length of matching string shifts the starting point of any successive searches with this regular expression to character index 5. That's exactly where the comma after the first "be" word is in the main string. If the global (`g`) modifier had not been appended to the regular expression, the `lastIndex` value would have remained at zero, because no subsequent search would be anticipated.

As the result of the `exec()` method, the `RegExp` object has had a number of its properties filled with results of the search:

Object.PropertyName	*Value*
`RegExp.input`	
`RegExp.multiline`	`false`
`RegExp.lastMatch`	`"be"`
`RegExp.lastParen`	
`RegExp.leftContext`	`"To "`
`RegExp.rightContext`	`", or not to be: That is the question:"`

From this object you can extract the string segment that was found to match the regular expression definition. The main string segments before and after the matching text are also available individually (in this example, the `leftContext` property has a space after "To"). Finally, looking into the array returned from the `exec()` method, some additional data is readily accessible:

Object.PropertyName	*Value*
`foundArray[0]`	`"be"`
`foundArray.index`	`3`
`foundArray.input`	`"To be, or not to be: That is the question:"`

The first element in the array, indexed as the zeroth element, is the string segment found to match the regular expression, which is the same as the `RegExp.lastMatch` value. The complete main string value is available as the input property. A potentially valuable piece of information to a script is the index for the start of the matched string found in the main string. From this last bit of data, you can extract from the found data array the same values as `RegExp.leftContext` (with `foundArray.input.substring(0, foundArray.index)`) and `RegExp. rightContext` (with `foundArray.input.substring(foundArray.index, foundArray[0].length)`).

Because the regular expression suggested a multiple execution sequence to fulfill the global flag, I can run the `exec()` method again without any change. While the JavaScript statement may not be any different, the search starts from the new `re.lastIndex` value. The effects of this second time through ripple through the resulting values of all three objects associated with this method:

```
var foundArray = re.exec(mainString);
```

Results of this execution are as follows (changes are in boldface).

Object.PropertyName	*Value*
`re.source`	`"\bbe\bg"`
`re.global`	`true`
`re.ignoreCase`	`false`
`re.lastIndex`	`19`
`RegExp.input`	
`RegExp.multiline`	`false`
`RegExp.lastMatch`	`"be"`
`RegExp.lastParen`	
`RegExp.leftContext`	`", or not to "`
`RegExp.rightContext`	`": That is the question:"`
`foundArray[0]`	`"be"`
`foundArray.index`	`17`
`foundArray.input`	`"To be, or not to be: That is the question:"`

Because there was a second match, `foundArray` comes back again with data. Its index property now points to the location of the second instance of the string matching the regular expression definition. The regular expression object's `lastIndex` value points to where the next search would begin (after the second "be"). And the `RegExp` properties that store the left and right contexts have adjusted accordingly.

If the regular expression were looking for something less stringent than a hard-coded word, some other properties may also be different. For example, if the regular expression defined a format for a ZIP code, the `RegExp.lastMatch` and `foundArray[0]` values would contain the actual found ZIP codes, which would likely be different from one match to the next.

Running the same `exec()` method once more does not find a third match in my original `mainString` value, but the impact of that lack of a match is worth noting. First of all, the `foundArray` value is `null` — a signal to our script that no more matches are available. The regular expression object's `lastIndex` property reverts to zero, ready to start its search from the beginning of another string. Most importantly, however, the `RegExp` object's properties maintain the same values from the last successful match. Therefore, if you put the `exec()` method invocations in a repeat loop that exits after no more matches are found, the `RegExp` object still has the data from the last successful match, ready for further processing by your scripts.

Using Regular Expressions

Despite the seemingly complex hidden workings of regular expressions, JavaScript provides a series of methods that make common tasks involving regular expressions quite simple to use (assuming you figure out the regular expression syntax to create good specifications). In this section, I present examples of syntax for specific kinds of tasks for which regular expressions can be beneficial in your pages.

Is there a match?

I said earlier that you can use `string.indexOf()` or `string.lastIndexOf()` to look for the presence of simple substrings within larger strings. But if you need the matching power of regular expression, you have two other methods to choose from:

```
regexObject.test(string)
string.search(regexObject)
```

The first is a regular expression object method, the second a string object method. Both perform the same task and influence the same related objects, but they return different values: a Boolean value for `test()` and a character offset value for `search()` (or `-1` if no match is found). Which method you choose depends on whether you need only a true/false verdict on a match or the location within the main string of the start of the substring.

Listing 42-1 demonstrates the `search()` method on a page that lets you get the Boolean and offset values for a match. Some default text and regular expression is provided (it looks for a five-digit number). You can experiment with other strings and regular expressions. Because this script creates a regular expression object with the new `RegExp()` constructor method, you do not include the literal forward slashes around the regular expression.

Listing 42-1: **Looking for a Match**

```
<html>
  <head>
     <title>Got a Match?</title>
     <script type="text/javascript">
     function findIt(form) {
        var re = new RegExp(form.regexp.value);
        var input = form.main.value;
        if (input.search(re) != -1) {
           form.output[0].checked = true;
        } else {
           form.output[1].checked = true;
        }
     }
     function locateIt(form) {
        var re = new RegExp(form.regexp.value);
        var input = form.main.value;
        form.offset.value = input.search(re);
     }
     </script>
  </head>
  <body>
     <b>Use a regular expression to test for the existence of a string:</b>
     <hr />
     <form>
        Enter some text to be searched:<br />
        <textarea name="main" cols="40" rows="4" wrap="virtual">
           The most famous ZIP code on Earth may be 90210.</textarea><br />
        Enter a regular expression to search:<br />
        <input type="text" name="regexp" size="30" value="\b\d\d\d\d\d\b" />
        <p><input type="button" value="Is There a Match?"
           onclick="findIt(this.form)" /> <input type="radio"
           name="output" />Yes <input type="radio" name="output" />No</p>
        <p><input type="button" value="Where is it?"
           onclick="locateIt(this.form)" /> <input type="text" name="offset"
           size="4" /></p>
        <p><input type="reset" /></p>
     </form>
  </body>
</html>
```

Getting information about a match

For the next application example, the task is not only to verify that a one-field date entry is in the desired format, but also to extract match components of the entry and use those values to perform further calculations in determining the day of the week. The regular expression in the example that follows is a fairly complex one, because it performs some rudimentary range checking to make sure the user doesn't enter a month over 12 or a date over 31. What it does

not take into account is the variety of lengths of each month. But the regular expression and method invoked with it extract each date object component in such a way that you can perform additional validation on the range to make sure the user doesn't try to give September 31 days. Also be aware that this is not the only way to perform date validations in forms. Chapter 43 offers additional thoughts on the matter that work without regular expressions for backward compatibility.

Listing 42-2 contains a page that has a field for date entry, a button to process the date, and an output field for display of a long version of the date, including the day of the week. At the start of the function that does all the work, I create two arrays to hold the plain language names of the months and days. These arrays are used only if the user enters a valid date.

Next comes the regular expression to be matched against the user entry. If you can decipher all the symbols, you see that three components are separated by potential hyphen or forward slash entries (`[\-\/]`). These symbols must be escaped in the regular expression. Importantly, each of the three component definitions is surrounded by parentheses, which are essential for the various objects created with the regular expression to remember their values for extraction later.

Here is a brief rundown of what the regular expression is looking for:

- A string beginning after a word break
- A string value for the month that contains a 1 plus a 0 through 2; or an optional 0 plus a 1 through 9
- A hyphen or forward slash
- A string value for the date that starts with a 0 plus a 1 through 9; or starts with a 1 or 2 and ends with a 0 through 9; or starts with a 3 and ends with 0 or 1
- Another hyphen or forward slash
- A string value for the year that begins with 19 or 20, followed by two digits

An extra pair of parentheses must surround the `19|20` segment to make sure that either one of the matching values is attached to the two succeeding digits. Without the parentheses, the logic of the expression attaches the digits only to 20.

For invoking the regular expression action, I select the `exec()` method, assigning the returned object to the variable `matchArray`. I can also use the *`string`*`.match()` method here. Only if the match is successful (that is, all conditions of the regular expression specification are met) does the major processing continue in the script.

The parentheses around the segments of the regular expression instruct JavaScript to assign each found value to a slot in the `matchArray` object. The month segment is assigned to `matchArray[1]`, the date to `matchArray[2]`, and the year to `matchArray[3]` (`matchArray[0]` contains the entire matched string). Therefore, the script can extract each component to build a plain-language date string with the help of the arrays defined at the start of the function. I even use the values to create a new date object that calculates the day of the week for me. After I have all pieces, I concatenate them and assign the result to the value of the output field. If the regular expression `exec()` method doesn't match the typed entry with the expression, the script provides an error message in the field.

Listing 42-2: **Extracting Data from a Match**

```
<html>
   <head>
      <title>Got a Match?</title>
      <script type="text/javascript">
      function extractIt(form) {
         var months = ["January","February","March","April","May","June",
            "July","August","September","October","November","December"];
         var days = ["Sunday","Monday","Tuesday","Wednesday","Thursday",
            "Friday","Saturday"];
         var re = /\b(1[0-2]|0?[1-9])[\-\/](0?[1-9]|[12][0-9]|3[01])[\-
\/]((19|20)\d{2})/;
         var input = form.entry.value;
         var matchArray = re.exec(input);
         if (matchArray) {
            var theMonth = months[matchArray[1] - 1] + " ";
            var theDate = matchArray[2] + ", ";
            var theYear = matchArray[3];
            var dateObj = new Date(matchArray[3],
               matchArray[1]-1,matchArray[2]);
            var theDay = days[dateObj.getDay()] + " ";
            form.output.value = theDay + theMonth + theDate + theYear;
         } else {
            form.output.value = "An invalid date.";
         }
      }
      </script>
   </head>
   <body>
      <b>Use a regular expression to extract data from a string:</b>
      <hr />
      <form>
         Enter a date in the format mm/dd/yyyy or mm-dd-yyyy:<br />
         <input type="text" name="entry" size="12" />
         <p><input type="button" value="Extract Date Components"
            onclick="extractIt(this.form)" /></p>
         <p>The date you entered was:<br />
            <input type="text" name="output" size="40" /></p>
         <p><input type="reset" /></p>
      </form>
   </body>
</html>
```

String replacement

To demonstrate using regular expressions for performing search-and-replace operations, I choose an application that may be of value to many page authors who have to display and format large numbers. Databases typically store large integers without commas. After five or six digits, however, such numbers are difficult for users to read. Conversely, if the user needs to enter a large number, commas help ensure accuracy.

Helping the procedure in JavaScript regular expressions is the *string*.replace() method (see Chapter 27). The method requires two parameters, a regular expression to search the string and a string to replace any match found in the string. The replacement string can be properties of the RegExp object as it stands after the most recent exec() method.

Listing 42-3 demonstrates how only a handful of script lines can do a lot of work when regular expressions handle the dirty work. The page contains three fields. Enter any number you want in the first one. A click of the Insert Commas button invokes the commafy() function in the page. The result is displayed in the second field. You can also enter a comma-filled number in the second field and click the Remove Commas button to see the inverse operation executed through the decommafy() function.

Specifications for the regular expression accept any positive or negative string of numbers. The keys to the action of this script are the parentheses around two segments of the regular expression. One set encompasses all characters not included in the second group: a required set of three digits. In other words, the regular expression is essentially working from the rear of the string, chomping off three-character segments and inserting a comma each time a set is found.

A while repeat loop cycles through the string and modifies the string (in truth, the string object is not being modified, but, rather, a new string is generated and assigned to the old variable name). I use the test() method because I don't need the returned value of the exec() method. The test() method impacts the regular expression and RegExp object properties the same way as the exec() method, but more efficiently. The first time the test() method runs, the part of the string that meets the first segment is assigned to the RegExp.$1 property; the second segment, if any, is assigned to the RegExp.$2 property. Notice that I'm not assigning the results of the exec() method to any variable, because for this application I don't need the array object generated by that method.

Next comes the tricky part. I invoke the *string*.replace() method, using the current value of the string (num) as the starting string. The pattern to search for is the regular expression defined at the head of the function. But the replacement string may look strange to you. The replacement string is replacing whatever the regular expression matches with the value of RegExp.$1, a comma, and the value of RegExp.$2. The RegExp object should not be part of the references used in the replace() method parameter. Because the regular expression matches the entire num string, the replace() method is essentially rebuilding the string from its components, plus adding a comma before the second component (the last freestanding three-digit section). Each replace() method invocation sets the value of num for the next time through the while loop and the test() method.

Looping continues until no matches occur — meaning that no more freestanding sets of three digits appear in the string. Then the results are written to the second field on the page.

Listing 42-3: **Replacing Strings via Regular Expressions**

```
<html>
  <head>
     <title>Got a Match?</title>
     <script type="text/javascript">
     function commafy(form) {
        var re = /(-?\d+)(\d{3})/;
        var num = form.entry.value;
        while (re.test(num)) {
           num = num.replace(re, "$1,$2");
        }
        form.commaOutput.value = num;
```

```
      }
      function decommafy(form) {
         var re = /,/g;
         form.plainOutput.value = form.commaOutput.value.replace(re,"");
      }
      </script>
   </head>
   <body>
      <b>Use a regular expression to add/delete commas from numbers:</b>
      <hr />
      <form>
         Enter a large number without any commas:<br />
         <input type="text" name="entry" size="15" />
         <p><input type="button" value="Insert commas"
            onclick="commafy(this.form)" /></p>
         <p>The comma version is:<br />
            <input type="text" name="commaOutput" size="20" /></p>
         <p><input type="button" value="Remove commas"
            onclick="decommafy(this.form)" /></p>
         <p>The un-comma version is:<br />
            <input type="text" name="plainOutput" size="15" /></p>
         <p><input type="reset" /></p>
      </form>
   </body>
</html>
```

Removing the commas is an even easier process. The regular expression is a comma with the global flag set. The `replace()` method reacts to the global flag by repeating the process until all matches are replaced. In this case, the replacement string is an empty string. For further examples of using regular expressions with string objects, see the discussions of the *`string`*`.match()`, *`string`*`.replace()`, and *`string`*`.split()` methods in Chapter 27.

Regular Expression Object

Properties	*Methods*	*Event Handlers*
`constructor`	`compile()`	
`global`	`exec()`	
`ignoreCase`	`test()`	
`lastIndex`		
`multilane`		
`source`		

Syntax

Accessing regular expression properties or methods:

`regularExpressionObject.property | method([parameters])`

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

About this object

The regular expression object is created on the fly by your scripts. Each regular expression object contains its own pattern and other properties. Deciding which object creation style to use depends on the way the regular expression will be used in your scripts.

When you create a regular expression with the literal notation (that is, with the two forward slashes), the expression is automatically compiled for efficient processing as the assignment statement executes. The same is true when you use the new `RegExp()` constructor and specify a pattern (and optional modifier flags) as a parameter. Whenever the regular expression is fixed in the script, use the literal notation; when some or all of the regular expression is derived from an external source (for example, user input from a text field), assemble the expression as a parameter to the new `RegExp()` constructor. A compiled regular expression should be used at whatever stage the expression is ready to be applied and reused within the script. Compiled regular expressions are not saved to disk or given any more permanence beyond the life of a document's script (that is, it dies when the page unloads).

However, there may be times in which the specification for the regular expression changes with each iteration through a loop construction. For example, if statements in a `while` loop modify the content of a regular expression, compile the expression inside the `while` loop, as shown in the following skeletal script fragment:

```
var srchText = form.search.value;
var re = new RegExp();  // empty constructor
while (someCondition) {
   re.compile("\\s+" + srchText + "\\s+", "gi");
   statements that change srchText
}
```

Each time through the loop, the regular expression object is both given a new expression (concatenated with metacharacters for one or more white spaces on both sides of some search text whose content changes constantly) and compiled into an efficient object for use with any associated methods.

Properties

`constructor`

(See `string.constructor` in Chapter 27)

`global`
`ignoreCase`

Value: Booleans. Read-Only
Compatibility: WinIE5.5+, MacIE5+, NN4+, Moz1+, Safari1+

These two properties reflect the regular expression `g` and `i` modifier flags, if any, associated with a regular expression. Settings are read-only and are determined as the object is created. Each property is independent of the other.

lastIndex

Value: Integer. Read/Write
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The `lastIndex` property indicates the index counter of the main string to be searched against the current regular expression object. When a regular expression object is created, this value is zero, meaning that there have been no searches with this object, and the default behavior of the first search is to start at the beginning of the string.

If the regular expression has the global modifier specified, the `lastIndex` property value advances to some higher value after the object is used in a method to match within a main string. The value is the position in the main string immediately after the previous matched string (and not including any character of the matched string). After locating the final match in a string, the method resets the `lastIndex` property to zero for the next time. You can also influence the behavior of matches by setting this value on the fly. For example, if you want the expression to begin its search at the fourth character of a target string, you change the setting immediately after creating the object, as follows:

```
var re = /somePattern/;
re.lastIndex = 3;  // fourth character in zero-based index system
```

Related Items: Match result object index property.

multiline

Value: Boolean. Read-Only
Compatibility: WinIE5.5+, MacIE5, NN6+, Moz1+, Safari1+

The `multiline` property reveals whether searches extend across multiple lines of a target string, as directed by the optional `m` modifier flag for a regular expression. NN4+ also includes the same-named property for the `RegExp` object (see the following section).

Related Items: `RegExp.multiline` property.

source

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The `source` property is simply the string representation of the regular expression used to define the object. This property is read-only.

Methods

compile("*pattern*", ["g" | "i" | "m"])

Returns: Regular expression object.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari-

Use the `compile()` method to compile on the fly a regular expression whose content changes continually during the execution of a script. See the discussion earlier about this object for an example. Other regular expression creation statements (the literal notation and the new `RegExp()` constructor that passes a regular expression) automatically compile their expressions. The `m` pattern modifier is available in IE5.5+ and NN6+/W3C.

exec("*string*")

Returns: Match array object or `null`.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The `exec()` method examines the string passed as its parameter for at least one match of the specification defined for the regular expression object. The behavior of this method is similar to that of the *`string`*`.match()` method (although the `match()` method is more powerful in completing global matches). Typically, a call to the `exec()` method is made immediately after the creation of a regular expression object, as in the following example:

```
var re = /somePattern/;
var matchArray = re.exec("someString");
```

Much happens as a result of the `exec()` method. Properties of both the regular expression object and window's `RegExp` object are updated based on the success of the match. The method also returns an object that conveys additional data about the operation. Table 42-4 shows the properties of this returned object.

Table 42-4: Match Found Array Object Properties

Property	*Description*
`index`	Zero-based index counter of the start of the match inside the string
`input`	Entire text of original string
`[0]`	String of most recent matched characters
`[1],...[n]`	Parenthesized component matches

Some of the properties in this returned object echo properties in the `RegExp` object. The value of having them in the regular expression object is that their contents are safely stowed in the object while the `RegExp` object and its properties may be modified soon by another call to a regular expression method. Items the two objects have in common are the `[0]` property (mapped to the `RegExp.lastMatch` property) and the `[1],. . .[n]` properties (the first nine of which map to `RegExp.$1. . .RegExp.$9`). Although the `RegExp` object stores only nine parenthesized subcomponents, the returned array object stores as many as are needed to accommodate parenthesis pairs in the regular expression.

If no match turns up between the regular expression specification and the string, the returned value is `null`. See Listing 42-2 for an example of how this method can be applied. An alternate shortcut syntax may be used for the `exec()` method. Turn the regular expression into a function, as in

```
var re = /somePattern/;
var matchArray = re("someString");
```

Related Items: `string.match()` method.

test("*string*")

Returns: Boolean.
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari1+

The most efficient way to find out if a regular expression has a match in a string is to use the `test()` method. Returned values are `true` if a match exists and `false` if not. In case you need more information, a companion method, *`string`*`.search()`, returns the starting index value of the matching string. See Listing 42-1 for an example of this method in action.

Related Items: `string.search()` method.

RegExp Object

Properties	*Methods*	*Event Handlers*
`input`	(None)	(None)
`lastMatch`		
`lastParen`		
`leftContext`		
`multilane`		
`prototype`		
`rightContext`		
`$1, ... $9`		

Syntax

Accessing `RegExp` properties:

`RegExp.`*`property`*

Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari-

About this object

Beginning with version 4 browsers, the browser maintains a single instance of a `RegExp` object for each window or frame. The object oversees the action of all methods that involve regular expressions (including the few related string object methods). Properties of this object are exposed not only to JavaScript in the traditional manner, but also to a parameter of the method *`string`*`.replace()` for some shortcut access (see Listing 42-3).

With one `RegExp` object serving all regular expression-related methods in your document's scripts, you must exercise care in accessing or modifying this object's properties. You must make sure that the `RegExp` object has not been affected by another method. Most properties are subject to change as the result of any method involving a regular expression. This may be reason enough to use the properties of the array object returned by most regular expression methods instead of the `RegExp` properties. The former stick with a specific regular expression object even after other regular expression objects are used in the same script. The `RegExp` properties reflect the most recent activity, irrespective of the regular expression object involved.

In the following listings, I supply the long, JavaScript-like property names. But each property also has an abbreviated, Perl-like manner to refer to the same properties. You can use these shortcut property names in the *`string`*`.replace()` method if you need the values.

Properties

input

Value: String. Read/Write
Compatibility: WinIE4+, MacIE4+, NN4+, Moz-, Safari-

The RegExp.input property is the main string against which a regular expression is compared in search of a match. In all of the example listings earlier in this chapter, the property was null. Such is the case when the main string is supplied as a parameter to the regular expression-related method.

But many text-related document objects have an unseen relationship with the RegExp object. If a text, textarea, select, or link object contains an event handler that invokes a function containing a regular expression, the RegExp.input property is set to the relevant textual data from the object. You don't have to specify any parameters for the event handler call or in the function called by the event handler. For text and textarea objects, the input property value becomes the content of the object; for the select object, it is the text (not the value) of the selected option; and for a link, it is the text highlighted in the browser associated with the link (and reflected in the link's text property).

Having JavaScript set the RegExp.input property for you may simplify your script. You can invoke either of the regular expression methods without having to specify the main string parameter. When that parameter is empty, JavaScript applies the RegExp.input property to the task. You can also set this property on the fly if you want. The short version of this property is $_ (dollar sign underscore).

Related Items: Matching array object input property.

multiline

Value: Boolean. Read/Write
Compatibility: WinIE5.5+, MacIE5+, NN4+, Moz1+, Safari-

The RegExp.multiline property determines whether searches extend across multiple lines of a target string. This property is automatically set to true as an event handler of a textarea triggers a function containing a regular expression. You can also set this property on the fly if you want. The short version of this property is $. This version of the property (as distinct from the multiline property of an instance of a regular expression) is not defined in the ECMA-262 specification and is based on an early, incorrect implementation.

Related Items: Regular expression instance object multiline property.

lastMatch

Value: String. Read-Only
Compatibility: WinIE5.5+, MacIE5+, NN4+, Moz1+, Safari-

After execution of a regular expression–related method, any text in the main string that matches the regular expression specification is automatically assigned to the RegExp.lastMatch property. This value is also assigned to the [0] property of the object array returned after the exec() and string.match() methods find a match. The short version of this property is $&.

Related Items: Matching array object [0] property.

lastParen

Value: String. Read-Only
Compatibility: WinIE5.5+, MacIE5+, NN4+, Moz1+, Safari-

When a regular expression contains many parenthesized subcomponents, the RegExp object maintains a list of the resulting strings in the $1, . . . $9 properties. You can also extract the value of the last matching parenthesized subcomponent with the RegExp.lastParen property, which is a read-only property. The short version of this property is $+.

Related Items: RegExp.$1, . . . $9 properties.

leftContext
rightContext

Value: String. Read-Only
Compatibility: WinIE5.5+, MacIE5+, NN4+, Moz1+, Safari-

After a match is found in the course of one of the regular expression methods, the RegExp object is informed of some key contextual information about the match. The leftContext property contains the part of the main string to the left of (up to but not including) the matched string. Be aware that the leftContext starts its string from the point at which the most recent search began. Therefore, for second or subsequent times through the same string with the same regular expression, the leftContext substring varies widely from the first time through.

The rightContext consists of a string starting immediately after the current match and extending to the end of the main string. As subsequent method calls work on the same string and regular expression, this value obviously shrinks in length until no more matches are found. At this point, both properties revert to null. The short versions of these properties are $` and $' for leftContext and rightContext, respectively.

prototype

(See String.prototype in Chapter 27)

$1...$9

Value: String. Read-Only
Compatibility: WinIE4+, MacIE4+, NN4+, Moz1+, Safari-

As a regular expression method executes, any parenthesized result is stored in RegExp's nine properties reserved for just that purpose (called backreferences). The same values (and any beyond the nine that RegExp has space for) are stored in the array object returned with the exec() and *string*.match() methods. Values are stored in the order in which the left parenthesis of a pair appears in the regular expression, regardless of nesting of other components.

You can use these backreferences directly in the second parameter of the *string*.replace() method, without using the RegExp part of their address. The ideal situation is to encapsulate components that need to be rearranged or recombined with replacement characters. For example, the following script function turns a name that is last name first into first name last:

```
function swapEm() {
   var re = /(\w+),\s*(\w+)/;
   var input = "Lincoln, Abraham";
   return input.replace(re,"$2 $1");
}
```

***RegExp*.$1**

In the `replace()` method, the second parenthesized component (just the first name) is placed first, followed by a space and the first component. The original comma is discarded. You are free to combine these shortcut references as you like, including multiple times per replacement, if it makes sense to your application.

Related Items: Matching array object `[1]. . .[n]` properties.

✦ ✦ ✦

CHAPTER 43

Data-Entry Validation

In This Chapter

Validating data as it is being entered

Validating data immediately prior to submission

Organizing complex data-validation tasks

Give users a field in which to enter data and you can be sure that some users will enter the wrong kind of data. Often the "mistake" is accidental — a slip of the pinkie on the keyboard; other times, users intentionally type the incorrect entry to test the robustness of your application. Whether you solicit a user's entry for client-side scripting purposes or for input into a server-based CGI or database, you should use JavaScript on the client to handle validation of the user's entry. Even for a form connected to a CGI script, it's far more efficient (from the perspective of bandwidth, server load, and execution speed) to let client-side JavaScript get the data straight before your server program deals with it.

Real-Time versus Batch Validation

You have two opportunities to perform data-entry validation in a form: as the user enters data into a form and just before the form is submitted. I recommend you take advantage of both of these opportunities.

Real-time validation triggers

The most convenient time to catch an error is immediately after the user makes it — especially for a long form that requests a wide variety of information. You can make the user's experience less frustrating if you catch an entry mistake just after the user enters the information: his or her attention is already focused on the nature of the content (or some paper source material may already be in front of the user). It is much easier for the user to address the same information entry right away.

A valid question for the page author is how to trigger the real-time validation. Backward-compatible text boxes have two potential event handlers for this purpose: `onchange` and `onblur`. I personally avoid `onblur` event handlers, especially ones that can display an alert dialog box (as a data-entry validation is likely to do). Because a good validation routine brings focus to the errant text box, you can get some odd behavior with the interaction of the `focus()` method and the `onblur` event handler. Users who wish to continue past an invalid field are locked in a seemingly endless loop.

The problem with using `onchange` as the validation trigger is that a user can defeat the validation. A change event occurs only when the text of a field indeed changes when the user tabs or clicks out of the field. If the user is alerted about some bad entry in a field and doesn't fix the error, the change event doesn't fire again. In some respects, this is good because a user may have a legitimate reason for passing by a particular form field initially with the intention of returning to the entry later. Because a user can defeat the `onchange` event handler trigger, I recommend you also perform batch validation prior to submission.

In NN4+, IE4+, and W3C browsers, you also have the luxury of letting keyboard events trigger validations. This is most helpful when you want to prevent some character(s) from being entered into a field. For example, if a field is supposed to contain only a positive integer value, you can use the `onkeypress` event handler of the text box to verify that the character just typed is a number. If the character is not a number, the event is trapped and no character reaches the text box. You should also alert the user in some way about what's going on. Listing 43-1 demonstrates a simplified version of this kind of keyboard trapping, compatible with NN4+/IE4+/W3C event models (but not fully functional in Safari 1.0). The message to the user is displayed in the status bar. Displaying the message there has the advantage of being less intrusive than an alert dialog box (and keeps the text insertion cursor in the text box), but it also means that users might not see the message. The `onsubmit` event handler in the listing prevents a press of the Enter key in this one-field form from reloading this sample page.

Listing 43-1: **Allowing Only Numbers into a Text Box**

```
<html>
   <head>
      <title>Letting Only Numbers Pass to a Form Field</title>
      <script type="text/javascript">
      function checkIt(evt) {
         evt = (evt) ? evt : window.event;
         var charCode = (evt.charCode) ? evt.charCode :
            ((evt.which) ? evt.which : evt.keyCode);
         if (charCode > 31 && (charCode < 48 || charCode > 57)) {
            status = "This field accepts numbers only.";
            return false;
         }
         status = "";
         return true;
      }
      </script>
   </head>
   <body>
      <h1>Letting Only Numbers Pass to a Form Field</h1>
      <hr />
      <form onsubmit="return false">
         Enter any positive integer: <input type="text" name="numeric"
         onkeypress="return checkIt(event)" />
      </form>
   </body>
</html>
```

Keyboard event monitoring isn't practical for most validation actions, however. For example, if the user is supposed to enter an e-mail address, you need to validate the complete entry for the presence of an @ symbol (via the `onchange` event handler). On the other hand, you can be granular about your validations and use both the `onchange` and `onkeypress` event handlers; you employ the latter for blocking invalid characters in e-mail addresses (such as spaces).

Batch mode validation

In all scriptable browsers, a form's `onsubmit` event handler cancels the submission if the handler evaluates to `return false`. Additional submission event cancelers include setting the IE4+ `event.returnValue` property to `false` and invoking the *evt*`.preventDefault()` method in NN6+/W3C (see Chapter 25 on event objects for details). You can see an example of the basic `return false` behavior in Listing 23-4 of Chapter 23. That example uses the results of a `window.confirm()` dialog box to determine the return value of the event handler. But you can also use a return value from a series of individual text box validation functions. If any one of the validations fails, the user is alerted and the submission is canceled.

Before you worry about two versions of validation routines loading down the scripts in your page, you'll be happy to know that you can reuse the same validation routines in both the real-time and batch validations. Later in this chapter, I demonstrate what I call "industrial-strength" data-entry validation adapted from a real intranet application. But before you get there, you should learn about general validation techniques that you can apply to both types of validations.

Designing Filters

The job of writing data-validation routines essentially involves designing filters that weed out characters or entries that don't fit your programming scheme. Whenever your filter detects an incorrect entry, it should alert the user about the nature of the problem and enable the user to correct the entry.

Before you put a text or `textarea` object into your document that invites users to enter data, you must decide if any possible entry can disturb the execution of the rest of your scripts. For example, if your script must have a number from that field to perform calculations, you must filter out any entry that contains letters or punctuation — except for periods if the program can accept floating-point numbers. Your task is to anticipate every possible entry users can make and allow only those entries your scripts can use.

Not every entry field needs a data-validation filter. For example, you may prompt a user for information that is eventually stored as a `document.cookie` or in a string database field on the server for future retrieval. If no further processing takes place on that information, you may not have to worry about the specific contents of that field.

One other design consideration is whether a text field is even the proper user interface element for the data required of the user. If the range of choices for a user entry is small (a dozen or fewer), a more sensible method is to avoid the data-entry problem altogether by turning that field into a `select` element. Your HTML attributes for the object ensure that you control the kind of entry made to that object. As long as your script knows how to deal with each of the options defined for that object, you're in the clear.

Building a Library of Filter Functions

A number of basic data-validation processes function repeatedly in form-intensive HTML pages. Filters for integers only, numbers only, empty entries, alphabet letters only, and the like are put to use every day. If you maintain a library of generalizable functions for each of your data-validation tasks, you can drop these functions into your scripts at a moment's notice and be assured that they will work. For NN3+ and IE4+, you can also create the library of validation functions as a separate `.js` library file and link the scripts into any HTML file that needs them.

Making validation functions generalizable requires careful choice of wording and logic so that they return Boolean values that make syntactical sense when called from elsewhere in your scripts. As you see later in this chapter, when you build a larger framework around smaller functions, each function is usually called as part of an `if...else` conditional statement. Therefore, assign a name that fits logically as part of an `if` clause in plain language. For example, you can name a function that checks whether an entry is empty `isEmpty()`. The calling statement for this function is

```
if (isEmpty(value)) { ... }
```

From a plain-language perspective, the expectation is that the function returns `true` if the passed value is empty. With this design, the statements nested in the `if` construction handle empty entry fields. I revisit this design later in this chapter when I start stacking multiple-function calls together in a larger validation routine.

To get you started with your library of validation functions, this chapter provides some building blocks that you can learn from and use as starting points for more specific filters of your own design. You can choose between fully backward-compatible (a) and regular expression (b) versions for many of them. Some of these functions are put to use in the JavaScript application in Chapter 50.

isEmpty()

This first function, shown in Listing 43-2a, checks to see if the incoming value is either empty or `null`. Adding a check for `null` means that you can use this function for purposes other than just text-object validation. For example, if another function defines three parameter variables, but the calling function passes only two, the third variable is set to `null`. If the script then performs a data-validation check on all parameters, the `isEmpty()` function responds that the `null` value is devoid of data.

Listing 43-2a: **Is an Entry Empty or Null?**

```
// general purpose function to see if an input value has been
// entered at all
function isEmpty(inputStr) {
   if (inputStr == null || inputStr == "") {
      return true;
   }
   return false;
}
```

This function uses a Boolean `or` operator (`||`) to test for the existence of a `null` value or an empty string in the value passed to the function. Because the name of the function implies a `true` response if the entry is empty, that value is the one that returns to the calling statement if either condition is true. Because a `return` statement halts further processing of a function, the `return false` statement lies outside of the `if` construction. If processing reaches this statement, the `inputStr` value has failed the test.

If this seems like convoluted logic — `return true` when the value is empty — you can also define a function that returns the inverse values. You can name it `isNotEmpty()`. As it turns out, however, typical processing of an empty entry is better served when the test returns a `true` than when the value is empty — aiding the `if` construction that calls the function in the first place.

The regular expression version, shown in Listing 43-2b, examines whether the input string has one or more characters of any kind in it.

Listing 43-2b: **Is an Entry Empty or Null? (regexp version)**

```
// regular expression function to see if an input value has been
// entered at all
function isEmpty(inputStr) {
   var re = /.+/;
   if (!inputStr.match(re)) {
      return true;
   }
   return false;
}
```

isPosInteger()

This next function examines each character of the value to make sure that only numbers from 0 through 9 with no punctuation or other symbols exist. The goal of the function in Listing 43-3a is to weed out any value that is not a positive integer.

Listing 43-3a: **Test for Positive Integers**

```
// general purpose function to see if a suspected numeric input
// is a positive integer
function isPosInteger(inputVal) {
   inputStr = inputVal.toString();
   for (var i = 0; i < inputStr.length; i++) {
      var oneChar = inputStr.charAt(i);
      if (oneChar < "0" || oneChar > "9") {
         return false;
      }
   }
   return true;
}
```

Notice that this function makes no assumption about the data type of the value that is passed as a parameter. If the value had come directly from a text object, it would already be a string and the line that forced data conversion to a string would be unnecessary. But to generalize the function, the conversion is included to accommodate the possibility that it may be called from another statement that has a numeric value to check.

The function requires you to convert the input value to a string because it performs a character-by-character analysis of the data. A `for` loop picks apart the value one character at a time. Rather than force the script to invoke the *`string`*`.charAt()` method twice for each time through the loop (inside the `if` condition), one statement assigns the results of the method to a variable, which is then used twice in the `if` condition. Placing the results of the `charAt()` method into a variable makes the `if` condition shorter and easier to read and also is microscopically more efficient.

In the `if` condition, the ASCII value of each character is compared against the range of 0 through 9. This method is safer than comparing numeric values of the single characters because one of the characters can be nonnumeric. (You can encounter all kinds of other problems trying to convert that character to a number for numeric comparison.) The ASCII value, on the other hand, is neutral about the meaning of a character: If the ASCII value is less than 0 or greater than 9, the character is not valid for a genuine positive integer. The function bounces the call with a false reply. On the other hand, if the `for` loop completes its traversal of all characters in the value without a hitch, the function returns `true`.

You may wonder why this validation function doesn't use the `parseInt()` global function (see Chapter 34). That function returns `NaN` only if the first character of the input string is not a number. But because `parseInt()` and `parseFloat()` peel off any initial numeric values from a string, neither returns `NaN` if the input string is, for example, `35a`.

The regular expression version (Listing 43-3b) allows a string consisting entirely of any number of numerals.

Listing 43-3b: **Test for Positive Integers (regexp version)**

```
// regular expression function to see if a suspected numeric input
// is a positive integer
function isPosInteger(inputStr) {
   var re = /^\d*$/;
   inputStr = inputStr.toString();
   if (!inputStr.match(re)) {
      return false;
   }
   return true;
}
```

isInteger()

The next possibility includes the entry of a negative integer value. Listing 43-4a shows that you must add an extra check for a leading negation sign.

Listing 43-4a: **Checking for Leading Minus Sign**

```
// general purpose function to see if a suspected numeric input
// is a positive or negative integer
function isInteger(inputVal) {
   inputStr = inputVal.toString();
   for (var i = 0; i < inputStr.length; i++) {
      var oneChar = inputStr.charAt(i);
      if (i == 0 && oneChar == "-") {
         continue;
      }
      if (oneChar < "0" || oneChar > "9") {
         return false;
      }
   }
   return true;
}
```

When a script can accept a negative integer, the filter must enable the leading minus sign to pass unscathed. You cannot just add the minus sign to the `if` condition of Listing 43-4a because you can accept that symbol only when it appears in the first position of the value — anywhere else makes the value an invalid number. To handle the possibility of a leading minus sign, you add another `if` statement whose condition looks for a special combination: the first character of the string (as indexed by the loop-counting variable) and the minus character. If both of these conditions are met, execution immediately loops back around to the update expression of the `for` loop (because of the `continue` statement) rather than carrying out the second `if` statement, which would obviously fail. By putting the `i == 0` comparison operation at the front of the condition, you ensure the entire condition short circuits to `false` for all subsequent iterations through the loop.

The regular expression version (Listing 43-4b) is identical to the one for `isPosInteger()` except that the regular expression allows for one leading hyphen.

Listing 43-4b: **Checking for Leading Minus Sign (regexp version)**

```
// regular expression function to see if a suspected numeric input
// is a positive or negative integer
function isInteger(inputStr) {
   var re = /^[-]?\d*$/;
   inputStr = inputStr.toString();
   if (!inputStr.match(re)) {
      return false;
   }
   return true;
}
```

isNumber()

The final numeric filter function in this series enables any integer or floating-point number to pass while filtering out all others (Listing 43-5a). All that distinguishes an integer from a floating-point number for data-validation purposes is the decimal point.

Listing 43-5a: **Testing for a Decimal Point**

```
// general purpose function to see if a suspected numeric input
// is a positive or negative number
function isNumber(inputVal) {
   oneDecimal = false;
   inputStr = inputVal.toString();
   for (var i = 0; i < inputStr.length; i++) {
      var oneChar = inputStr.charAt(i);
      if (i == 0 && oneChar == "-") {
         continue;
      }
      if (oneChar == "." && !oneDecimal) {
         oneDecimal = true;
         continue;
      }
      if (oneChar < "0" || oneChar > "9") {
         return false;
      }
   }
   return true;
}
```

Anticipating the worst, however, the function cannot simply treat a decimal point at any position within the string as a valid character. Such an act assumes that no one would ever enter more than one decimal point into a numeric text field. Only one decimal point is allowed for this function (as well as for JavaScript math). Therefore, you add a Boolean flag variable (`oneDecimal`) to the function and a separate `if` condition that sets that flag to `true` when the function encounters the first decimal point. Should another decimal point appear in the string, the final `if` statement gets a crack at the character. Because the character falls outside the ASCII range of 0 through 9, it fails the entire function.

If you want to accept only positive floating-point numbers, you can make a new version of this function by removing the statement that lets the leading minus sign through. Be aware that this function works only for values that are not represented in exponential notation.

A slight modification to the regular expression from Listing 43-4b — allowing a single decimal point between any number of numerals — turns the validation function into one that tests for any positive or negative number, with or without numbers to the right of the decimal, as shown in Listing 43-5b.

Listing 43-5b: **Testing for a Decimal Point (regexp version)**

```
// regular expression function to see if a suspected numeric input
// is a positive or negative number
function isNumber(inputStr) {
   var re = /^[-]?\d*\.?\d*$/;
   inputStr = inputStr.toString();
   if (!inputStr.match(re)) {
      return false;
   }
   return true;
}
```

Custom validation functions

The listings shown so far in this chapter should give you plenty of source material to use in writing customized validation functions for your applications. Listing 43-6 shows an example of such an application-specific variation (extracted from the application in Chapter 50).

Listing 43-6: **A Custom Validation Function**

```
// function to determine if value is in acceptable range
// for this application
function inRange(inputStr) {
   num = parseInt(inputStr);
   if (num < 1 || num > 586 && num < 596 || num > 599 && num < 700 ||
      num > 728) {
      return false;
   }
   return true;
}
```

For this application, you need to see if an entry falls within multiple ranges of acceptable numbers. The first statement of the `inRange()` function converts the incoming value to a number (via the `parseInt()` function) so that the value can be compared numerically against maximum and minimum values of several ranges within the database. Following the logic of the previous validation functions, the `if` condition looks for values outside the acceptable range, so it can alert the user and return a `false` value.

The `if` condition is quite a long sequence of operators. As you noticed in the list of operator precedence (see Chapter 32), the Boolean `and` operator (`&&`) has precedence over the Boolean `or` operator (`||`). Therefore, the `and` expressions evaluate first, followed by the `or` expressions. Parentheses may help you better visualize what's going on in that monster condition:

```
if (num < 1 || (num > 586 && num < 596) ||(num > 599 && num < 700) ||
   num > 728)
```

In other words, you exclude four possible ranges from consideration:

- ✦ Values less than 1
- ✦ Values between 586 and 596
- ✦ Values between 599 and 700
- ✦ Values greater than 728

Any value for which any one of these tests is true yields a Boolean `false` from this function. Combining all these tests into a single condition statement eliminates the need to construct an otherwise complex series of nested `if` constructions.

Combining Validation Functions

When you design a page that requests a particular kind of text input from a user, you often need to call more than one data-validation function to handle the entire job. For example, if you merely want to test for a positive integer entry, your validation should test for the presence of any entry as well as the validation as an integer.

After you know the kind of permissible data that your script will use after validation, you're ready to plot the sequence of data validation. Because each page's validation task is different, I supply some guidelines to follow in this planning rather than prescribe a fixed route for all to take.

My preferred sequence is to start with examinations that require less work and increase the intensity of validation detective work with succeeding functions. I borrow this tactic from real life: When a lamp fails to turn on, I look for a pulled plug or a burned-out lightbulb before tearing the lamp's wiring apart to look for a short.

Using the data-validation sequence from the data-entry field (which must be a three-digit number within a specified range) in Chapter 50, I start with the test that requires the least amount of work: Is there an entry at all? After my script is ensured an entry of some kind exists, it next checks whether that entry is "all numbers as requested of the user." If so, the script compares the number against the ranges of numbers in the database.

To make this sequence work together efficiently, I create a master validation function consisting of nested `if...else` statements. Each `if` condition calls one of the generalized data-validation functions. Listing 43-7 shows the master validation function.

Listing 43-7: **Master Validation Function**

```
// Master value validator routine
function isValid(inputStr) {
   if (isEmpty(inputStr)) {
      alert("Please enter a number into the field before clicking the
         button.");
      return false;
   } else {
      if (!isNumber(inputStr)) {
         alert("Please make sure entries are numbers only.");
         return false;
      } else {
         if (!inRange(inputStr)) {
```

```
            var msg = "Sorry, the number you entered is not part of our
               database.";
            msg += "Try another three-digit number.";
            alert(msg);
            return false;
         }
      }
   }
   return true;
}
```

This function, in turn, is called by the function that controls most of the work in this application. All that the main function wants to know is whether the entered number is valid. The details of validation are handed off to the `isValid()` function and its special-purpose validation testers.

I construct the logic in Listing 43-7 so that if the input value fails to be valid, the `isValid()` function alerts the user of the problem and returns `false`. That means I have to watch my `true`s and `false`s very carefully.

In the first validation test, an empty value is a bad thing; thus, when the `isEmpty()` function returns `true`, the `isValid()` function returns `false` because an empty string is not a valid entry. In the second test, a number value is good so the logic has to flip 180 degrees. The `isValid()` function returns `false` only if the `isNumber()` function returns `false`. But because `isNumber()` returns `true` when the value is a number, I switch the condition to test for the opposite results of the `isNumber()` function by negating the function name (preceding the function with the Boolean `not` (`!`) operator). This operator works only with a value that evaluates to a Boolean expression — which the `isNumber()` function always does. The final test for being within the desired range works on the same basis as `isNumber()`, using the Boolean `not` operator to turn the results of the `inRange()` function into the method that works best for this sequence.

Finally, if all validation tests fail to find bad or missing data, the entire `isValid()` function returns `true`. The statement that calls this function can now proceed with processing, ensured that the value entered by the user will work.

The Boolean logic of the backward-compatible and regular expression versions of the core validation functions is the same. Therefore, you can use either version with the master validation function.

There is one additional point worth reinforcing, especially for newcomers. Although all these functions seem to be passing around the same input string as a parameter, notice that any changes made to the value (such as converting it to a string or number) are kept private to each function. These subfunctions never touch the original value in the calling function — they work only with copies of the original value. Therefore, even after the data validation takes place, the original value is in its original form and ready to go.

Date and Time Validation

You can scarcely open a bigger can of cultural worms than when trying to program around the various date and time formats in use around the world. If you have ever looked through the possible settings in your computer's operating system, you can begin to understand the difficulty of this issue.

Trying to write JavaScript that accommodates all of the world's date and time formats for validation is an enormous, if not wasteful, challenge. It's one thing to validate that a text box contains data in the form `xx/xx/xxxx`, but there are also valid value concerns that can get very messy on an international basis. For example, while North America typically uses the `mm/dd/yyyy` format, a large portion of the rest of the world uses `dd/mm/yyyy` (with different delimiter characters, as well). Therefore, how should your validation routine treat the entry `20/03/2002`? Is it incorrect because there are not 20 months in a year; or is it correct as March 20th? To query a user for this kind of information, I suggest you divide the components into individually validated fields (separate text objects for hours and minutes) or make `select` element entries whose individual values can be assembled at submit time into a hidden date field for processing by the database that needs the date information. (Alternately, you can let your server CGI handle the conversion.)

Despite my encouragement to "divide and conquer" date entries, there may be situations in which you feel it's safe to provide a single text box for date entry (perhaps for a form that is used on a corporate intranet strictly by users in one country). You see some more sophisticated code later in this chapter, but a "quick-and-dirty" solution runs along these lines:

1. Use the entered data (for example, in `mm/dd/yyyy` format) as a value passed to the `new Date()` constructor function.
2. From the newly created date object, extract each of the three components (month, day, and year) into separate numeric values (with the help of `parseInt()`).
3. Compare each of the extracted values against the corresponding date, month, and year values returned by the date object's `getDate()`, `getMonth()`, and `getFullYear()` methods (adjusting for zero-based values of `getMonth()`).
4. If all three pairs of values match, the entry is apparently valid.

Listing 43-8a puts this action sequence to work in a backward-compatible way. The `validDate()` function receives a reference to the field being checked. A copy of the field's value is made into a date object, and its components are read. If any part of the date conversion or component extraction fails (because of improperly formatted data or unexpected characters), one or more of the variable values becomes `NaN`. This code assumes that the user enters a date in the `mm/dd/yyyy` format, which is the sequence that the `Date` object constructor expects its data. If the user enters `dd/mm/yyyy`, the validation fails for any day beyond the twelfth.

Listing 43-8a: **Simple Date Validation**

```
<html>
  <head>
     <title>Simple Date Validation</title>
     <script type="text/javascript">
     function validDate(fld) {
        var testMo, testDay, testYr, inpMo, inpDay, inpYr, msg;
        var inp = fld.value;
        status = "";
        // attempt to create date object from input data
        var testDate = new Date(inp);
        // extract pieces from date object
        testMo = testDate.getMonth() + 1;
        testDay = testDate.getDate();
        testYr = testDate.getFullYear();
```

```
         // extract components of input data
         inpMo = parseInt(inp.substring(0, inp.indexOf("/")), 10);
         inpDay = parseInt(inp.substring((inp.indexOf("/") + 1),
            inp.lastIndexOf("/")), 10);
         inpYr = parseInt(inp.substring((inp.lastIndexOf("/") + 1),
            inp.length), 10);
         // make sure parseInt() succeeded on input components
         if (isNaN(inpMo) || isNaN(inpDay) || isNaN(inpYr)) {
            msg = "There is some problem with your date entry.";
         }
         // make sure conversion to date object succeeded
         if (isNaN(testMo) || isNaN(testDay) || isNaN(testYr)) {
            msg = "Couldn't convert your entry to a valid date. Try again.";
         }
         // make sure values match
         if (testMo != inpMo || testDay != inpDay || testYr != inpYr) {
            msg = "Check the range of your date value.";
         }
         if (msg) {
            // there's a message, so something failed
            alert(msg);
            // work around IE timing problem with alert by
            // invoking a focus/select function through setTimeout();
            // must pass along reference of fld (as string)
            setTimeout("doSelection(document.forms['" +
               fld.form.name + "'].elements['" + fld.name + "'])", 0);
            return false;
         } else {
            // everything's OK; if browser supports new date method,
            // show just date string in status bar
            status = (testDate.toLocaleDateString) ?
               testDate.toLocaleDateString() : "Date OK";
            return true;
         }
      }

      // separate function to accommodate IE timing problem
      function doSelection(fld) {
         fld.focus();
         fld.select();
      }
      </script>
   </head>
   <body>
      <h1>Simple Date Validation</h1>
      <hr />
      <form name="entryForm" onsubmit="return false">
         Enter any date (mm/dd/yyyy): <input type="text" name="startDate"
         onchange="validDate(this)" />
      </form>
   </body>
</html>
```

You can also use regular expressions for date validation. A version of the above listing using regular expressions at the core is found in Listing 43-8b.

Listing 43-8b: **Simple Date Validation**

```
<html>
   <head>
      <title>Simple Date Validation (regexp)</title>
      <script type="text/javascript">
      function validDate(fld) {
         var testMo, testDay, testYr, inpMo, inpDay, inpYr, msg;
         var inp = fld.value;
         status = "";
         var re = /\b\d{1,2}[\/-]\d{1,2}[\/-]\d{4}\b/;
         if (re.test(inp)) {
            var delimChar = (inp.indexOf("/") != -1) ? "/" : "-";
            var delim1 = inp.indexOf(delimChar);
            var delim2 = inp.lastIndexOf(delimChar);
            mo = parseInt(inp.substring(0, delim1), 10);
            day = parseInt(inp.substring(delim1+1, delim2), 10);
            yr = parseInt(inp.substring(delim2+1), 10);
            var testDate = new Date(yr, mo-1, day);
            if (testDate.getDate() == day) {
                if (testDate.getMonth() + 1 == mo) {
                    if (testDate.getFullYear() == yr) {
                        msg = "";
                    } else {
                        msg = "There is a problem with the year entry.";
                    }
                } else {
                    msg = "There is a problem with the month entry.";
                }
            } else {
                msg = "There is a problem with the date entry.";
            }
         } else {
            msg = "Incorrect date format. Enter as mm/dd/yyyy.";
         }
         if (msg) {
            // there's a message, so something failed
            alert(msg);
            // work around IE timing problem with alert by
            // invoking a focus/select function through setTimeout();
            // must pass along reference of fld (as string)
            setTimeout("doSelection(document.forms['" +
               fld.form.name + "'].elements['" + fld.name + "'])", 0);
            return false;
         } else {
            // everything's OK; if browser supports new date method,
            // show just date string in status bar
            window.status = (testDate.toLocaleDateString) ?
               testDate.toLocaleDateString() : "Date OK";
            return true;
         }
      }
```

```
      // separate function to accommodate IE timing problem
      function doSelection(fld) {
         fld.focus();
         fld.select();
      }
      </script>
   </head>
   <body>
      <h1>Simple Date Validation</h1>
      <hr />
      <form name="entryForm" onsubmit="return false">
         Enter any date (mm/dd/yyyy): <input type="text" name="startDate"
         onchange="validDate(this)" />
      </form>
   </body>
</html>
```

Selecting Text Fields for Reentry

During both real-time and batch validations, it is especially helpful to the user if your code—upon discovering an invalid entry—not only brings focus to the subject text field, but also selects the content for the user. By preselecting the entire field, you make it easy for the user to just retype the data into the field for another attempt (or to begin using the left and right arrow keys to move the insertion cursor for editing). The reverse type on the field text also helps bring attention to the field. (Not all operating systems display a special rectangle around a focused text field.)

Form fields have both `focus()` and `select()` methods, which you should invoke for the subject field in that order. IE for Windows, however, exhibits undesirable behavior when trying to focus and select a field immediately after you close an alert dialog box. In most cases, the field does not keep its focus or selection. This is a timing problem, but one that you can cure by processing the focus and select actions through a `setTimeout()` method. The bottom of the script code of Listing 43-9 demonstrates how to do this.

Method calls to the form field reside in a separate function (called `doSelection()` in this example). Obviously, the methods need a reference to the desired field, so the `doSelection()` function requires access to that reference. You can use a global variable to accomplish this (set the value in the validation function; read it in the `doSelection()` function), but globals are not elegant solutions to passing transient data. Even though the validation function receives a reference to the field, that is an object reference, and the `setTimeout()` function's first parameter cannot be anything but a string value. Therefore, the reference to the text field provides access to names of both the form and field. The names fill in as index values for arrays so that the assembled string (upon being invoked) evaluates to a valid object reference:

```
"doSelection(document.forms['" + fld.form.name + "'].elements['" + fld.name +
"'])"
```

Notice the generous use of built-in `forms` and `elements` object arrays, which allow the form and field names to assemble the reference without resorting to the onerous `eval()` function.

For timing problems such as this one, no additional time is truly needed to let IE recover from whatever ails it. Thus, the time parameter is set to 0 milliseconds. Using the `setTimeout()` portal is enough to make everything work. There is no penalty for using this construction with NN or MacIE, even though they don't need it.

An "Industrial-Strength" Validation Solution

I had the privilege of working on a substantial intranet project that included dozens of forms, often with two or three different kinds of forms displayed simultaneously within a frameset. Data-entry accuracy was essential to the validity of the entire application. My task was to devise a data-entry validation strategy that not only ensured accurate entry of data types for the underlying (SQL) database, but also intelligently prompted users who made mistakes in their data entry.

Structure

From the start, the validation routines were to be in a client-side library linked in from an external `.js` file. That would allow all forms to share the validation functions. Because there were multiple forms displayed in a frameset, it would prove too costly in download time and memory requirements to include the `validations.js` file in every frame's document. Therefore, the library was moved to load in with the frameset. The `<script src="validations.js"></script>` tag set went in the Head portion of the framesetting document.

This logical placement presented a small challenge for the workings of the validations because there had to be two-way conversations between a validation function (in the frameset) and a form element (nested in a frame). The mechanism required that a reference to the frame containing the form element be passed as part of the validation routine so that the validation script could make corrections, automatic formatting, and erroneous field selections from the frameset document's script. (In other words, the frameset script needed a path back to the form element making the validation call.)

Dispatch mechanism

From the specification drawn up for the application, it is clear that there are more than two dozen specific types of validations across all the forms. Moreover, multiple programmers work on different forms. It is helpful to standardize the way validations are called, regardless of the validation type (number, string, date, phone number, and so on).

My idea was to create one `validate()` function that contained parameters for the current frame, the current form element, and the type of validation to perform. This would make it clear to anyone reading the code later that an event handler calling `validate()` performed validation, and the details of the code were in the `validations.js` library file.

In `validations.js`, I converted a string name of a validation type into the name of the function that performs the validation in order to make this idea work. As a bridge between the two, I created what I called a *dispatch lookup table* for all the primary validation routines that would be called from the forms. Each entry of the lookup table had a label consisting of the name of the validation and a method that invoked the function. Listing 43-9 shows an excerpt of the entire lookup table creation mechanism.

Listing 43-9: **Creating the Dispatch Lookup Table**

```
/*
   Begin validation dispatching mechanism
*/
function dispatcher(validationFunc) {
   this.doValidate = validationFunc;
}
var dispatchLookup = new Array();
dispatchLookup["isNotEmpty"] = new dispatcher(isNotEmpty);
dispatchLookup["isPositiveInteger"] = new dispatcher(isPositiveInteger);
dispatchLookup["isDollarsOnly8"] = new dispatcher(isDollarsOnly8);
dispatchLookup["isUSState"] = new dispatcher(isUSState);
dispatchLookup["isZip"] = new dispatcher(isZip);
dispatchLookup["isExpandedZip"] = new dispatcher(isExpandedZip);
dispatchLookup["isPhone"] = new dispatcher(isPhone);
dispatchLookup["isConfirmed"] = new dispatcher(isConfirmed);
dispatchLookup["isNY"] = new dispatcher(isNY);
dispatchLookup["isNum16"] = new dispatcher(isNum16);
dispatchLookup["isM90_M20Date"] = new dispatcher(isM90_M20Date);
dispatchLookup["isM70_0Date"] = new dispatcher(isM70_0Date);
dispatchLookup["isM5_P10Date"] = new dispatcher(isM5_P10Date);
dispatchLookup["isDateFormat"] = new dispatcher(isDateFormat);
```

Each entry of the array is assigned a `dispatcher` object, whose custom object constructor assigns a function reference to the object's `doValidate()` method. For these assignment statements to work, their corresponding functions must be defined earlier in the document. You can see some of these functions later in this section.

The link between the form elements and the dispatch lookup table is the `validate()` function, shown in Listing 43-10. A call to `validate()` requires a minimum of three parameters, as shown in the following example:

```
<input type="text" name="phone" size="10"
onchange="parent.validate(window, this, 'isPhone')" />
```

The first is a reference to the frame containing the document that is calling the function (passed as a reference to the current window). The second parameter is a reference to the form control element itself (using the `this` operator). After that, you see one or more individual validation function names as strings. This last design allows more than one type of validation to take place with each call to `validate()` (for example, in case a field must check for a data type and check that the field is not empty).

Listing 43-10: **Main Validation Function**

```
// main validation function called by form event handlers
function validate(frame, field, method) {
   gFrame = frame;
   gField =  window.frames[frame.name].document.forms[0].elements[field.name];
   var args = validate.arguments;
```

Continued

Listing 43-10 *(continued)*

```
   for (i = 2; i < args.length; i++) {
      if (!dispatchLookup[args[i]].doValidate()) {
         return false;
      }
   }
   return true;
}
```

In the `validate()` function, the frame reference is assigned to a global variable that is declared at the top of the `validations.js` file. Validation functions in this library need this information to build a reference back to a companion field required of some validations (explained later in this section). A second global variable contains a reference to the calling form element. Because the form element reference by itself does not contain information about the frame in which it lives, the script must build a reference out of the information passed as parameters. The reference must work from the framesetting document down to the frame, its form, and form element name. Therefore, I use the `frame` and `field` object references to get their respective names (within the `frames` and `elements` arrays) to assemble the text field's object reference; the resulting value is assigned to the `gField` global variable. I choose to use global variables in this case because passing these two values to numerous nested validation functions could be difficult to track reliably. Instead, the only parameter passed to specific validation functions is the value under test.

Next, the script creates an array of all arguments passed to the `validate()` function. A `for` loop starts with an index value of `2`, the third parameter containing the first validation function name. For each one, the named item's `doValidate()` method is called. If the validation fails, this function returns `false`; but if all validations succeed, this function returns `true`. Later you see that this function's returned value is the one that allows or disallows a form submission.

Sample validations

Above the dispatching mechanism in the `validations.js` are the validation functions themselves. Many of the named validation functions have supporting utility functions that other named validation functions often use. Because of the eventual large size of this library file (the production version was about 40KB), I organized the functions into two groups: the named functions first, and the utility functions below them (but still before the dispatching mechanism at the bottom of the document).

To demonstrate how some of the more common data types are validated for this application, I show several validation functions and, where necessary, their supporting utility functions. Figure 43-1 shows a sample form that takes advantage of these validations. (You have a chance to try it later in this chapter.) When you are dealing with critical corporate data, you must go to extreme lengths to ensure valid data. And to help users see their mistakes quickly, you need to build some intelligence into validations where possible.

U.S. state name

The design specification for forms that require entry of a U.S. state calls for entry of the state's two-character abbreviation. A companion field to the right displays the entire state name as user feedback verification. The `onchange` event handler not only calls the validation, but it also feeds the focus to the field following the expanded state field so users are less likely to type into it.

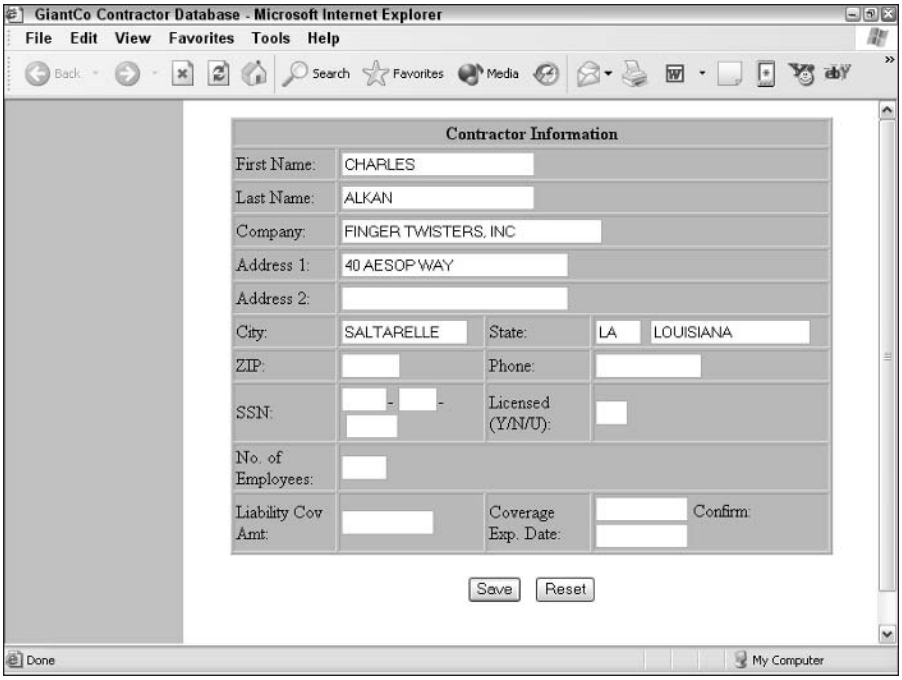


Figure 43-1: Sample form for industrial-strength validations.

Before the validation can even get to the expansion part, it must first validate that the entry is a valid, two-letter abbreviation. Because I need both the abbreviation and the full state name for this validation, I create an array of all the states using each state abbreviation as the index label for each entry. Listing 43-11 shows that array creation.

Listing 43-11: **Creating a U.S. States Array**

```
// States array
var USStates = new Array(51);
USStates["AL"] = "ALABAMA";
USStates["AK"] = "ALASKA";
USStates["AZ"] = "ARIZONA";
USStates["AR"] = "ARKANSAS";
USStates["CA"] = "CALIFORNIA";
USStates["CO"] = "COLORADO";
USStates["CT"] = "CONNECTICUT";
USStates["DE"] = "DELAWARE";
USStates["DC"] = "DISTRICT OF COLUMBIA";
USStates["FL"] = "FLORIDA";
USStates["GA"] = "GEORGIA";
USStates["HI"] = "HAWAII";
USStates["ID"] = "IDAHO";
USStates["IL"] = "ILLINOIS";
USStates["IN"] = "INDIANA";
```

Continued

Listing 43-11 *(continued)*

```
USStates["IA"] = "IOWA";
USStates["KS"] = "KANSAS";
USStates["KY"] = "KENTUCKY";
USStates["LA"] = "LOUISIANA";
USStates["ME"] = "MAINE";
USStates["MD"] = "MARYLAND";
USStates["MA"] = "MASSACHUSETTS";
USStates["MI"] = "MICHIGAN";
USStates["MN"] = "MINNESOTA";
USStates["MS"] = "MISSISSIPPI";
USStates["MO"] = "MISSOURI";
USStates["MT"] = "MONTANA";
USStates["NE"] = "NEBRASKA";
USStates["NV"] = "NEVADA";
USStates["NH"] = "NEW HAMPSHIRE";
USStates["NJ"] = "NEW JERSEY";
USStates["NM"] = "NEW MEXICO";
USStates["NY"] = "NEW YORK";
USStates["NC"] = "NORTH CAROLINA";
USStates["ND"] = "NORTH DAKOTA";
USStates["OH"] = "OHIO";
USStates["OK"] = "OKLAHOMA";
USStates["OR"] = "OREGON";
USStates["PA"] = "PENNSYLVANIA";
USStates["RI"] = "RHODE ISLAND";
USStates["SC"] = "SOUTH CAROLINA";
USStates["SD"] = "SOUTH DAKOTA";
USStates["TN"] = "TENNESSEE";
USStates["TX"] = "TEXAS";
USStates["UT"] = "UTAH";
USStates["VT"] = "VERMONT";
USStates["VA"] = "VIRGINIA";
USStates["WA"] = "WASHINGTON";
USStates["WV"] = "WEST VIRGINIA";
USStates["WI"] = "WISCONSIN";
USStates["WY"] = "WYOMING";
```

The existence of this array comes in handy in determining if the user enters a valid, two-state abbreviation. Listing 43-12 shows the actual `isUSState()` validation function that puts this array to work.

The function's first task is to assign an uppercase version of the entered value to a local variable (`inputStr`), which is the value being analyzed throughout the rest of the function. If the user enters something in the field (`length > 0`) but no entry in the `USStates` array exists for that value, the entry is not a valid state abbreviation. Time to go to work to help out the user.

Listing 43-12: **Validation Function for U.S. States**

```
// input value is a U.S. state abbreviation; set entered value to all uppercase
// also set companion field (NAME="<xxx>_expand") to full state name
function isUSState() {
   var inputStr = gField.value.toUpperCase();
   if (inputStr.length > 0 && USStates[inputStr] == null) {
      var msg = "";
      var firstChar = inputStr.charAt(0);
      if (firstChar == "A") {
         msg += "\n(Alabama = AL; Alaska = AK; Arizona = AZ; Arkansas = AR)";
      }
      if (firstChar == "D") {
         msg += "\n(Delaware = DE; District of Columbia = DC)";
      }
      if (firstChar == "I") {
         msg += "\n(Idaho = ID; Illinois = IL; Indiana = IN; Iowa = IA)";
      }
      if (firstChar == "M") {
         msg += "\n(Maine = ME; Maryland = MD; Massachusetts = MA;  " +
            "Michigan = MI; Minnesota = MN; Mississippi = MS; " +
            "Missouri = MO; Montana = MT)";
      }
      if (firstChar == "N") {
         msg += "\n(Nebraska = NE; Nevada = NV)";
      }
      alert("Check the spelling of the state abbreviation." + msg);
      gField.focus();
      gField.select();
      return false;
   }
   gField.value = inputStr;
   var expandField =
      window.frames[gFrame.name].document.forms[0].elements[gField.name +
      "_expand"];
   expandField.value = USStates[inputStr];
   return true;
}
```

The function assumes that the user tried to enter a valid state abbreviation but either had incorrect source material or momentarily forgot a particular state's abbreviation. Therefore, the function examines the first letter of the entry. If that first letter is any one of the five identified as causing the most difficulty, a legend for all states beginning with that letter is assigned to the `msg` variable (for running on newer browsers only, a `switch` construction is preferred). An alert message displays the generic alert, plus any special legend if one is assigned to the `msg` variable. When the user closes the alert, the field has focus and its text is selected. (This application runs solely on Navigator, so the IE `setTimeout()` workaround isn't needed — but you can add it very easily, especially thanks to the global variable reference for the field.) The function returns `false` at this point.

If, on the other hand, the abbreviation entry is a valid one, the field is handed the uppercase version of the entry. The script then uses the two global variables set in `validate()` to create a reference to the expanded display field (whose name must be the same as the entry field plus `"_expand"`). That expanded display field is then supplied the `USStates` array entry value corresponding to the abbreviation label. All is well with this validation, so it returns `true`.

You can see here that the so-called validation routine is doing far more than simply checking validity of the data. By communicating with the field, converting its contents to uppercase, and talking to another field in the form, a simple call to the validation function yields a lot of mileage.

Date validation

Many of the forms in this application have date fields. In fact, dates are an important part of the data maintained in the database behind the forms. All users of this application are familiar with standard date formats in use in the United States, so I don't have to worry about the possibility of cultural variations in date formats. Even so, I want the date entry to accommodate the common date formats, such as `mmddyyyy`, `mm/dd/yyyy`, and `mm-dd-yyyy` (as well as accommodate two-digit year entries spanning 1930 to 2029).

The plan also calls for going further in helping users enter dates within certain ranges. For example, a field used for a birth date (the listings are for medical professionals) should recommend dates starting no more than 90 years, and no less than 20 years, from the current date. And to keep this application running well into the future, the ranges should be on a sliding scale from the current year, no matter when it might be. Whatever the case, the date range validation is only a recommendation and not a transaction stopper.

Rather than create separate validation functions for each date field, I create a system of reusable validation functions for each date range (several fields on different forms require the same date ranges). Each one of these individual functions calls a single, generic date-validation function that handles the date-range checking. Listing 43-13 shows a few examples of these individual range-checking functions.

Listing 43-13: **Date Range Validations**

```
 // Date Minus 90/Minus 20
function isM90_M20Date() {
   if (gField.value.length == 0)
      return true;
   var thisYear = getTheYear();
   return isDate((thisYear - 90),(thisYear - 20));
}

// Date Minus 70/Minus 0
function isM70_0Date() {
   if (gField.value.length == 0)
      return true;
   var thisYear = getTheYear();
   return isDate((thisYear - 70),(thisYear));
}

// Date Minus 5/Plus 10
```

```
function isM5_P10Date() {
   if (gField.value.length == 0)
      return true;
   var thisYear = getTheYear();
   return isDate((thisYear - 5),(thisYear + 10));
}
```

The naming convention I create for the functions includes the two range components relative to the current date. A letter "M" means the range boundary is minus a number of years from the current date; "P" means the range is plus a number of years. If the boundary should be the current year, a zero is used. Therefore, the `isM5_P10Date()` function performs range checking for boundaries between 5 years before and 10 years after the current year.

Before performing any range checking, each function makes sure there is some value to validate. If the field entry is empty, the function returns `true`. This is fine here because dates are not required when the data is unknown.

Next, the functions get the current four-digit year. The code here had to work originally with browsers that did not have the `getFullYear()` method available yet. Therefore, the Y2K fix described in Chapter 29 was built into the application:

```
function getTheYear() {
   var thisYear = (new Date()).getYear();
   thisYear = (thisYear < 100) ? thisYear + 1900 : thisYear;
   return thisYear;
}
```

The final call from the range validations is to a common `isDate()` function, which handles not only the date range validation but also the validation for valid dates (for example, making sure that September has only 30 days). Listing 43-14 shows this monster-sized function. Because of the length of this function, I interlace commentary within the code listing.

Listing 43-14: **Primary Date Validation Function**

```
// date field validation (called by other validation functions that specify
// minYear/maxYear)
function isDate(minYear,maxYear,minDays,maxDays) {
   var inputStr = gField.value;
```

To make it easier to work with dates supplied with delimiters, I first convert hyphen delimiters to slash delimiters. The pre-regular expression `replaceString()` function is the same one described in Chapter 27; it is located in the utility functions part of the `validations.js` file.

```
   // convert hyphen delimiters to slashes
   while (inputStr.indexOf("-") != -1) {
      inputStr = replaceString(inputStr,"-","/");
   }
```

For validating whether the gross format is OK, I check whether zero or two delimiters appear. If the value contains only one delimiter, the overall formatting is not acceptable. The error alert shows models for acceptable date-entry formats.

```
   var delim1 = inputStr.indexOf("/");
   var delim2 = inputStr.lastIndexOf("/");
   if (delim1 != -1 && delim1 == delim2) {
      // there is only one delimiter in the string
      alert("The date entry is not in an acceptable format.\n\nYou can enter
dates in the following formats: mmddyyyy, mm/dd/yyyy, or mm-dd-yyyy.  (If the
month or date data is not available, enter \'01\' in the appropriate
location.)");
      gField.focus();
      gField.select();
      return false;
   }
```

If there are two delimiters, I tear apart the string into components for month, day, and year. Because two-digit entries can begin with zeros, I make sure the `parseInt()` functions specify base-10 conversions.

```
   if (delim1 != -1) {
      // there are delimiters; extract component values
      var mm = parseInt(inputStr.substring(0,delim1),10);
      var dd = parseInt(inputStr.substring(delim1 + 1,delim2),10);
      var yyyy = parseInt(inputStr.substring(delim2 + 1, inputStr.length),10);
```

For no delimiters, I tear apart the string and assume two-digit entries for the month and day and two or four digits for the year.

```
   } else {
      // there are no delimiters; extract component values
      var mm = parseInt(inputStr.substring(0,2),10);
      var dd = parseInt(inputStr.substring(2,4),10);
      var yyyy = parseInt(inputStr.substring(4,inputStr.length),10);
   }
```

The `parseInt()` functions reveal whether any entry is not a number by returning `NaN`, so I check whether any of the three values is not a number. If so, an alert signals the formatting problem and supplies acceptable models.

```
   if (isNaN(mm) || isNaN(dd) || isNaN(yyyy)) {
      // there is a non-numeric character in one of the component values
      alert("The date entry is not in an acceptable format.\n\nYou can enter
         dates in the following formats: mmddyyyy, mm/dd/yyyy, or
         mm-dd-yyyy.");
      gField.focus();
      gField.select();
      return false;
   }
```

Next, I perform some gross range validation on the month and date to make sure that months are entered from 1 to 12 and dates from 1 to 31. I take care of aligning exact month lengths later.

```
   if (mm < 1 || mm > 12) {
      // month value is not 1 thru 12
      alert("Months must be entered between the range of 01 (January) and 12
         (December).");
      gField.focus();
      gField.select();
      return false;
   }
```

```
if (dd < 1 || dd > 31) {
   // date value is not 1 thru 31
   alert("Days must be entered between the range of 01 and a maximum of 31
      (depending on the month and year).");
   gField.focus();
   gField.select();
   return false;
}
```

Before getting too deep into the year validation, I convert any two-digit year within the specified range to its four-digit equivalent.

```
// validate year, allowing for checks between year ranges
// passed as parameters from other validation functions
if (yyyy < 100) {
   // entered value is two digits, which we allow for 1930-2029
   if (yyyy >= 30) {
      yyyy += 1900;
   } else {
      yyyy += 2000;
   }
}

var today = new Date();
```

I designed this function to work with a pair of year ranges or date ranges (so many days before and/or after today). If the function is passed date ranges, the first two parameters must be passed as `null`. This first batch of code works with the date ranges (because the `minYear` parameter is `null`).

```
if (!minYear) {
   // function called with specific day range parameters
   var dateStr = new String(monthDayFormat(mm) + "/" + monthDayFormat(dd) +
      "/" + yyyy);
   var testDate = new Date(dateStr);
   if (testDate.getTime() < (today.getTime() + (minDays * 24 * 60 * 60 *
      1000))) {
      alert("The most likely range for this entry begins " + minDays +
         " days from today.");
   }
   if (testDate.getTime() > today.getTime() + (maxDays * 24 * 60 * 60 *
      1000)) {
      alert("The most likely range for this entry ends " + maxDays +
         " days from today.");
   }
```

You can also pass hard-wired, four-digit years as parameters. The following branch compares the entered year against the range specified by those passed year values.

```
} else if (minYear && maxYear) {
   // function called with specific year range parameters
   if (yyyy < minYear || yyyy > maxYear) {
      // entered year is outside of range passed from calling function
      alert("The most likely range for this entry is between the years " +
         minYear + " and " + maxYear + ".  If your source data indicates a
         date outside this range, then enter that date.");
   }
} else {
```

For year parameters passed as positive or negative year differences, I begin processing by getting the four-digit year for today's date. Then I compare the entered year against the passed range values. If the entry is outside the desired range, an alert reveals the preferred year range within which the entry should fall. But the function does not return any value here because an out-of-range value is not critical for this application.

```
      // default year range (now set to (this year - 100) and (this year + 25))
      var thisYear = today.getYear();
      if (thisYear < 100) {
         thisYear += 1900;
      }
      if (yyyy < minYear || yyyy > maxYear) {
         alert("It is unusual for a date entry to be before " + minYear + " or
            after " + maxYear + ". Please verify this entry.");
      }
   }
```

One more important validation is to make sure that the entered date is valid for the month and year. Therefore, the various date components are passed to functions to check against month lengths, including the special calculations for the varying length of February. Listing 43-15 shows these functions. The alert messages they display are smart enough to inform the user what the maximum date is for the entered month and year.

```
   if (!checkMonthLength(mm,dd)) {
      gField.focus();
      gField.select();
      return false;
   }
   if (mm == 2) {
      if (!checkLeapMonth(mm,dd,yyyy)) {
         gField.focus();
         gField.select();
         return false;
      }
   }
```

The final task is to reassemble the date components into a format that the database wants for its date storage and stuff it into the form field. If the user enters an all-number or hyphen-delimited date, it is automatically reformatted and displayed as a slash-delimited, four-digit-year date.

```
   // put the Informix-friendly format back into the field
   gField.value = monthDayFormat(mm) + "/" + monthDayFormat(dd) + "/" + yyyy;
   return true;
}
```

A utility function invoked multiple times in the previous function converts a single-digit month or day number to a string that might have a leading zero:

```
// convert month or day number to string,
// padding with leading zero if needed
function monthDayFormat(val) {
   if (isNaN(val) || val == 0) {
      return "01";
   } else if (val < 10) {
      return "0" + val;
   }
   return "" + val;
}
```

Listing 43-15: **Functions to Check Month Lengths**

```
 // check the entered month for too high a value
function checkMonthLength(mm,dd) {
   var months = new Array("","January","February","March","April","May",
      "June","July","August","September","October","November","December");
   if ((mm == 4 || mm == 6 || mm == 9 || mm == 11) && dd > 30) {
      alert(months[mm] + " has only 30 days.");
      return false;
   } else if (dd > 31) {
      alert(months[mm] + " has only 31 days.");
      return false;
   }
   return true;
}

// check the entered February date for too high a value
function checkLeapMonth(mm,dd,yyyy) {
   if (yyyy % 4 > 0 && dd > 28) {
      alert("February of " + yyyy + " has only 28 days.");
      return false;
   } else if (dd > 29) {
      alert("February of " + yyyy + " has only 29 days.");
      return false;
   }
   return true;
}
```

This is a rather extensive date-validation routine, but it demonstrates how thorough you must be when a database relies on accurate entries. The more prompting and assistance you can give to users to ferret out problems with invalid entries, the happier those users will be.

Cross-confirmation fields

The final validation type that I cover here is probably not a common request, but it demonstrates how the dispatch mechanism created at the outset expands so easily to accommodate this enhanced client request. The situation is that some fields (mostly dates in this application) are deemed critical pieces of data because this data triggers other processes from the database. As a further check to ensure entry of accurate data, a number of values are set up for entry twice in separate fields — and the fields have to match exactly. In many ways, this mirrors the two passes you are often requested to make when you set a password: enter two copies and let the computer compare them to make sure you typed what you intended to type.

I established a system that places only one burden on the many programmers working on the forms: although you can name the primary field anything you want (to help alignment with database column names, for example), you must name the secondary field the same plus `"_xcfm"` — which stands for *cross-confirm*. Then, pass the `isConfirmed` validation name to the `validate()` function after the date range validation name, as follows:

```
onchange="parent.validate(window, this, 'isM5_P10Date','isConfirmed')"
```

In other words, after the entered value is initially checked against a required date range, the `isConfirmed()` validation function compares the fully vetted, properly formatted date in the current field against its parallel entry.

Listing 43-16 shows the one function in `validations.js` that handles the confirmation in both directions. After assigning a copy of the entry field value to the `inputStr` variable, the function next sets a Boolean flag (`primary`) that lets the rest of the script know if the entry field is the primary or secondary field. If the string `"_xcfm"` is missing from the field name, the entry field is the primary field.

For the primary field branch, the script assembles the name of the secondary field and compares the content of the secondary field's value against the `inputStr` value. If they are not the same, the user is entering a new value into the primary field, and the script empties the secondary field to force reentry to verify that the user enters the proper data.

For the secondary field entry branch, the script assembles a reference to the primary field by stripping away the final five characters of the secondary field's name. I can use the `lastIndexOf()` string method instead of the longer way involving the string's length; but after experiencing so many platform-specific problems with `lastIndexOf()` in Navigator, I decided to play it safe. Finally, the two values are compared, with an appropriate alert displayed if they don't match.

Listing 43-16: **Cross-Confirmation Validation**

```
 // checks an entry against a parallel, duplicate entry to
// confirm that correct data has been entered
// Parallel field name must be the main field name plus "_xcfm"
function isConfirmed() {
   var inputStr = gField.value;
   // flag for whether field under test is primary (true) or confirmation field
   var primary = (gField.name.indexOf("_xcfm") == -1);
   if (primary) {
      // clear the confirmation field if primary field is changed
      var xcfmField =
window.frames[gFrame.name].document.forms[0].elements[gField.name + "_ xcfm"];
      var xcfmValue = xcfmField.value;
      if (inputStr != xcfmValue) {
         xcfmField.value = "";
         return true;
      }
   } else {
      var xcfmField =
window.frames[gFrame.name].document.forms[0].elements[gField.name.substring(0,(g
Field.name.length-5))];
      var xcfmValue = xcfmField.value;
      if (inputStr != xcfmValue) {
         alert("The main and confirmation entry field contents do not match.
            Both fields must have EXACTLY the same content to be accepted by
            the database.");
         gField.focus();
         gField.select();
         return false;
      }
   }
   return true;
}
```

Last-minute check

Every validation event handler is designed to return `true` if the validation succeeds. This comes in handy for the batch validation that performs one final check of the entries triggered by the form's `onsubmit` event handler. This event handler calls a `checkForm()` function and passes the form control object as a parameter. That parameter helps create a reference to the form element that is passed to each validation function.

Because successful validations return `true`, you can nest consecutive validation tests so that the most nested statement of the construction is `return true` because all validations have succeeded. The form's `onsubmit` event handler is as follows:

```
onsubmit="return checkForm(this)"
```

And the following code fragment is an example of a `checkForm()` function. A separate `isDateFormat()` validation function called here checks whether the field contains an entry in the proper format — meaning that it has likely survived the range checking and format shifting of the real-time validation check.

```
function checkForm(form) {
   if (parent.validate(window, form.birthdate, "isDateFormat")) {
      if (parent.validate(window, form.phone, "isPhone")) {
         if (parent.validate(window, form.name, "isNotEmpty")) {
            return true;
         }
      }
   }
   return false;
}
```

If any one validation fails, the field is given focus and its content is selected (controlled by the individual validation function). In addition, the `checkForm()` function returns `false`. This, in turn, cancels the form submission.

Try it out

Listing 43-17 is a definition for a frameset that not only loads the validation routines described in this section, but also loads a page with a form that exercises the validations in real-time and batch mode just prior to submission. The form appears earlier in this chapter in Figure 43-1.

Listing 43-17: **Frameset for Trying validation.js**

```
<html>
  <head>
     <title>GiantCo Contractor Database</title>
     <script type="text/javascript" src="validation.js">
     </script>
     <script type="text/javascript">
     function blank() {
        return "<html><body bgcolor='lightsteelblue'><\/body><\/html>";
     }
```

Continued

Listing 43-17 *(continued)*

```
      </script>
   </head>
   <frameset frameborder="" cols="20%,80%">
      <frame name="toc" src="javascript:parent.blank()" />
      <frame name="entries" src="lst43-18.htm" />
   </frameset>
</html>
```

The application scenario for the form is the entry of data into a company's contractor database. Some fields are required, and the date field must be cross-confirmed with a second entry of the same data. If the form passes its final validation prior to submission, the form reloads and you see a readout of the form data that would have been submitted from the previous form had the `action` been set to a server CGI program URI.

Plan for Data Validation

I devoted this entire chapter to the subject of data validation because it represents the one area of error checking that almost all JavaScript authors should be concerned with. If your scripts (client-side or server-side) perform processing on user entries, you want to prevent script errors at all costs.

✦ ✦ ✦

CHAPTER 44

Scripting Java Applets and Plug-Ins

✦ ✦ ✦ ✦

In This Chapter

Communicating with Java applets from scripts

Accessing scripts and objects from Java applets

Controlling scriptable plug-ins

✦ ✦ ✦ ✦

Netscape was the first to implement the facility enabling JavaScript scripts, Java applets, and plug-ins to communicate with each other under one technology umbrella, called LiveConnect (first implemented in NN3). Microsoft met the challenge and implemented a large part of that technology for WinIE4, but of course without using the Netscape-trademarked name for the technology. The name is a convenient way to refer to the capability, so you find it used throughout this chapter applying to any browser that supports such facilities. This chapter focuses on the scripting side of LiveConnect: approaching applets and plug-ins from scripts and accessing scripts from Java applets.

Except for the part about talking to scripts from inside a Java applet, I don't assume you have any knowledge of Java programming. The primary goal here is to help you understand how to control applets and plug-ins (including ActiveX controls in WinIE) from your scripts. If you're in a position to develop specifications for applets, you also learn what to ask of your Java programmers. But if you are also a Java applet programmer, you learn the necessary skills to get your applets in touch with HTML pages and scripts.

LiveConnect Overview

Before you delve too deeply into the subject, you should be aware that LiveConnect features are not available in all modern browsers, much to the chagrin of many. The following browsers do not support this technology:

- MacIE
- NN4.6 (due to an oversight when the version was released)
- NN6.0 (although it is supported in NN7+)
- Safari 1.0

Such a broad swath of browsers not supporting the feature makes it difficult to design a public Web application that relies on LiveConnect features. Design your pages accordingly.

The internal mechanisms that allow scripts to communicate with applets and plug-ins are quite different for NN and IE. NN3 and NN4 relied exclusively on the Java virtual machine (JVM) that shipped with most OS platform versions of the browsers. In NN4+, the JVM doesn't load until it is needed, sometimes causing a brief delay in initial execution. For the most part, though, the underlying Java engine is invisible to the scripter (you) and certainly to the visitors of your sites. At most, visitors see status bar messages about applets loading and running.

WinIE, on the other hand, has its own internal architecture for communicating between processes. To Windows, most processes are treated as components that have properties and methods accessible to other components.

Whether you use the technology to communicate with a Java applet or an ActiveX control, the advantage to you as an author is that LiveConnect extends the document object model to include objects and data types that are not a part of the HTML world. HTML, for instance, does not have a form control element that displays real-time stock ticker data; nor does HTML have the capability to treat a sound file like anything more than a URL to be handed off to a helper application. With LiveConnect, however, your scripts can treat the applet that displays the stock ticker as an object whose properties and methods can be modified after the applet loads; scripts can also tell the sound when to play or pause by controlling the plug-in that manages the incoming sound file.

Why Control Java Applets?

A question I often hear from experienced Java programmers is, "Why bother controlling an applet via a script when you can build all the interactivity you want into the applet itself?" This question is valid if you come from the Java world, but it takes a viewpoint from the HTML and scripting world to fully answer it.

Java applets exist in their own private rectangles, remaining largely oblivious to the HTML surroundings on the page. Applet designers who don't have extensive Web page experience tend to regard their applets as the center of the universe rather than as components of HTML pages.

As a scripter, on the other hand, you may want to use those applets as powerful components to spiff up the overall presentation. Using applets as prewritten objects enables you to make simple changes to the HTML pages — including the geographic layout of elements and images — at the last minute, without having to rewrite and recompile Java program code. If you want to update the look with an entirely new graphical navigation or control bar, you can do it directly via HTML and scripting.

When it comes to designing or selecting applets for inclusion into my scripted page, I prefer using applet interfaces that confine themselves to data display, putting any control of the data into the hands of the script, rather than using onscreen buttons in the applet rectangle. I believe this setup enables much greater last-minute flexibility in the page design — not to mention consistency with HTML form element interfaces — than putting everything inside the applet rectangle.

A Little Java

If you plan to look at a Java applet's scripted capabilities, you can't escape having to know a little about Java applets and some terminology. The discussion goes more deeply into object orientation than you have seen with JavaScript, but I'll try to be gentle.

Java building blocks classes

One part of Java that closely resembles JavaScript is that Java programming deals with objects, much the way JavaScript deals with a page's objects. Java objects, however, are not the familiar HTML objects but rather more basic building blocks, such as tools that draw to the screen and data streams. But both languages also have some non-HTML kinds of objects in common: strings, arrays, numbers, and so on.

Every Java object is known as a *class* — a term from the object-orientation world. When you use a Java compiler to generate an applet, that applet is also a class, which happens to incorporate many Java classes, such as strings, image areas, font objects, and the like. The applet file you see on the disk is called a class file and its file extension is `.class`. This file is the one you specify for the `code` attribute of an `<applet>` tag, or the newer `<object>` tag (the `<applet>` tag is deprecated in HTML 4.0).

Java methods

Most JavaScript objects have methods attached to them that define what actions the objects are capable of performing. A string object, for instance, has the `toUpperCase()` method that converts the string to all uppercase letters. Java classes also have methods. Many methods are predefined in the base Java classes embedded inside the virtual machine. But inside a Java applet, the author can write methods that either override the base method or deal exclusively with a new class created in the program. These methods are, in a way, like the functions you write in JavaScript for a page.

Not all methods, however, are created the same. Java lets authors determine how visible a method is to outsiders. The types of methods that you, as a scripter, are interested in are the ones declared as public methods. You can access such methods from JavaScript via a syntax that falls very much in line with what you already know. For example, a common public method in applets stops an applet's main process. Such a Java method may look like this:

```
public void stop() {
   if(thread != null) {
      thread = null;
   }
}
```

The `void` keyword simply means that this method does not return any values (compilers need to know this stuff). Assuming that you have one applet loaded in your page, the JavaScript call to this applet method is

```
document.applets[0].stop();
```

Listing 44-1a shows how all this works with the <applet> tag for a scriptable digital clock applet example. The script includes calls to two of the applet's methods: to stop and to start the clock.

Listing 44-1a: **Stopping and Starting an Applet**

```
<html>
  <head>
     <title>A Script That Could Stop a Clock</title>
     <script type="text/javascript">
     function pauseClock() {
        document.clock1.stop();
     }
     function restartClock() {
        document.clock1.start();
     }
     </script>
  </head>
  <body>
     <h1>Simple control over an applet</h1>
     <hr />
     <applet code="ScriptableClock.class" name="clock1" width="500"
     height="45">
        <param name="bgColor" value="Green" />
        <param name="fgColor" value="Blue" />
     </applet>
     <form name="widgets1">
        <input type="button" value="Pause Clock" onclick="pauseClock()" />
        <input type="button" value="Restart Clock" onclick="restartClock()" />
     </form>
  </body>
</html>
```

The syntax for accessing the method (in the two functions) is just like JavaScript in that the references to the applet's methods include the applet object (clock1 in the example), which is contained by the document object.

XHTML-friendly applets

Because the XHTML standard does not recognize the <applet> tag, you need to work a little markup magic to use modern <object> tags that operate in both IE and Mozilla-based browsers. The difficulty arises in the way the two browsers specify some attribute or parameter values for Java applets. To demonstrate how you can embed an applet into a document exclusively with the <object> tag and work simultaneously in IE and Mozilla-based browsers, Listing 44-1b shows a modified version of Listing 44-1a.

An important feature of the listing is a Microsoft proprietary markup feature called *conditional comments* (msdn.microsoft.com/workshop/author/dhtml/overview/ccomment_ovw.asp). These HTML comment tags, with their special comment text, allow IE to skip over HTML markup. In Listing 44-1b, only the first <object> tag is rendered in WinIE; Mozilla, on

the other hand, loads both, but does not load the applet in the first tag because the attribute and parameter values are not in the format that Mozilla requires for loading a Java applet. In addition to the tag markup differences, note that the functions controlling the applet create references to the applet object differently. Mozilla has two `object` elements with the same name deal with, meaning that there is an array of objects with that name; thus the script pulls a reference to the second one when two are detected.

Listing 44-1b: **Stopping and Starting an Applet (XHTML)**

```
<html>
   <head>
      <title>A Script That Could Stop a Clock</title>
      <script type="text/javascript">
      function pauseClock() {
         // get ref to second object for non-WinIE
         var applet = (document.clock1.length) ? document.clock1[1] :
document.clock1;
         applet.stop();
      }
      function restartClock() {
         var applet = (document.clock1.length) ? document.clock1[1] :
document.clock1;
         applet.start();
      }
      </script>
   </head>
   <body>
      <h1>Simple control over an applet</h1>
      <hr />
      <object name="clock1" classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
         width="500" height="45">
         <param name="code" value="ScriptableClock.class" />
         <param name="codebase" value="." />
         <param name="bgColor" value="Green" />
         <param name="fgColor" value="Blue" />
         <!--[if !IE]> Non-WinIE Browsers -->
         <object name="clock1" classid="java:ScriptableClock.class"
codebase="."
            width="500" height="45">
            <param name="bgColor" value="Green" />
            <param name="fgColor" value="Blue" />
          </object>
          <!-- <![endif]-->
       </object>

      <form name="widgets1">
         <input type="button" value="Pause Clock" onclick="pauseClock()" />
         <input type="button" value="Restart Clock" onclick="restartClock()" />
      </form>
   </body>
</html>
```

Java applet "properties"

The Java equivalents of JavaScript object properties are called public instance variables. These variables are akin to JavaScript global variables. If you have access to some Java source code, you can recognize a public instance variable by its `public` keyword:

```
public String fontName;
```

Java authors must specify a variable's data type when declaring any variable. That's why the `String` data type appears in the preceding example.

Your scripts can access these variables with the same kind of syntax that you use to access JavaScript object properties. If the `fontName` variable in `ScriptableClock.class` had been defined as a public variable (it is not), you could access or set its value directly, as shown in the following example:

```
var theFont = document.applets[0].fontName;
document.applets[0].fontName = "Courier";
```

Accessing Java fields

In a bit of confusing lingo, public variables and methods are often referred to as *fields*. These elements are not the kind of text entry fields that you see on the screen; rather, they're like slots (another term used in Java) where you can slip in your requests and data. Remember these terms, because they may appear from time to time in error messages as you begin scripting applets.

Scripting Applets in Real Life

Because the purpose of scripting an applet is to gain access to the inner sanctum of a compiled program, the program should be designed to handle such rummaging around by scripters. If you can't acquire a copy of the source code or don't have any other documentation about the scriptable parts of the applet, you may have a difficult time knowing what to script and how to do it.

Getting to scriptable methods

If you write your own applets or are fortunate enough to have the source code for an existing applet, the safest way to modify the applet variable settings or the running processes is through applet methods. Although setting a public variable value may enable you to make a desired change, you don't know how that change may impact other parts of the applet. An applet designed for scriptability should have a number of methods defined that enable you to make scripted changes to variable values.

To view a sample of an applet designed for scriptability, open the Java source code file for Listing 44-2 from the CD-ROM. A portion of that program listing is shown in the following example.

Listing 44-2: **Partial Listing for ScriptableClock.java /***

```
      Begin public methods for getting
      and setting data via LiveConnect
*/
public void setTimeZone(String zone) {
   stop();
   timeZone = (zone.startsWith("GMT")) ? true : false;
   start();
}

public void setFont(String newFont, String newStyle, String newSize) {
   stop();
   if (newFont != null && newFont != "")
      fontName = newFont;
   if (newStyle != null && newStyle != "")
      setFontStyle(newStyle);
   if (newSize != null && newSize != "")
      setFontSize(newSize);
   displayFont = new Font(fontName, fontStyle, fontSize);
   start();
}

public void setColor(String newbgColor, String newfgColor) {
   stop();
   bgColor = parseColor(newbgColor);
   fgColor = parseColor(newfgColor);
   start();
}

public String getInfo() {
   String result = "Info about ScriptableClock.class\r\n";
   result += "Version/Date: 1.0d1/2 May 1996\r\n";
   result += "Author: Danny Goodman (dannyg@dannyg.com)\r\n";
   result += "Public Variables:\r\n";
   result += "   (None)\r\n\r\n";
   result += "Public Methods:\r\n";
   result += "   setTimeZone(\"GMT\" | \"Locale\")\r\n";
   result += "   setFont(\"fontName\",\"Plain\" |\"Bold\" | \"Italic\",
      \"fontSize\")\r\n";
   result += "   setColor(\"bgColorName\", \"fgColorName\")\r\n";
   result += "       colors: Black, White, Red, Green, Blue, Yellow\r\n";
   return result;
}
/*
   End public methods for scripted access.
*/
```

The methods shown in Listing 44-2 are defined specifically for scripted access. In this case, they safely stop the applet thread before changing any values. The last method is one I recommend to applet authors. The method returns a small bit of documentation containing information about the kind of methods that the applet likes to have scripted and what you can have as the passed parameter values.

Now that you see the amount of scriptable information in this applet, look at Listing 44-3, which takes advantage of that scriptability by providing several HTML form elements as user controls for the clock. The results are shown in Figure 44-1.

Listing 44-3: **A More Fully Scripted Clock**

```
<html>
   <head>
      <title>Clock with Lots o' Widgets</title>
      <script type="text/javascript">
      function setTimeZone(popup) {
         var choice = popup.options[popup.selectedIndex].value;
         document.clock2.setTimeZone(choice);
      }

      function setColor(form) {
         var bg = form.backgroundColor.options[
            form.backgroundColor.selectedIndex].value;
         var fg = form.foregroundColor.options[
            form.foregroundColor.selectedIndex].value;
         document.clock2.setColor(bg, fg);
      }

      function setFont(form) {
         var fontName = form.theFont.options[form.theFont.selectedIndex].value;
         var fontStyle = form.theStyle.options[
            form.theStyle.selectedIndex].value;
         var fontSize = form.theSize.options[form.theSize.selectedIndex].value;
         document.clock2.setFont(fontName, fontStyle, fontSize);
      }
      function getAppletInfo(form) {
         form.details.value = document.clock2.getInfo();
      }

      function showSource() {
         var newWindow = window.open("ScriptableClock.java","",
            "width=450,height=300,resizable,scrollbars");
      }
      </script>
   </head>
   <body>
      <applet code="ScriptableClock.class" name="clock2" width="500"
      height="45">
         <param name="bgColor" value="Black" />
         <param name="fgColor" value="Red" />
      </applet>
      <form name="widgets2">
```

```
         Select Time Zone: <select name="zone" onchange="setTimeZone(this)">
            <option selected="selected" value="Locale">Local Time</option>
            <option value="GMT">Greenwich Mean Time</option>
         </select>
         <p>Select Background Color: <select name="backgroundColor"
            onchange="setColor(this.form)">
               <option value="White">White</option>
               <option selected="selected" value="Black">Black</option>
               <option value="Red">Red</option>
               <option value="Green">Green</option>
               <option value="Blue">Blue</option>
               <option value="Yellow">Yellow</option>
            </select> Select Color Text Color: <select name="foregroundColor"
            onchange="setColor(this.form)">
               <option value="White">White</option>
               <option value="Black">Black</option>
               <option selected="selected" value="Red">Red</option>
               <option value="Green">Green</option>
               <option value="Blue">Blue</option>
               <option value="Yellow">Yellow</option>
            </select></p>
         <p>Select Font: <select name="theFont" onchange="setFont(this.form)">
               <option selected="selected" value="TimesRoman">Times
                  Roman</option>
               <option value="Helvetica">Helvetica</option>
               <option value="Courier">Courier</option>
               <option value="Arial">Arial</option>
            </select><br />
            Select Font Style: <select name="theStyle"
            onchange="setFont(this.form)">
               <option selected="selected" value="Plain">Plain</option>
               <option value="Bold">Bold</option>
               <option value="Italic">Italic</option>
            </select><br />
            Select Font Size: <select name="theSize"
            onchange="setFont(this.form)">
               <option value="12">12</option>
               <option value="18">18</option>
               <option selected="selected" value="24">24</option>
               <option value="30">30</option>
            </select></p>
         <hr />
         <input type="button" name="getInfo" value="Applet Info"
         onclick="getAppletInfo(this.form)" />
         <p><textarea name="details" rows="11" cols="70">
            </textarea></p>
      </form>
      <hr />
   </body>
</html>
```

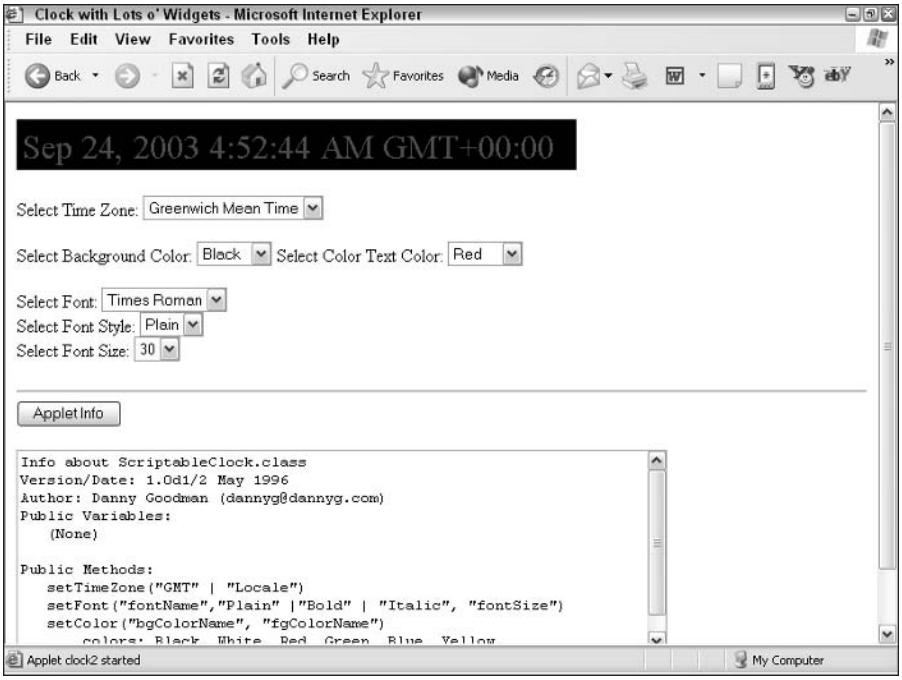


Figure 44-1: Scripting more of the ScriptableClock applet.

Very little of the code here controls the applet — only the handful of functions near the top. The rest of the code makes up the HTML user interface for the form element controls. After you open this document from the CD-ROM, be sure to click the Applet Info button to see the methods that you can script and the way that the parameter values from the JavaScript side match up with the parameters on the Java method side.

Applet limitations

Because of concerns about security breaches via LiveConnect, Netscape clamps down on some powers that would be nice to have via a scripted applet. The most noticeable barrier is the one that prevents applets from accessing the network under scripted control. Therefore, even though a Java applet has no difficulty reading or writing text files from the server, such capabilities — even if built into an applet of your own design — won't be carried out if triggered by a JavaScript call to the applet.

Some clever hacks used to be posted on the Web, but they were rather cumbersome to implement and may no longer work on more modern browsers. You can also program the Java applet to fetch a text file after it starts up and then script the access of that value from JavaScript (as described in the following section). Signed scripts (see Chapter 46) and applets can break through these security barriers after the user has given explicit permission to do so.

Faceless applets

Until LiveConnect came along, Java applets were generally written to show off data and graphics — to play a big role in the presentation on the page. But if you prefer to let an applet do the heavy algorithmic lifting for your pages while the HTML form elements and images (or Dynamic HTML facilities of newer browsers) do the user interface, you essentially need what I call a *faceless applet*.

The method for embedding a faceless applet into your page is the same as embedding any applet: Use the `<applet>` and/or `<object>` tag; the `<object>` tag is the recommended approach for modern browsers, but the `<applet>` tag is better supported in early Java-powered browsers. For a faceless applet, specify only 1 pixel for both the `height` and `width` attributes (0 has strange side effects). This setting creates a dot on the screen, which, depending on your page's background color, may be completely invisible to page visitors. Place it at the bottom of the page, if you like.

To show how nicely this method can work, Listing 44-4 provides the Java source code for a simple applet that retrieves a specific text file and stores the results in a Java variable available for fetching by the JavaScript shown in Listing 44-5. The HTML even automates the loading process by triggering the retrieval of the Java applet's data from an `onload` event handler.

Listing 44-4: **Java Applet Source Code**

```
import java.net.*;
import java.io.*;

public class FileReader extends java.applet.Applet implements Runnable {

   Thread thread;
   URL url;
   String output;
   String fileName = "Bill of rights.txt";

   public void getFile(String fileName) throws IOException {
      String result, line;
      InputStream connection;
      DataInputStream dataStream;
      StringBuffer buffer = new StringBuffer();

      try {
         url = new URL(getDocumentBase(),fileName);
      }
      catch (MalformedURLException e) {
         output = "AppletError " + e;
      }

      try {
         connection = url.openStream();
         dataStream = new DataInputStream(new BufferedInputStream(connection));
```

Continued

Listing 44-4 *(continued)*

```
            while ((line = dataStream.readLine()) != null) {
               buffer.append(line + "\n");
            }
            result = buffer.toString();
         }
         catch (IOException e) {
            result = "AppletError: " + e;
         }
         output = result;
      }
public String fetchText() {
         return output;
      }

      public void init() {
      }

      public void start() {
         if (thread == null) {
            thread = new Thread(this);
            thread.start();
         }
      }
      public void stop() {
         if (thread != null) {
            thread = null;
         }
      }

      public void run(){
         try {
            getFile(fileName);
         }
         catch (IOException e) {
            output = "AppletError: " + e;
         }
      }
}
```

All the work of actually retrieving the file is performed in the `getFile()` method (which runs immediately after the applet loads). Notice that the name of the file to be retrieved, `Bill of Rights.txt`, is stored as a variable near the top of the code, making it easy to change for a recompilation, if necessary. You can also modify the applet to accept the filename as an applet parameter, specified in the HTML code. Meanwhile, the only hook that JavaScript needs is the one public method called `fetchText()`, which merely returns the value of the output variable, which in turn holds the file's contents.

This Java source code must be compiled into a Java class file (already compiled and included on the CD-ROM as `FileReader.class`) and placed in the same directory as the HTML file that loads this applet. Also, no explicit pathname for the text file is supplied in the source code, so the text file is assumed to be in the same directory as the applet.

Listing 44-5: **HTML Asking Applet to Read Text File**

```
<html>
   <head>
      <title>Letting an Applet Do The Work</title>
      <script type="text/javascript">
      function getFile(form) {
         var output = document.readerApplet.fetchText();
         form.fileOutput.value = output;
      }
      function autoFetch() {
         var output = document.readerApplet.fetchText();
         if (output != null) {
            document.forms[0].fileOutput.value = output;
            return;
         }
         var t = setTimeout("autoFetch()",1000);
      }
      </script>
   </head>
   <body onload="autoFetch()">
      <h1>Text from a text file...</h1>
      <form name="reader">
         <input type="button" value="Get File" onclick="getFile(this.form)" />
         <p><textarea name="fileOutput" rows="10" cols="60" wrap="hard">
            </textarea></p>
         <p><input type="Reset" value="Clear" /></p>
      </form>
      <applet code="FileReader.class" name="readerApplet" width="1" height="1">
      </applet>
   </body>
</html>
```

Because an applet is usually the last detail to finish loading in a document, you can't use an applet to generate the page immediately. At best, an HTML document can display a pleasant welcome screen while the applet finishes loading itself and running whatever it does to prepare data for the page's form elements. In IE4+ and NN6+/MoZ, the page can then be dynamically constructed out of the retrieved data; for NN4, you can create a new layer object, and use `document.write()` to install content into that layer. Notice in Listing 44-5 that the `onload` event handler calls a function that checks whether the applet has supplied the requested data. If not, the same function is called repeatedly in a timer loop until the data is ready and the textarea can be set. The `<applet>` tag is located at the bottom of the Body, set to 1 pixel square — invisible to the user. No user interface exists for this applet, so you have no need to clutter up the page with any placeholder or bumper sticker.

Figure 44-2 shows the page generated by the HTML and applet working together. The Get File button is merely a manual demonstration of calling the same applet method that the `onload` event handler calls.

A faceless applet may be one way for Web authors to hide what may otherwise be JavaScript code that is open to any visitor's view. For example, if you want to deliver a small data collection lookup with a document, but don't want the array of data to be visible in the JavaScript code, you can create the array and lookup functionality inside a faceless applet. Then use

form controls and JavaScript to act as query entry and output display devices (or dynamically generate a table in IE4+ and W3C DOM browsers). Because the parameter values passed between JavaScript and Java applets must be string, numeric, or Boolean values, you won't be able to pass arrays without performing some amount of conversion either within the applet or the JavaScript code (JavaScript's *string*`.split()` and *array*`.join()` methods help a great deal here).

Data type conversions

The example in this chapter does not pass any parameters to the applet's methods, but you are free to do so. You need to pay attention to the way in which values are converted to Java data types. JavaScript strings and Boolean values are converted to Java String and Boolean objects. All JavaScript numbers, regardless of their subtype (that is, integer or floating-point number), are converted to `Float` objects. Therefore, if a method must accept a numeric parameter from a script, the parameter variable in the Java method must be defined as a `Float` type.

The distinction between JavaScript string values and string objects can impact data being passed to an applet. If an applet method requires a string object as a parameter, you may have to explicitly convert a JavaScript string value (for example, a string from a text field) to a string object via the `new String()` constructor (see Chapter 27).

You can also pass references to objects, such as form control elements. Such objects get wrapped with a `JSObject` type (see discussion about this class later in the chapter). Therefore, parameter variables must be established as type `JSObject` (and the `netscape.javascript.JSObject` class must be imported into the applet).

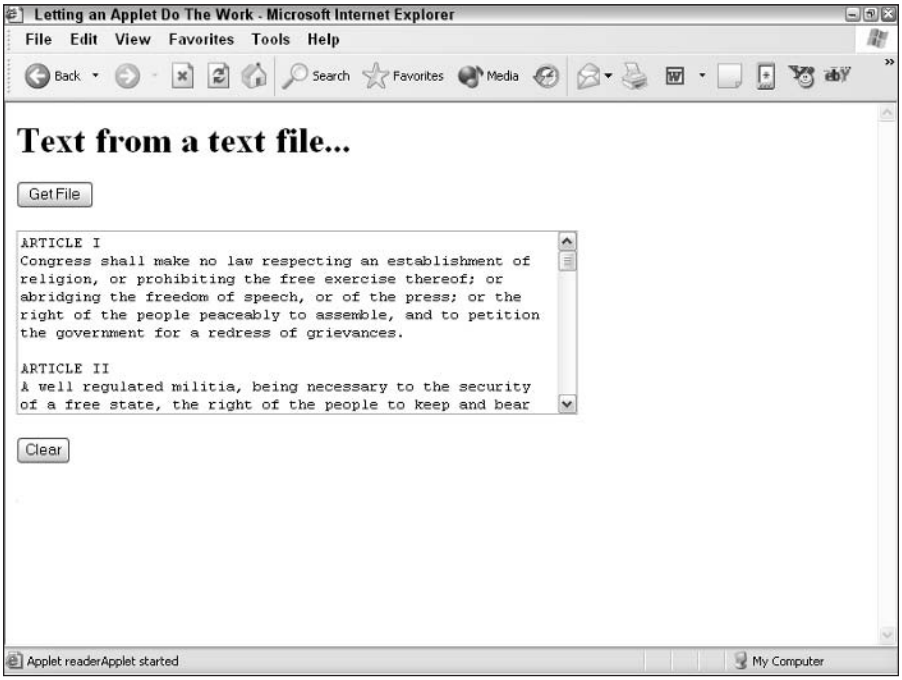


Figure 44-2: The page with text retrieved from a server file.

Applet-to-Script Communication

The flip side of scripted applet control is having an applet control script and HTML content in the page. Before you undertake this avenue in page design, you must bear in mind that any calls made from the applet to the page are hard-wired for the specific scripts and HTML elements in the page. If this level of tight integration and dependence suits the application, the link up will be successful.

The discussion of applet-to-script communication assumes you have experience writing Java applets. I use Java jargon quite freely in this discussion.

What your applet needs

NN3 and later (including Mozilla) come with a zipped set of special class files tailored for use in LiveConnect. In NN3, the file is named `java_30` or `java_301`, the latter one being the latest version; in NN4, the file is named `java40.jar`. For NN6+/Moz, the class files are located in an archive called `jaws.jar` (Windows) or `MRJPlugin.jar` (Mac). Use the file search facility of the OS to locate the relevant file on your system. Microsoft versions of these class files are also included in IE4+, buried in one of the large `.zip` files in the Windows\Java\Packages directory (the files you need are in one of the multi-megabyte `.zip` files, whose gibberish names change from version to version — open each with an unzip utility and look for the two packages mentioned next). The browser must see these class files (and have both Java and JavaScript enabled in the preferences screens) for LiveConnect to work.

The easiest way to access the ZIP file for LiveConnect classes is to install the Java SDK, which includes the `jaws.jar` file in its runtime `lib` directory. You then need to add the file to the Java `classpath` environment variable. Following is an example of how this is accomplished at the command line:

```
set classpath=c:\j2sdk1.4.1_02\jre\lib\jaws.jar
```

Of course, your specific Java SDK installation may be different in terms of version numbering, but the command should be very similar. With the `jaws.jar` file available in the `classpath`, you're ready to use LiveConnect objects in Java code and build an applet.

Following are the two vital classes in the `netscape` package (yes, even in IE), which is the Java package made available in the LiveConnect ZIP file (`jaws.jar`):

```
netscape.javascript.JSObject
netscape.javascript.JSException
```

Both classes must be imported to your applet via the Java `import` compiler directive:

```
import netscape.javascript.*;
```

When the applet runs, the LiveConnect-aware browser knows how to find the two classes, so that the user doesn't have to do anything special as long as the supporting files are in their default locations.

What your HTML needs

As a security precaution, an `<applet>` tag requires one extra attribute to give the applet permission to access the HTML and scripting inside the document. That attribute is the single word `mayscript`, and it can go anywhere inside the `<applet>` tag, as follows:

```
<applet code="myApplet.class" height="200" width="300" mayscript="mayscript" />
```

If you are using the <object> tag, add the mayscript feature as a parameter:

```
<param name="mayscript" value="true" />
```

Permission is not required for JavaScript to access an applet's methods or properties, but if the applet initiates contact with the page, this attribute is required.

About JSObject class

The portal between the applet and the HTML page that contains it is the netscape.javascript.JSObject class. This object's methods let the applet contact document objects and invoke JavaScript statements. Table 44-1 shows the object's methods and one static method.

Table 44-1: JSObject Class Methods

Method	*Description*
call(String functionName, Object args[])	Invokes JavaScript function, argument(s) passed as an array
eval(String expression)	Invokes a JavaScript statement
getMember(String elementName)	Retrieves a named object belonging to a container
getSlot(Int index)	Retrieves indexed object belonging to a container
getWindow(Applet applet)	Static method retrieves applet's containing window
removeMember(String elementName)	Removes a named object belonging to a container
setMember(String elementName, Object value)	Sets value of a named object belonging to a container
setSlot(int index, Object value)	Sets value of an indexed object belonging to a container
toString()	Returns string version of JSObject

Just as the window object is the top of the document object hierarchy for JavaScript references, the window object is the gateway between the applet code and the scripts and document objects. To open that gateway, use the JSObject.getWindow() method to retrieve a reference to the document window. Assign that object to a variable that you can use throughout your applet code. The following code fragment shows the start of an applet that assigns the window reference to a variable named mainwin:

```
import netscape.javascript.*;

public class myClass extends java.applet.Applet {
   private JSObject mainwin;

   public void init() {
      mainwin = JSObject.getWindow(this);
   }
}
```

If your applet will be making frequent trips to a particular object, you may want to create a variable holding a reference to that object. To accomplish this, the applet needs to make progressively deeper calls into the document object hierarchy with the `getMember()` method. For example, the following sequence assumes `mainwin` is a reference to the applet's document window. Eventually the statements set a form's field object to a variable for use elsewhere in the applet:

```
JSObject doc = (JSObject) mainwin.getMember("document");
JSObject form = (JSObject) doc.getMember("entryForm");
JSObject phonefld = (JSObject) form.getMember("phone");
```

Another option is to use the Java `eval()` method to execute an expression from the point of view of any object. For example, the following statement gets the same field object from the preceding fragment:

```
JSObject phonefld = mainwin.eval("document.entryForm.phone");
```

As soon as you have a reference to an object, you can access its properties via the `getMember()` method, as shown in the following example, which reads the `value` property of the text box, and casts the value into a Java `String` object:

```
String phoneNum = (String) phonefld.getMember("value");
```

Two `JSObject` class methods let your applet execute arbitrary JavaScript expressions and invoke object methods: the `eval()` and `call()` methods. Use these methods with any `JSObject`. If a value is to be returned from the executed statement, you must cast the result into the desired object type. The parameter for the `eval()` method is a string of the expression to be evaluated by JavaScript. Scope of the expression depends on the object attached to the `eval()` method. If you use the `window` object, the expression would exist as if it were a statement in the document script (not defined inside a function).

Using the `call()` method is convenient for invoking JavaScript functions in the document, although it requires a little more preparation. The first parameter is a string of the function name. The second parameter is an array of arguments for the function. Parameters can be of mixed data types, in which case the array would be of type `Object`. If you don't need to pass a parameter to the function call, you can define an array of a single empty string value (for example, `String arg[] = {""}`) and pass that array as the second parameter.

Data type conversions

The strongly typed Java language is a mismatch for loosely typed JavaScript. As a result, with the exception of Boolean and string objects (which are converted to their respective JavaScript objects), you should be aware of the way LiveConnect adapts data types to JavaScript.

Any Java object that contains numeric data is converted to a JavaScript number value. Because JavaScript numbers are `ieee` doubles, they can accommodate just about everything Java can throw its way.

If the applet extracts an object from the document and then passes that `JSObject` type back to JavaScript, that passed object is converted to its original JavaScript object type. But objects of other classes are passed as their native objects wrapped in JavaScript "clothing." JavaScript can access the applet object's methods and properties as if the object were a JavaScript object. Finally, Java arrays are converted to the same kind of JavaScript array created via the `new Array()` constructor. Elements can be accessed by integer index values (not named index values). All other JavaScript array properties and methods apply to this object as well.

Example applet-to-script application

To demonstrate several techniques for communicating from an applet to both JavaScript scripts and document objects, I present an applet that displays two simple buttons (see Figure 44-3). One button generates a new window, spawned from the main window, filling the window with dynamically generated content from the applet. The second button communicates from the applet to that second window by invoking a JavaScript function in the document. One last part of the demonstration shows the applet changing the value of a text box when the applet starts up.

Listing 44-6 shows the source code for the Java applet.

Because the applet generates two buttons, the code begins by importing the `AWT` interface builder classes. I also import the `netscape.javascript` package to get the `JSObject` class. The name of this sample class is `JtoJSDemo`. I declare four global variables: two for the windows, two for the applet button objects.

Listing 44-6: **Java Applet Source Code**

```
import java.awt.*;
import java.awt.event.*;
import netscape.javascript.*;

public class JtoJSDemo extends java.applet.Applet implements ActionListener {
   private JSObject mainwin, subwin;
   private Button newWinButton, toggleButton;
```

The applet's `init()` method establishes the user interface elements for this simple applet. A white background is matched in the HTML with a white document background color, making the applet appear to blend in with the page. I use this opportunity to set the `mainwin` variable to the browser window that contains the applet.

```
   public void init() {
      setBackground(Color.white);
      newWinButton = new Button("New Browser Window");
      toggleButton = new Button("Toggle SubWindow Color");
      this.add(newWinButton);
      this.add(toggleButton);
      newWinButton.addActionListener(this);
      toggleButton.addActionListener(this);
      mainwin = JSObject.getWindow(this);
   }
```

As soon as the applet starts, it changes the `value` property of a text box in the HTML form. Because this is a one-time access to the field, I elected to use the `eval()` method from the point of view of the main window, rather than build successive object references through the object hierarchy with the `getMember()` method.

```
   public void start() {
      mainwin.eval("document.indicator.running.value = 'Yes'");
   }
```

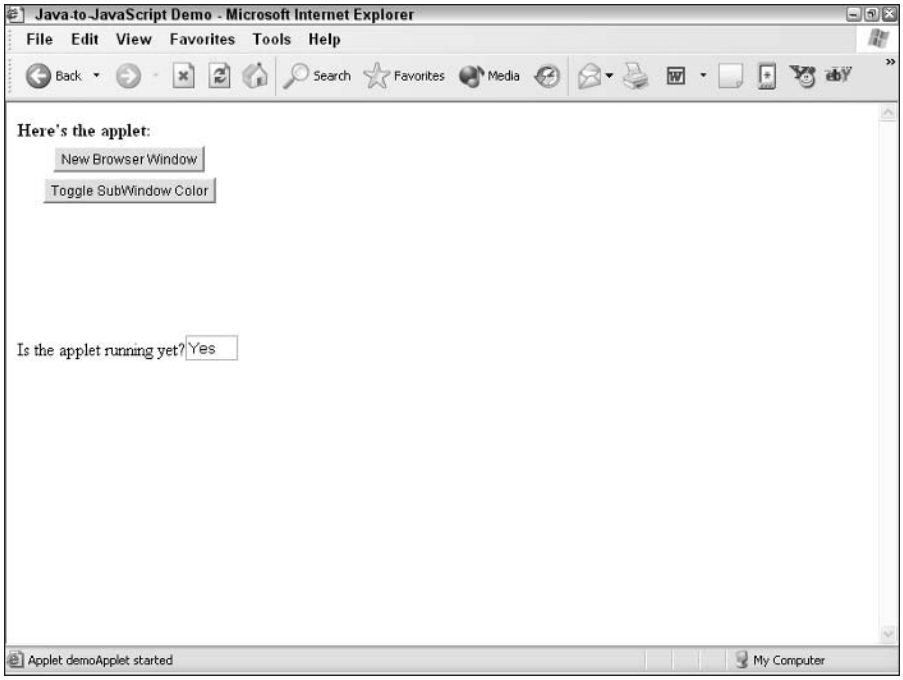


Figure 44-3: The applet displays two buttons seamlessly on the page.

Event handling is quite simple in this application. A click of the first button invokes `doNewWindow()`; a click of the second invokes `toggleColor()`. Both methods are defined later in the applet.

```
public void actionPerformed(ActionEvent evt) {
   Button source = (Button)evt.getSource();
   if (source == newWinButton) {
      doNewWindow();
   } else if (source == toggleButton) {
      toggleColor();
   }
}
```

One of the applet's buttons calls the `doNewWindow()` method defined here. I use the `eval()` method to invoke the JavaScript `window.open()` method. The string parameter of the `eval()` method is exactly like the statement that appears in the page's JavaScript to open a new window. The `window.open()` method returns a reference to that subwindow, so that the statement here captures the returned value, casting it as a `JSObject` type for the `subwin` variable. That `subwin` variable can then be used as a reference for another `eval()` method that writes to that second window. Notice that the object to the left of the `eval()` method governs the recipient of the `eval()` method's expression. The same is true for closing the writing stream to the subwindow.

Unfortunately, the IE4+ implementation of `JSObject` does not provide a suitable reference to the external window after it is created. Therefore, the window does not receive its content or respond to color changes in this example. Due to other anomalies with subwindows, I advise against using LiveConnect powers with multiple windows in IE4+.

Listing 44-6 (continued): **Java Applet Source Code**

```
   void doNewWindow() {
      subwin = (JSObject) mainwin.eval(
         "window.open('','fromApplet','height=200,width=200')");
      subwin.eval("document.write('<html><body bgcolor=white>Howdy from the
         applet!</body></html>')");
      subwin.eval("document.close()");
   }
```

The second button in the applet calls the `toggleColor()` method. In the HTML document, a JavaScript function named `toggleSubWindowColor()` takes a `window` object reference as an argument. Therefore, I first assemble a one-element array of type `JSObject` consisting of the `subwin` object. That array is the second parameter of the `call()` method, following a string version of the JavaScript function name being called.

```
   void toggleColor() {
      if (subwin != null) {
         JSObject arg[] = {subwin};
         mainwin.call("toggleSubWindowColor", arg);
      }
   }
}
```

Now onto the HTML that loads the preceding applet class and is the recipient of its calls. The document is shown in Listing 44-7. One function is called by the applet. A text box in the form is initially set to "No" but gets changed to "Yes" by the applet after it has finished its initialization. The only other item of note is that the `<applet>` tag includes a `mayscript` attribute to allow the applet to communicate with the page.

Listing 44-7: **HTML Document Called by Applet**

```
<html>
   <head>
      <title>Java-to-JavaScript Demo</title>
      <script type="text/javascript">
      function toggleSubWindowColor(wind) {
         if (wind.closed) {
            alert("The subwindow is closed.  Can't change it's color.");
         } else {
            wind.document.bgColor = (wind.document.bgColor == "#ffffff") ?
               "red" : "white";
         }
      }
      </script>
   </head>
```

```
   <body bgcolor="#FFFFFF">
      <b>Here's the applet:</b><br />
      <applet code="JtoJSDemo.class" name="demoApplet" height="150" width="200"
      mayscript="mayscript">
      </applet>
      <form name="indicator">
         Is the applet running yet?<input type="text" name="running" size="4"
         value="No" />
      </form>
   </body>
</html>
```

Scripting Plug-Ins

Controlling a plug-in (or Windows ActiveX control in IE) from JavaScript is much like controlling a Java applet. But you have more browser-specific concerns to worry about, even at the HTML level. Not all plug-ins are scriptable, of course, nor do all browsers permit such scripting, as described at the start of this chapter. Yet even when you have found the right combination of browser version(s) and plug-in(s), you must also learn what the properties and/or methods of the plug-in are so that your scripts can control them. For common plug-in duties, such as playing audio, the likelihood that all users will have the same audio playback plug-in installed in a particular browser brand and operating system is perhaps too small to entrust your programming to a single plug-in. If, on the other hand, you are using a plug-in that works only with a special data type, your page need check only that the plug-in is installed (and that it is the desired minimum version).

In this section of the chapter, you'll begin to understand the HTML issues and then examine two separate audio playback examples. One example lets users change tunes being played back; the other arrives with five sounds, each of which is controlled by a different onscreen interface element. Both of these audio playback examples employ a library that has been designed to provide basic audio playback interfaces to the most popular scriptable audio playback plug-in, Windows Media Player.

The main goal of the library is to act as an API (Application Programming Interface) between your scripts and the player. The API presents a simple vocabulary to let your scripts control the Windows Media Player. If you wish to control only a more modern version of the player (version 9 or later), you can modify the details for that player's more complex syntax in the API, while leaving your other interface code untouched.

The HTML side

Depending on the browser, operating system, and plug-in technology that you're using, one of two tags can be used to put a plug-in's powers into the page. With the plug-in embedded within the page (even if you don't see it), the plug-in becomes part of the document's object model, which means that your scripts can address it.

Using embed

The old way of embedding non-document content into a page was to use the `<embed>` tag. Even though the W3C HTML standard has never recognized the `embed` element, it has been a part of browser implementations since the first embeddable media. The element is also a bit of a chameleon, because beyond a common set of recognized attributes, such as the `src` attribute

that points to the content file to be loaded into the plug-in, its attributes are extensible to include items that apply only to a given plug-in. Uncovering the precise lists of attributes and values for a plug-in is not always easy, and frequently requires digging deeply into the developer documentation of the plug-in's producer. It is not unusual for a page author to anticipate that multiple plug-ins could play a particular kind of data (as is the case in the audio examples later in this chapter). Therefore, a single `embed` element may include attributes that apply to more than one plug-in. You have to hope that the plug-ins' developers chose unique names for their attributes or that like-named attributes mean the same thing in multiple plug-ins. Any attributes that a plug-in doesn't recognize are ignored.

Typical behavior for a plug-in is to display some kind of controller or other panel in a rectangle associated with the media. You definitely need to specify the `height` and `width` attribute values of such an `embed` element if it is to display visual media (some video plug-ins let you hide the controls, while still showing the viewing area). For audio, however, you can specify a 1-pixel value for both dimensions, and leave the controls to your HTML content. Browsers that recognize stylesheets can also set `embed` elements to be invisible.

As an example of what an `embed` element may look like, the following code could be used in Netscape 4 to load a QuickTime or LiveAudio player into a page (with the goal of using scripts to control the player):

```
<embed name="jukebox"
   height="1"
   width="1"
   src="Beethoven.aif"
   hidden="true"
   autostart="false"
   autoplay="false"
   enablejavascript="true"
   mastersound="mastersound">
</embed>
```

After the page loads and encounters this tag, the browser reaches out to the server and loads the sound file into the plug-in, where it sits quietly until the plug-in is instructed to play it.

WinIE/W3C object

The more modern and now preferred way to get external media into the document is to load the plug-in as an object via the `<object>` tag. The `object` element is endorsed by the W3C HTML standard, and has been supported in IE since IE4. In many ways the `<object>` tag works like the `<applet>` tag in that aside from specifying attributes that load the plug-in, additional nested `param` elements let you make numerous settings to the plug-in while it loads, including the name of the file to pre-load. As with a plug-in's attributes, an object's parameters are unique to the object and are documented (somewhere) for every object intended to be put into an HTML page.

The Windows operating system has a special (that is, far from intuitive) way it refers to the plug-in program (ActiveX control): through its class `id` (also known as a `guid`). You must know this long string of numbers and letters in order to embed the object into your page.

The following example is an `object` element that loads the Windows Media Player 9.*x* plug-in (ActiveX control) into a page:

```
<object id="jukebox" width="1" height="1"
   classid="CLSID:6BF52A52-394A-11d3-B153-00C04F79FAA6"
   <param name="FileName" value="Beethoven.aif" />
   <param name="AutoStart" value="false" />
</object>
```

When you compare the `embed` and `object` approaches, you can see many similar properties and values, which are just expressed differently (for example, attributes versus `param` elements).

Using embed and object together

Because a public Web page must usually appeal to a broad range of browsers, you should design such a page to work with as many browsers as possible. For the convenience of your scripting, referring to a plug-in object by the same identifier is helpful, whether it is loaded via an `embed` or `object` element.

To the rescue comes a handy behavior of the `object` element. It is designed in such a way that you can nest the associated `embed` element inside the `object` element's tag set. If the browser doesn't know about the `object` element, that element is ignored, but the `embed` element is picked up. Similarly, if the browser that knows about the `object` element fails to load the plug-in identified in its attributes, the nested `embed` elements also get picked up. Therefore, you can combine the `object` and `embed` elements as shown in the following example, which demonstrates loading a movie file into a QuickTime plug-in:

```
<object classid="clsid:02BF25D5-8C17-4B23-BC80-D3488ABDDC6B"
        codebase="http://www.apple.com/qtactivex/qtplugin.cab"
        width="180" height="160" id="clip1" >
   <param name="src" value="videoClip.mov" />
   <embed width="180" height="160" src="videoClip.mov"
      name="clip1" enablejavascript="true">
   </embed>
</object>
```

Notice that the identifier assigned to the `id` of the `object` element and to the `name` of the `embed` element are the same. In NN3 and NN4 only one of these two elements will be valid in the document, but because Mozilla browsers recognize both types of elements, you'll have to reference only one of the identically named elements (as described with Listing 44-1b earlier in this chapter).

The API approach

In this section, you see one version of an API that is tailor-made for the Windows Media Player. The API has its own initialization routine, which alerts users of ill-equipped browsers with a relevant message about why their browser can't get the most out of the page.

This API is far from the be-all, end-all library, although you will see that it does quite a bit as-is. The code is offered as a starting point for your further development. Such development may take the shape of adding more operations to the API or adding capabilities for additional scriptable plug-ins. For example, although the API as shown supports the older Windows Media Player 6, Microsoft continues to upgrade the Player to new versions (with new `guid`s for your `object` tags) that have new command vocabularies. There is no reason that the API cannot be extended for new generations of Windows Media Player, while maintaining backward compatibility for the Version 6 generation.

You can find the complete API code on the CD-ROM within the folder of example listings for this chapter. The API file is named `MPAudioAPI.js`. Check out the following high points of this library.

Loading the library

Adding the library to your page is no different from any external `.js` library file. Include the following tag in the `head` of your page:

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

Except for two global variable initializations, no immediate code runs from the library. All of its activity is invoked from event handlers or other script statements in the main page.

Initializing the library

The first job for the library is to validate that the player has loaded successfully. Before the library can do this, all loading of the `object` or `embed` elements must be concluded so that the objects exist for the initialization routine to examine. Therefore, use the `onload` event handler in the `body` to invoke the `initAudioAPI()` function. Parameters to be passed to this function are IDs of the `object` elements that represent individual players (in case there are more than one).

The following is an excerpt from Listing 44-9, which shows how the `jukebox` player object is initialized (all sound files for examples in this chapter have the `.aif` filename extension):

```
onload="initAudioAPI('jukebox')"
```

As you see later in Listing 44-10, the `initAudioAPI()` function lets you initialize multiple player objects. Each object has its own ID. For example, the following initializes the library for two different embedded plug-in objects:

```
onload="initAudioAPI('cNatural','cSharp')"
```

When the function receives multiple arguments, it loops through them, performing the initializations in sequence. The `initAudioAPI()` function follows:

```
function initAudioAPI() {
   var args = initAudioAPI.arguments;
   var id;
   for (var i = 0; i < args.length; i++) {
      // don't init any more if browser lacks scriptable sound
      if (OKToTest) {
         id = args[i];
         players[id] = new API(id);
         validateSupport(id);
      }
   }
}
```

Notice that parameter variables are not explicitly declared for the function, but are, instead, retrieved via the `arguments` property of the function. The global `OKToTest` flag, initialized to true when the library loads, is set to `false` if the validation of a plug-in fails. The conditional construction here prevents multiple alerts from appearing when multiple plug-in parameters are passed to the initialization function.

Sound player API objects

One of the jobs of the initialization routine is to create a player object for each plug-in identifier. The object's constructor is as follows:

```
// AudioAPI object constructor
function API(id) {
   this.id = id;
   this.play = API_play;
   this.stop = API_stop;
   this.pause = API_pause;
   this.rewind = API_rewind;
   this.load = API_load;
   this.getVolume = API_getVolume;
   this.setVolume = API_setVolume;
}
```

The object becomes a convenient place to preserve properties for each sound controller. But the bulk of the object is reserved for assigning methods — the methods that your main page's scripts invoke to play and stop the player, adjust its volume, and so on. The method names to the left of the assignment statements in the object constructor are the names your scripts use; the functions in the library (for example, `API_play()`) are the ones that send the right command to the right plug-in.

Each of these objects (even if there is only one for the page) is maintained in a hash table-like array (named `players[]`) in the library. The plug-in object's identifier is the string index for the array entry. This provides the gateway to your page's scripts. For example, if you initialize the library with a single identifier, `jukebox`, you access the methods of the library's jukebox-related player object through the array and the identifier:

```
players["jukebox"].rewind();
```

Invoking methods

Many of the player's method names are simple enough as-is (e.g., `play()`), but developing an API allows you to devise your own vocabulary for more complex operations. For example, Windows Media Player has no explicit method for rewinding the current tune, but you can assemble the equivalent operations into your own `rewind()` method, as shown in Listing 44-8. When your script invokes `players["jukebox"].rewind()`, the combined operations of the `API_rewind()` method do their jobs.

Listing 44-8: **The API's Primary Functions**

```
function API_play(n) {
   if (document.all(this.id).HasError) {
      alert("MediaPlayer Alert: "  + document.all(this.id).ErrorDescription);
   } else {
      document.all(this.id).PlayCount = n;
      document.all(this.id).Play();
   }
}

function API_stop() {
   document.all(this.id).Stop();
}

function API_pause() {
   // Pause() method broken for IE5+
   document.all(this.id).Stop();
}

function API_rewind() {
   document.all(this.id).Stop();
   document.all(this.id).CurrentPosition = 0;
}

function API_load(URL) {
   document.all(this.id).Open(URL);
}
```

Building a jukebox

The first example that utilizes the `MPAudioAPI.js` library is a jukebox that provides an interface (admittedly not pretty — that's for you to whip up) for selecting and controlling multiple sound files with a single plug-in tag set. The assumption for this application is that only one sound at a time need be handy for immediate playing.

Listing 44-9 shows the code for the jukebox. All sound files specified in the example are in the same folder as the listing on the companion CD-ROM (the AIFF-format files sound better in some plug-ins than others, so don't worry about the audio quality of these demo sounds).

Listing 44-9: **A Scripted Jukebox**

```
<html>
   <head>
      <title>Oldies but Goody's</title>
      <script type="text/javascript" src="MPAudioAPI.js"></script>
      <script type="text/javascript">
      // make sure currently selected tune is preloaded
      function loadFirst(id) {
         var choice = document.forms[0].musicChoice;
         var sndFile = choice.options[choice.selectedIndex].value;
         players[id].load(sndFile);
      }
      // swap tunes
      function changeTune(id, choice) {
         players[id].load(choice.options[choice.selectedIndex].value);
      }
      // control and display volume setting
      function raiseVol(id) {
         var currLevel = players[id].getVolume();
         currLevel += Math.ceil(Math.abs(currLevel)/10);
         players[id].setVolume(currLevel);
         displayVol(id);
      }
      function lowerVol(id) {
         var currLevel = players[id].getVolume();
         currLevel -= Math.floor(Math.abs(currLevel)/10);
         players[id].setVolume(currLevel);
         displayVol(id);
      }
      function displayVol(id) {
         document.forms[0].volume.value = players[id].getVolume();
      }
      </script>
   </head>
   <body onload="initAudioAPI(['jukebox'); loadFirst('jukebox');
      displayVol('jukebox')">
      <form>
         <table border="2" align="center">
            <caption align="top">
               <font size="+3">Classical Piano Jukebox</font>
            </caption>
```

```
<tr>
   <td colspan="2" align="center">
      <select name="musicChoice"
      onchange="changeTune('jukebox', this)">
         <option value="Beethoven.aif" selected="selected">
            Beethoven's Fifth Symphony (Opening)
         </option>
         <option value="Chopin.aif">
            Chopin Ballade #1 (Opening)
         </option>
         <option value="Scriabin.aif">
            Scriabin Etude in D-sharp minor (Finale)
         </option>
      </select>
   </td>
</tr>
<tr>
   <th rowspan="4">Action:</th>
   <td><input type="button" value="Play"
      onclick="players['jukebox'].play(parseInt(this.form.frequency[
      this.form.frequency.selectedIndex].value))" />
      <select name="frequency">
         <option value="1" selected="selected">
            Once
         </option>
         <option value="2">
            Twice
         </option>
         <option value="3">
            Three times
         </option>
         <option value="TRUE">
            Continually
         </option>
      </select></td>
</tr>
<tr>
   <td><input type="button" value="Stop"
      onclick="players['jukebox'].stop()" /></td>
</tr>
<tr>
   <td><input type="button" value="Pause"
      onclick="players['jukebox'].pause()" /></td>
</tr>
<tr>
   <td><input type="button" value="Rewind"
      onclick="players['jukebox'].rewind()" /></td>
</tr>
<tr>
   <th rowspan="3">Volume:</th>
   <td>Current Setting:<input type="text" size="10" name="volume"
      onfocus="this.blur()" /></td>
</tr>
<tr>
   <td><input type="button" value="Higher"
      onclick="raiseVol('jukebox')" /></td>
```

Continued

Listing 44-9 *(continued)*

```
            </tr>
            <tr>
               <td><input type="button" value="Lower"
                  onclick="lowerVol('jukebox')" /></td>
            </tr>
         </table>
      </form>

      <object classid="clsid:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
           codebase="#Version=6,0,0,0"
           width="2" height="2" id="jukebox" >
         <param name="autstart" value="false" />
      </object>
   </body>
</html>
```

You can see the user interface in Figure 44-4. One `select` element contains a list of three possible choices. Most of the interface, however, consists of buttons that ultimately invoke methods of the current plug-in.

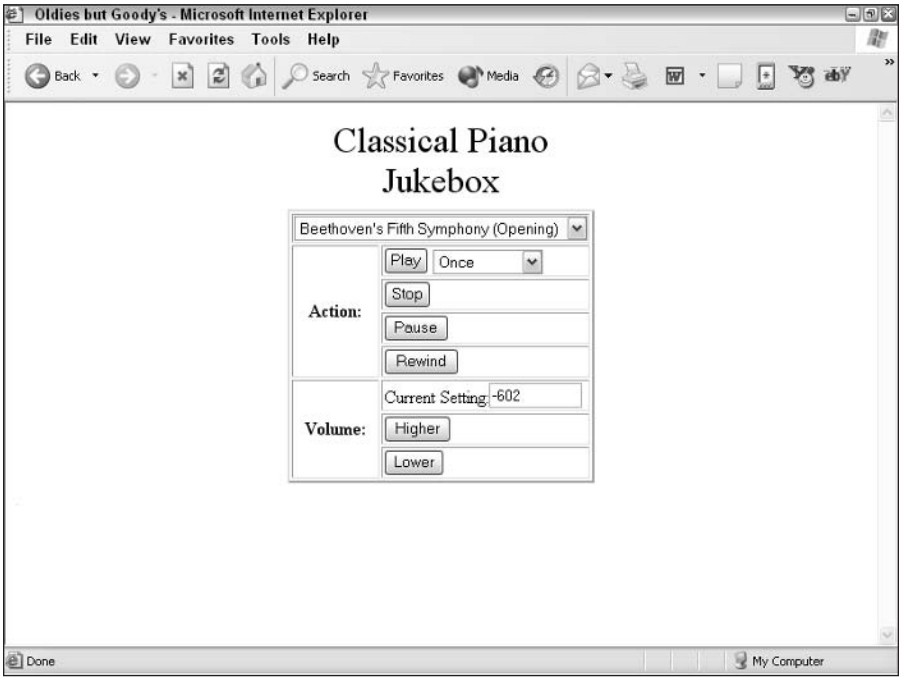


Figure 44-4: The jukebox page.

All functions defined for this page are designed to be as generalizable as possible. Thus, the identifier of the plug-in is passed as a parameter to each. If another plug-in were added to this page, the same functions could be used without modification, provided calls to the functions passed the identifier of the other plug-in.

All of the button controls are pretty straightforward except the Play button's `onclick` event handler. It invokes the `players[id].play()` method, but that method requires a parameter of how many times the sound should be played. In this user interface, a `select` element controls that information. Getting the value of the selected item creates a lengthy reference, but that's what is taking up so much space in the parameter slot of the `play()` method call.

Embedding multiple sounds

The final example of embedded media serves as a base on which you can build a page that needs to play multiple sounds without the user explicitly loading them. For example, you may have buttons generate different sounds after users click them (I'm not recommending this interface, but that won't necessarily stop you). Figure 44-5 shows you the simple five-key piano keyboard. The page loads five different sounds into the page, one for each note (actual piano sounds in this case). Each sound was recorded for about four seconds, so that you can get the action of attack and delay, just like a real piano. If you mouse-down on a key, the sound plays for up to four seconds (getting softer all the time) or until you mouse up on the key (the attack time on the sample sounds on the CD-ROM is not instantaneous, so you may have to hold a key down for a fraction of a second to start the sound). The colors of the keys also change slightly to provide further user feedback to the action.

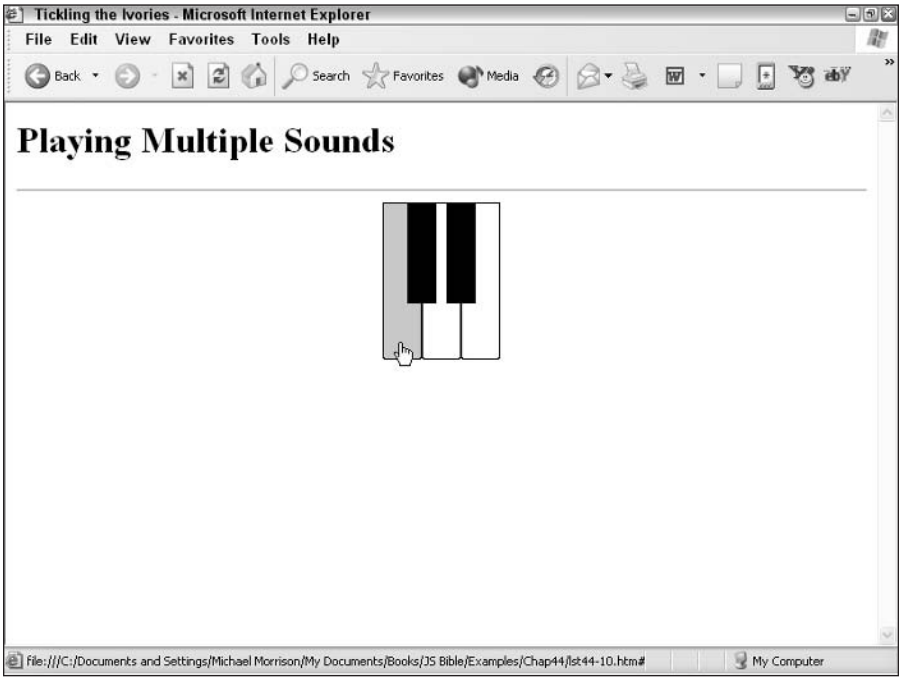


Figure 44-5: Controller for five sounds.

Thanks to the `MPAudioAPI.js` library, very little code in this page is associated with the sounds. Far more is involved with the image swaps and the loading of the five plug-ins. Listing 44-10 shows the code for the page.

Listing 44-10: **Scripting Multiple Sounds**

```
<html>
   <head>
      <title>Tickling the Ivories</title>
      <style type="text/css">
      object {visibility:hidden}
      </style>
      <script type="text/javascript" src="MPAudioAPI.js"></script>
      <script type="text/javascript">
      // pre-cache 10 images
      var onImages = new Array();
      onImages["c"] = new Image(35, 140);
      onImages["c"].src = "whiteDown.gif";
      onImages["d"] = new Image(35, 140);
      onImages["d"].src = "whiteDown.gif";
      onImages["e"] = new Image(35, 140);
      onImages["e"].src = "whiteDown.gif";
      onImages["cHalf"] = new Image(26, 90);
      onImages["cHalf"].src = "blackDown.gif";
      onImages["dHalf"] = new Image(26, 90);
      onImages["dHalf"].src = "blackDown.gif";

      var offImages = new Array();
      offImages["c"] = new Image(35, 140);
      offImages["c"].src = "whiteUp.gif";
      offImages["d"] = new Image(35, 140);
      offImages["d"].src = "whiteUp.gif";
      offImages["e"] = new Image(35, 140);
      offImages["e"].src = "whiteUp.gif";
      offImages["cHalf"] = new Image(26, 90);
      offImages["cHalf"].src = "blackUp.gif";
      offImages["dHalf"] = new Image(26, 90);
      offImages["dHalf"].src = "blackUp.gif";

      // swap images (on)
      function imgOn(img) {
         if (document.images) {
            // handle NN4 layers that hold images
            if (document.layers) {
               if (img.length == 1) {
                  document.ivories.document.images[img].src = onImages[img].src;
               } else {
                  document.ivories.document.layers["ivory" +
                  img].document.images[img].src = onImages[img].src;
               }
            } else {
               document.images[img].src = onImages[img].src;
            }
         }
      }
```

```
      // swap images (off)
      function imgOff(img) {
         if (document.images) {
            // handle NN4 layers that hold images
            if (document.layers) {
               if (img.length == 1) {
                  document.ivories.document.images[img].src =
offImages[img].src;
               } else {
                  document.ivories.document.layers["ivory" +
                  img].document.images[img].src = offImages[img].src;
               }
            } else {
               document.images[img].src = offImages[img].src;
            }
         }
      }

      // play a note (mousedown)
      function playNote(id) {
         players[id].rewind();
         players[id].play(1);
      }
      // stop playing (mouseup)
      function stopNote(id) {
         players[id].stop();
         players[id].rewind();
      }
      </script>
   </head>
   <body
   onload="initAudioAPI('cNatural','cSharp','dNatural','dSharp','eNatural')">
      <h1>
         Playing Multiple Sounds
      </h1>
      <hr />
      <table align="center">
         <tr>
            <td><div id="ivories" style="position:relative">
               <a href="#"
               onmousedown="playNote('cNatural');imgOn('c');return false"
               onmouseup="imgOff('c');stopNote('cNatural')"><img alt="image"
               name="c" src="whiteUp.gif" height="140" width="35"
               border="0" /></a><a href="#"
               onmousedown="playNote('dNatural');imgOn('d');return false"
               onmouseup="imgOff('d');stopNote('dNatural')"><img alt="image"
               name="d" src="whiteUp.gif" height="140" width="35"
               border="0" /></a><a href="#"
               onmousedown="playNote('eNatural');imgOn('e');return false"
               onmouseup="imgOff('e');stopNote('eNatural')"><img alt="image"
               name="e" src="whiteUp.gif" height="140" width="35"
               border="0" /></a> <span id="ivorycHalf"
               style="position:absolute; left:22px"><a href="#"
               onmousedown="playNote('cSharp');imgOn('cHalf');return false"
               onmouseup="imgOff('cHalf');stopNote('cSharp')"><img
               alt="image" name="cHalf" src="blackUp.gif" height="90"
```

Continued

Listing 44-10 *(continued)*

```
            width="26" border="0" /></a></span> <span id="ivorydHalf"
            style="position:absolute; left:57px"><a href="#"
            onmousedown="playNote('dSharp');imgOn('dHalf');return false"
            onmouseup="imgOff('dHalf');stopNote('dSharp')"><img
            alt="image" name="dHalf" src="blackUp.gif" height="90"
            width="26" border="0" /></a></span>
         </div></td>
      </tr>
   </table>
   <object id="cNatural" width="1" height="1"
   classid="clsid:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
   codebase="#Version=6,0,0,0">
      <param name="FileName" value="c.aif" />
      <param name="AutoStart" value="false" />
      <param name="BufferingTime" value="30" />
   </object> <object id="cSharp" width="1" height="1"
   classid="clsid:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
   codebase="#Version=6,0,0,0">
      <param name="FileName" value="cSharp.aif" />
      <param name="AutoStart" value="false" />
      <param name="BufferingTime" value="30" />
   </object> <object id="dNatural" width="1" height="1"
   classid="clsid:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
   codebase="#Version=6,0,0,0">
      <param name="FileName" value="d.aif" />
      <param name="AutoStart" value="false" />
      <param name="BufferingTime" value="30" />
   </object> <object id="dSharp" width="1" height="1"
   classid="clsid:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
   codebase="#Version=6,0,0,0">
      <param name="FileName" value="dSharp.aif" />
      <param name="AutoStart" value="false" />
      <param name="BufferingTime" value="30" />
   </object> <object id="eNatural" width="1" height="1"
   classid="clsid:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
   codebase="#Version=6,0,0,0">
      <param name="FileName" value="e.aif" />
      <param name="AutoStart" value="false" />
      <param name="BufferingTime" value="30" />
   </object>
</body>
</html>
```

Perhaps the trickiest part of this entire demonstration lies in the way the keyboard art and user interface are created. Because the white keys are not rectangular, the black key art is dropped atop the white keys by way of positioned elements.

When you use the page, you may notice a slight delay in getting the sound to be heard after pressing down on a key. On older, slower machines, this delay is even more noticeable. Take this behavior into account when designing interactive sound.

Scripting Java Classes Directly

LiveConnect, as implemented in Netscape and Mozilla browsers, allows scripts to access Java classes as if they were part of the JavaScript environment. Because you need to know your way around Java before programming Java classes directly from JavaScript, I won't get into too much detail in this book. Fortunately, the designers of JavaScript have done a good job of creating JavaScript equivalents for the most common Java language functionality, so there is not a strong need to access Java classes on a daily basis.

To script Java classes, it helps to have a good reference guide to the classes built into Java. Though intended for experienced Java programmers, *Java in a Nutshell* (O'Reilly & Associates, Inc.) offers a condensed view of the classes, their constructors, and their methods.

Java's built-in classes are divided into major groups (called *packages*) to help programmers find the right class and method for any need. Each package focuses on one particular aspect of programming, such as classes for user interface design in application and applet windows, network access, and basic language constructs, such as strings, arrays, and numbers. References to each class (object) defined in Java are "dot" references, just as in JavaScript. Each item following a dot helps zero-in on the desired item. As an example, consider one class that is part of the base language class. The base language class is referred to as

```
java.lang
```

One of the objects defined in `java.lang` is the `String` object, whose full reference is

```
java.lang.String
```

To access one of its methods, you use an invocation syntax with which you are already familiar:

```
java.lang.String.methodName([parameters])
```

To demonstrate accessing Java from JavaScript, I call upon one of Java's `String` object methods, `java.lang.String.equalsIgnoreCase()`, to compare two strings. Equivalent ways are available for accomplishing the same task in JavaScript (for example, comparing both strings in their `toUpperCase()` or `toLowerCase()` versions), so don't look to this Java demonstration for some great new powers along these lines.

Before you can work with data in Java, you have to construct a new object. Of the many ways to construct a new `String` object in Java, you use the one that accepts the actual string as the parameter to the constructor:

```
var mainString = new java.lang.String("TV Guide");
```

At this point, your JavaScript variable, `mainString`, contains a reference to the Java object. From here, you can call this object's Java methods directly:

```
var result = mainString.equalsIgnoreCase("tv Guide");
```

Even from JavaScript, you can use Java classes to create objects that are Java arrays and access them via the same kind of array references (with square brackets) as JavaScript arrays. In a few cases, you can use Java classes to obtain additional information about the user environment, such as the user's IP address (but not e-mail address). The process involves a couple of Java class calls, as follows:

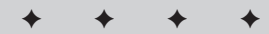
```
var localHost = java.net.InetAddress.getLocalHost();
var IP = localhost.getHostAddress();
```

The more you work with these two languages, the more you see how much Java and JavaScript have in common.

✦ ✦ ✦

CHAPTER 45

Debugging Scripts

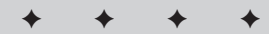


In This Chapter

Identifying the type of error plaguing a script

Interpreting error messages

Preventing problems before they occur



One of the first questions that an experienced programmer asks about a programming environment is what support is there for debugging code. Even the best coders in the world make mistakes when they draft programs. Sometimes, the mistakes are a mere slip of a finger on the keyboard; other times, they result from not being careful with expression evaluation or object references. The cause of the mistake is not the issue: finding the mistake and getting help to fix it is.

Some debugging tools are available for the latest browsers. For the most part, they have come from the browser makers themselves, or they are tied very closely to a particular authoring environment. Some of these tools are very quirky; others require significant investments in authoring environments. Discussion about debugging tools in this chapter, however, focuses on simple tools you can download online. By understanding the true meaning of error messages and working out the problem with the tools provided here, you should be able to overcome your bugs.

Syntax versus Runtime Errors

As a page loads into a JavaScript-enabled browser, the browser attempts to create an object model out of the HTML and JavaScript code in the document. Some types of errors crop up at this point. These are mostly syntax errors, such as failing to include a closing brace after a function's statements. Such errors are structural in nature, rather than about values or object references.

Runtime errors involve failed connections between function calls and their functions, mismatched data types, and undeclared variables located on the wrong side of assignment operators. Such runtime errors can occur as the page loads if the script lines run immediately as the page loads. Runtime errors located in functions won't crop up until the functions are called — either as the page loads or in response to user action.

Because of the interpreted nature of JavaScript, the distinction between syntax and runtime errors blurs. But as you work through whatever problem halts a page from loading or a script from running, you have to be aware of differences between true errors in language and your errors in logic or evaluation.

Error Message Notification

As the browsers have evolved through several generations, the ways in which script errors are reported to the user (and to you as the author) have also changed. The biggest changes came in WinIE4 and NN4.5. Prior to those versions, script errors always displayed some kind of alert dialog box with information about the error. Because these alerts could confuse non-technical users, the newer browsers are much more subtle about the presence of errors. In fact the notification mechanism is so subtle, that it is easy to miss the fact that a script error has occurred. Even if you do notice, you must then exercise your mouse a bit more to view the details.

When a script error occurs in WinIE4+, the status bar displays a yellow alert icon plus a brief text message indicating that an error has occurred. A syntax error that occurs while the page loads usually signifies that the page has loaded, but with errors. A runtime error's message simply indicates that an error occurred. To view details about the error, you must double-click the yellow icon in the status bar. The default appearance of the error message alert dialog box window includes a button named Show Details. Clicking this button expands the window to reveal whatever details the browser is reporting about the error. If you leave the window expanded, the next time it opens, it will also be expanded. It is a good idea for scripters to also check the box that forces the browser to show the error dialog box whenever an error occurs. This is simply a shortcut to manually double-clicking the status bar error icon.

Netscape/Mozilla console windows

For NN4 browsers starting with NN4.5, a user receives error notification through a message in the status bar. The instructions there indicate how to view the error details: If you type

```
javascript:
```

into the toolbar's Location box or into the dialog box that lets you open a new page, an entirely new, non-modal window appears. This window is called the Communicator Console. In contrast to the one message per window approach of IE, the Communicator Console window continues to record all script errors in sequence (in a scrolling frame), even when the Console window is closed. You can keep this window open all the time, and simply bring it to the front whenever you need to view errors. If you are developing on a large video monitor, you can let the Console window stick out to the right of the browser window. If an error occurs, not only does the message appear in the browser status bar, but you'll also see the scrollbar of the Console window's top frame appear — an even more explicit indication that an error occurred (assuming you click the Clear Console button every time you are ready to try another test run).

Netscape changed the name of the window for NN6+, now calling it the JavaScript Console. Opening this window is less cryptic than before: Choose Tasks/Tools/JavaScript Console from the menu bar. NN6+/Mozilla does not provide notification of errors in the status bar, so it is up to you to be vigilant for something running amok. This is all the more reason to keep the JavaScript Console window open while you are writing and debugging your scripts. Even if things appear to be OK, periodically check the Console window to be sure.

Safari errors

As delivered to users, Safari 1.0 completely conceals script errors. To be able to see script errors at all, you must enter a command into the MacOS X Terminal program window. The first step is to enable an otherwise hidden Debug menu in the Safari application. With Safari closed, launch Terminal and enter the following command exactly as shown:

```
defaults write com.apple.Safari IncludeDebugMenu 1
```

Launch Safari and another MacOS X utility application called Console. In the Safari Debug menu, choose "Log JavaScript Exceptions" so that the menu item is checked. Hereafter in the current session, all script errors appear in the Console window (along with other system-generated messages). Each time you launch Safari, you must enable the logging feature in the Debug menu.

Multiple error messages

The modality of IE error message alert dialog boxes tends to force just one message to appear. In other words, when the first error occurs, the browser stops recording further errors. In NN/Moz, however, it is not uncommon for multiple errors to be listed (or, in older versions, multiple error windows to show up). But you need to understand how to treat these multiple errors to get to the root of the problem.

The usual reaction is to look at the last message to appear in the sequence. That, however, is usually the error message least likely to lead you to the true problem. Error messages are dumped to the NN/Moz Console window in the order in which they occur. This means that the first error in the list is the most important error message of them all. More than likely, the first error points to a problem that throws off the rest of the script, thus triggering all of the other error messages. For example, if a statement that initializes a variable has a syntax error in it, all other statements that rely on that variable will fail, because the variable appears to be undefined.

When you encounter multiple errors, don't start any serious debugging until you locate the first error message. You must tackle this one before any others. The solution to the first one may cause the other errors to go away. This is all the more reason, when authoring in NN4.5+, to keep the Console window open, and clear it before loading any page or executing any scripts.

Error Message Details

Error reporting comes in three flavors depending on the browser: NN/Moz, WinIE, or MacIE. One of these groups may be better (that is, more accurate and explicit) at reporting some kinds of errors than the other groups. By and large, however, you can count on error details to include three basic clues to help you track down the error: the file in which the error occurred, the location of the error within the source code, and a textual description of the error. Browsers outside of NN/Moz and IE typically utilize an approach similar to one of the three mentioned here, but Safari 1.0's error reporting supplies precious little detail about any error. The Mozilla-based browsers tend to furnish the most accurate and helpful error messages.

Error filename

Although MacIE error messages do not explicitly reveal the name of the file whose source code contains the error, in practice, only the NN/Moz browsers do the best job of telling the truth. Of course, when the script and HTML are all on one page, it doesn't require a brain surgeon to know that the error occurs from that page's source code. But if you link in external `.js` libraries, the NN/Moz browsers provide the URL to the `.js` file. WinIE, on the other hand, indicates the HTML page that loads the external library, making it difficult to know precisely where the error is.

Error location

All browsers (except Safari 1.0) provide a source code line number and character position where the error supposedly occurs. For self-contained pages with no dynamically created content, the reporting tends to be accurate (also see the IE "Object expected" error message details described later in this chapter), but the accuracy is much closer in NN browsers than IE. And if your page links in an external library, the line number provided by WinIE and MacIE is practically useless. The sense you get is that the lines of the `.js` file become embedded within the main page's script, but how that is supposed to help an author find the precise problem line is a mystery — even the most feature-laden text editor knows only how to display line numbers for a single document.

NN/Moz browsers, however, not only point to the correct `.js` file, but to the line number within that file. You are much more likely to get to the root of a problem, especially in an external `.js` file, through NN/Moz error messages.

Line number reporting has improved with each browser generation, but anomalies still exist. Perhaps the most egregious is the tendency for IE to report a problem at a line number whose source code is HTML with an event handler. The problem, it turns out, will be somewhere in the function being invoked by the event handler. Another possibility in all browsers is that the line number being reported is below the line that contains the problem. Consider the following simple source code listing (with line numbers from the source code editor) that intentionally contains a syntax error (a missing brace after the first function):

```
1:    <html>
2:       <head>
3:          <script type="text/javascript">
4:          function tarzan() {
5:             var x = 1;
6:
7:          function jane() {
8:             var y = 3;
9:          }
10:         </script>
11:      </head>
12:      <body>
13:      Hello.
14:      </body>
15:   </html>
```

When you load this page into browsers, all of them report a problem with a missing right brace (NN is a bit more explicit with its message, indicating that a right brace is missing after a function body). But where do the browsers point to the error? By looking at the code as a human, you can see that the missing brace belongs in Line 6. But now examine the code from the point of view of a script interpreter engine: It sees the opening brace on Line 4, and then a new function declaration starting on Line 7. To the interpreter, this means that the `jane()` function is probably nested inside the `tarzan()` function, and it is the `tarzan()` function that is lacking the right brace following the `jane()` function. Therefore, the error line number comes in at Line 10 (although MacIE5 reports Line 9). Your scripts won't likely be this simple, so the distance between the reported error line number and the location of the actual problem can be substantial and difficult to spot without using some of the tips and tools described later in this chapter.

IE sometimes has a nasty habit of identifying the location of the problem at Line 1, Character 1. All this means is that you need to put your detective skills to work that much harder. Common causes for this behavior are references to HTML objects that don't exist (or there is a mismatch between the identifier of the element and your script reference to it) and errors that affect global functions or window methods. To find the genuine problem line, you can use tracing techniques described later in this chapter.

Error message text

Because so many permutations exist of the potential errors you can make in scripts and the ways the JavaScript interpreters in different browsers regard these errors, presenting hard-and-fast solutions to every JavaScript error message is impossible. What I can do, however, is list the most common and head-scratch–inducing error messages and relate the kinds of non-obvious problems that can trigger such messages.

"Object expected"

This error message is often one of the least helpful that you see in IE. The line number associated with the message typically points to a line in the source code that invokes a function. If you define event handlers as attributes of element tags, the line number being reported may correspond to the line containing that HTML tag.

The most obvious problem is that the function being invoked is not regarded as a valid function in the page (the "object" referred to here is the function object). This problem can be the result of an HTML or script error earlier in the document. The problem can also be the result of some error in the function itself that failed to let the interpreter treat the function as a genuine function object. Most typically, these kinds of problems are detected as syntax errors while the page loads (for example, an imbalanced set of parentheses or braces), but not always.

As a first-strike tactic, you need to determine if the function is being invoked at all. By placing an alert in the first line of the function and triggering the function, you can see if script execution is reaching that point. If that works okay, move the alert downward through the function to find out where the error is actually being triggered. The line before the alert that fails is the likely culprit.

"Expected <something>"

This message usually points straight at the problem line. Most of the "things" that the statement expects are self-explanatory. If a right parenthesis is missing from a pair, that is the "thing" shown to be expected. Detecting in the message the difference between a brace and parenthesis isn't always easy, so look at the message carefully. Not quite as intuitive is when the message says "Expected identifier". This error refers to an expression that typically is trying to use a reserved word as a variable name. Look into Appendix B for a list of reserved words, none of which you may use as names of things or variables.

"<Something> is undefined"

This message is fairly easy to understand, yet at times difficult to diagnose. For variable names, the message usually means that you have an uninitialized variable name sitting in the position of a right-hand operand or a unary operand. This variable name has not been declared or assigned with any value prior to this erroneous statement. Perhaps you're attempting to use a variable name that has been initialized only as a local variable in another function. You may

also have intended the right-hand value to be a string, but you forgot to enclose it in quotes, forcing JavaScript to look upon it as a reference to something. Another possibility is that you misspelled the name of a previously declared variable. JavaScript rightly regards this item as a new, undeclared variable. Misspellings, you will recall, include errors in upper- and lower-case in the very case-sensitive JavaScript world.

If the item is a function name, you may have to perform a bit of detective work. Though the function may be defined properly, a problem in the script above the function (for example, imbalanced braces) makes JavaScript fail to see the function. In other cases, you may be trying to invoke a function in another window or frame but forgot to include the reference to that distant spot in the call to the function.

A less likely case, but a confusing one to diagnose, is when you are assembling string versions of function calls or array references out of literal strings and variable values. The following simplified example is assembling a string that is a function call to be triggered by `setTimeout()`:

```
function doA() {
   var x = "joe";
   setTimeout("doB(" + x + ")", 5000);
}
```

Even though the value of `x` is a string when it is concatenated to the call to the `doB()` function, the value gets evaluated as if it were a variable name. An error crops up saying that "joe is undefined". Because you want to pass the value of `x` as a parameter, you must nest its value inside a pair of quotes, as follows:

```
function doA() {
   var x = "joe";
   setTimeout("doB('" + x + "')", 5000);
}
```

The difference in the code is extremely subtle, but absolutely necessary.

"<Something> is not a function"

As with the preceding one, this error message can be one of the most frustrating, because when you look at the script, it appears as though you have clearly defined a function by that name, and you're simply having an event handler or other running statement call that function. The first problems to look for are mismatched case of letters in the calling statement and function and the reuse of a variable or HTML object name by the function name.

This latter item is a no-no — it confuses JavaScript into thinking that the function doesn't exist, even though the object name doesn't have parentheses appended to it and the function does. I've also seen this error appear when other problems existed in the script above the function named in the error message, and the named function was the last one in a script.

In NN, this message appears when you attempt to invoke a function that is not implemented for a particular object. For example, if you attempt to use a W3C DOM method in NN4, the error reports that the method you tried to invoke "is not a function."

"Object doesn't support this property or method"

This IE message reports that a valid object does not provide support for a method you just attempted to invoke. In practice, this message rarely appears as the result of referencing an object's nonexistent property, because the language allows for extending an object's list of properties by assignment. If you do a lot of development in IE5+ for Windows, you may see a lot of this message when testing the page in IE5 for the Macintosh, whose complement of implemented object methods is somewhat smaller.

"Unterminated string literal"

"Unterminated string constant"

NN is far more helpful with this type of message, because along with the error message, it displays the code fragment that tripped the error. You will see the beginning (or all) of the string that is the culprit. If you simply forgot to close a string quote pair, the error most frequently appears when you try to concatenate strings or nest quoted strings. Despite the claim that you can nest alternating double and single quotes, I often have difficulties using this nesting method beyond the second nested level (single quotes inside a double-quoted string). At different times, I've gotten away with using a pair of `\"` inline quote symbols for a third layer. If that syntax fails, I break up the string so that nesting goes no deeper than two layers. If necessary, I even back out the most nested string and assign it to a variable in the preceding line — concatenating it into the more complex string in the next line.

In all versions of Navigator through NN4, avoid statements in scripts that extend for more than 255 characters. If you use a text editor that counts the column number as you type, use this measure as a guide for long statements. Break up long statements into shorter strings, but make sure to close the string at the end of the line. If you place a carriage return in the middle of a string in your source code, the first line's string is unterminated. If you want to break long lines into shorter ones, close the string at the end of each line and use the + operator:

```
var myString = "Lorem ipsum dolor sit amet, consectetaur " +
"adipisicing elit, sed do eiusmod tempor incididunt " +
"ut labore et dolore magna aliqua."
```

"Missing } after function body"

"Expected }"

This error usually is easy to recognize in a simple function definition because the closing brace is missing at the end of the function. But when the function includes additional nested items, such as `if...else` or `for` loop constructions, you begin dealing with multiple pairs of braces within the function. The JavaScript interpreter doesn't always determine exactly where the missing brace belongs, and thus it simply defaults to the end of the function. This location is a natural choice, I guess, because from a global perspective of the function, one or more of the right braces that ripple down to the end of the function usually are missing.

In any case, this error message means that a brace is missing somewhere in a function above the referenced line number. Do an inventory count for left and right braces and see whether a discrepancy occurs in the counts. One of those nested constructions is probably missing a closing brace. Some programmer-oriented text editors also include tools for finding balanced pairs of braces and parentheses.

"<Something> is not a number"

The variable name singled out in this error message is most likely a string or null value. The line of JavaScript that trips it up has an operator that demands a number. When in doubt about the data type of a variable destined for a math operation, use the `parseInt()` or `parseFloat()` functions to convert strings to numbers.

I have also encountered this error when it provides no clue about what isn't a number — the error message simply says, "is not a number." The root of the problem ended up having nothing to do with numbers. A structural imbalance in the script triggered this bogus error message.

"<Something> has no property named . . ."

"<Something> has no properties"

When a statement trips this error message, an object reference has usually gone awry in an assignment or comparison expression. You probably attempted to reference a property of an object, but something is wrong with the object reference, or you're trying to retrieve a property that doesn't exist for that object. If the reference is an extended one, you may have to dig to find the precise problem with the reference. Consider the following two statements that attempt to access the `value` property of a button named `calcMe`:

```
document.forms.calcme.value
document.forms[0].calcme.value
```

The NN errors for these two statements would read "document.forms.calcme has no properties" and "document.forms[0].calcme has no properties". Causes for the two errors are quite different. The obvious problem with them both may seem to be that the button's name is incorrectly referenced as `calcme` instead of `calcMe`. That, indeed, is the error for the second statement. But a more fundamental problem also plagues the first statement: the `document.forms` reference (a valid object, returning an array of forms) needs an array index in this instance, because it needs to look into a particular form for one of its objects. Unfortunately, both error messages look alike at first glance, and you cannot tell from them which statement has two errors and which has one.

But what you can do when this kind of error appears is use the reference that is returned with the error message to check your work. Start verifying the accuracy of your references from left to right. Later in this chapter, you see how to use the embeddable Evaluator tool to verify the existence of object references.

"<Something> is null or not an object"

This message is the IE version of the previous NN error message. A big difference is that the reference returned as part of the error message includes the complete reference. Therefore, a reference to a nonexistent `calcme` button in a form yields the error message "'document.forms[0].calcme.value' is null or not an object". Your first instinct is to be suspicious of the `value` property part of the reference. The detective work to find the problem is the same as in the NN version: verify the reference piece by piece, working from left to right. Again, the embeddable Evaluator described later in this chapter can assist in this task.

"<Something> has no property indexed by [i]"

Look carefully at the object reference in this error message. The last item has an array index in the script, but the item is not an array value type. Users commonly make this mistake within the complex references necessary for radio buttons and `select` options. Make sure that you know which items in those lengthy references are arrays and which are simply object names that don't require array values.

"<Something> can't be set by assignment"

This error message tells you either that the property shown is read-only or that the reference points to an object, which must be created via a constructor function rather than by simple assignment.

"Test for equality (==) mistyped as assignment (=)? Assuming equality test."

The first time I received this error, I was amazed by JavaScript's intelligence. I had, indeed, meant to use the equality comparison function (==) but had entered only a single equal sign. JavaScript is good at picking out these situations where Boolean values are required. In NN6+, this message is demoted to just a warning rather than an error.

"Function does not always return a value"

Often while designing deeply nested `if...else` constructions, your mind follows a single logic path to make sure that a particular series of conditions is met, and that the function returns the desired values under those conditions. What is easy to overlook is that there may be cases in which the decision process may "fall through" all the way to the bottom without returning any value, at which point the function must indicate a value that it returns, even if it is a 0 or empty (but most likely a Boolean value). JavaScript checks the organization of functions to make sure that each condition has a value returned to the calling statement. The error message doesn't tell you where you're missing the return statement, so you have to do a bit of logic digging yourself.

"Access disallowed from scripts at <URL> to documents at <URL>"

"Access is denied"

These messages (NN and IE versions, respectively) indicate that a script in one frame or window is trying to access information in another frame or window that has been deemed a potential security threat. Such threats include any `location` object property or other information about the content of the other frame when the other frame's document comes from a protocol, server, or host that is different from the one serving up the document doing the fetching.

Even the best of intentions can be thwarted by these security restrictions. For example, you may be developing an application that blends data in cooperation with another site. Security restrictions, of course, don't know that you have a cooperative agreement with the other Web site, and you have no workaround for accessing a completely different domain unless you use signed scripts for NN (see Chapter 46) or an IE user has browser security levels set dangerously loose.

Another possible trigger for these errors is that you are using two different servers in the same domain or different protocols (for example, using `https:` for the secure part of your commerce site, while all catalog info uses the `http:` protocol). If the two sites have the same domain (for example, `giantco.com`) but different server names or protocols, you can set the `document.domain` properties of documents so that they recognize each other as equals. See Chapter 46 for details on these issues and the restrictions placed on scripts that mean well, but that can be used for evil purposes.

IE, especially Windows versions, frequently clamps down too severely on inter-window and inter-frame communication. Don't be surprised to encounter security problems trying to communicate between a main window and another window whose content is dynamically generated by scripts in the main window. This error can be incredibly frustrating. Sometimes, serving the main page from a server (instead of reading it from a local hard disk) can solve the problem, but not always. You are safest if the content of both windows or frames are documents served from the same server and domain.

"Unspecified error"

This completely unhelpful IE error message is not a good sign because it means that whatever error is occurring is not part of the well-traveled decision tree that the browser uses to report errors. All is not lost, however. That the browser has not crashed means that you can still attempt to get at the root of the problem through various tracing tactics described later in this chapter.

"Uncaught exception"

You may encounter these messages in NN6+/Moz, although usually not as a result of your scripts unless you are using some of the browser's facilities to dive into inner workings of the browser. These messages are triggered by the browser's own programming code, and indicate a processing error that was not properly wrapped inside error trapping mechanisms. The details associated with such an error point to the Mozilla browser's own source code modules and internal routines. If you can repeat the error and can do so in a small test case page, you are encouraged to submit a report to `http://bugzilla.mozilla.org`, the bug tracking site for the Mozilla browser.

"Too many JavaScript errors"

You may see this message in NN if it detects a runaway train generating errors uncontrollably. This message was far more important in the days of separate error windows, because a buggy repeat loop could cause NN to generate more error windows than it could do safely.

Warnings in Mozilla's Console

The Mozilla browser's JavaScript Console window reports both outright errors and another class of notices called *warnings*. Whereas an error may cause script execution to stop, a warning is generally less catastrophic. A warning typically alerts the content author of looming problems, such as deprecated DOM terms that are still supported merely for backward compatibility — but that should be avoided going forward.

Another common warning advises that the server delivered some external content without correct headers. This happens frequently when a server supplies `.css` filters for link elements without the correct content-type header (`text/css` or similar). These are server configuration issues that should be fixed if they're under your control. You may turn off warnings in the JavaScript Console window by clicking the Errors button.

Sniffing Out Problems

It doesn't take many error-tracking sessions to get you in the save-switch-reload mode quickly. Assuming that you know this routine (described in Chapter 3), the following are some techniques I use to find errors in my scripts when the error messages aren't being helpful in directing me right to the problem.

Check the HTML tags

Before I look into the JavaScript code, I review the document carefully to make sure that I've written all my HTML tags properly. That includes making sure that all tags have closing angle brackets and that all tag pairs have balanced opening and closing tags. Digging deeper, especially in tags near the beginning of scripts, I make sure that all tag attributes that must be

enclosed in quotes have the quote pairs in place. Running your document through an HTML validator can help you spot markup mistakes in complex documents. The W3C's Markup Validation Service is a good place to start (`validator.w3c.org`).

A browser may be forgiving about sloppy HTML as far as layout goes, but the JavaScript interpreter isn't as accommodating. Finally, I ensure that the `<script>` tag pairs are in place (they may be in multiple locations throughout my document) and that the `type="text/javascript"` attribute value has both of its quotes.

View the source

Your success in locating bugs by viewing the source in the browser varies widely with the kind of content on the page and the browser you use. Very frequently, authors place perhaps too much importance in what they see in the source window.

For a straight, no-frame HTML page, viewing the source provides a modicum of comfort by letting you know that the entire page has arrived from the server. Some versions of NN might flash a questionable HTML construction, but don't expect miracles.

NN4 exhibits a notorious bug in the source view if your HTML tags include style attributes for element-specific stylesheets. You may "see double" in these lines, whereby the style attribute appears to be repeated (although usually beginning with "ttyle...") in what looks to be gibberish. This problem is a bug in the source viewer and does not accurately represent what the browser-rendering engine is using as source code.

Examining the source code for framesetting documents or individual frames, you must first give focus to the desired element. For an individual frame, click in the frame, and then right-click (or Ctrl-click on the Mac) on the frame's background to get the contextual menu. One of the items should indicate a source view of the frame. To view the framesetter's source, press the Tab key until the Address/Location field of the browser is selected. Then choose to view the source from the Edit menu.

Where the source view would be most helpful, but often fails, is to display dynamically generated HTML. Your best chance will be for pages whose entire content is generated by script. This is about the only place you can appreciate the difference between `document.write()` and `document.writeln()`, because the latter puts carriage returns after the end of each string passed as a parameter to the method. The result is a more readable source view. Most recent browsers, with the exception of NN6+/Moz, display the HTML as written by your script. NN4 does this in a window whose URL indicates the `wysiwyg:` protocol — an internal indication of dynamically generated content.

But when only part of the page is generated by script, few browsers combine the hard-wired and dynamic code in the source view. Instead, you see only the hard-wired HTML and scripts. To work around this for IE4+ and NN6+, you can use the embeddable Evaluator (later in this chapter) and read the `innerHTML` property of any elements you want.

Timing problems

One problem category that is very difficult to diagnose is the so-called timing problem. There are no hard-and-fast rules that govern when you are going to experience such a problem. Very experienced scripters develop an instinct about when timing is causing a problem that has no other explanation.

A timing problem usually means that one or more script statements are executing before the complete action of an earlier statement has finished its task. JavaScript runs within a single thread inside the browser, meaning that only one statement can run at a time. But there are times when a statement invokes some browser service that operates in its own thread and therefore doesn't necessarily finish before the next JavaScript statement runs. If the next JavaScript statement relies on the previous statement's entire task having been completed, the script statement appears to fail, even though it actually runs as it should.

These problems crop up especially when scripts work with another browser window, and especially in IE for Windows (ironic in a way). In discussions in this book about form field validation, for example, I recommend that after an instructive alert dialog box notifies the user of the problem with the form, the affected text field should be given focus and its content selected. In WinIE, however, after the user closes the alert dialog box, the script statements that focus and select operate before the operating system has finished putting the alert away and refreshing the screen. The result is that the focused and selected text box loses its focus by the time the alert has finally disappeared.

The solution is to artificially slow down the statements that perform the focus and select operations. By placing these statements in a separate function, and invoking this function via the `window.setTimeout()` method, the browser catches its breath before executing the separate function — even when the delay parameter is set to zero. A similar delay is utilized when opening and writing to a new window, as shown in the example for `window.open()` in Chapter 16.

Reopen the file

If I make changes to the document that I truly believe should fix a problem, but the same problem persists after a reload, I reopen the file via the File menu. Sometimes, when you run an error-filled script more than once, the browser's internals get a bit confused. Reloading does not clear the bad stuff, although sometimes an unconditional reload (clicking Reload while holding Shift) does the job. Reopening the file, however, clears the old file entirely from the browser's memory and loads the most recently fixed version of the source file. I find this situation to be especially true with documents involving multiple frames and tables and those that load external `.js` script library files. In severe cases, you may even have to restart the browser to clear its cobwebs, but this is less necessary in recent browsers. You should also consider turning off the cache in your development browser(s).

Find out what works

When an error message supplies little or no clue about the true location of a runtime problem, or when you're faced with crashes at an unknown point (even during document loading), you need to figure out which part of the script execution works properly.

If you have added a lot of scripting to the page without doing much testing, I suggest removing (or commenting out) all scripts except the one(s) that may get called by the document's `onload` event handler. This is primarily to make sure that the HTML is not way out of whack. Browsers tend to be quite lenient with bad HTML, so that this tactic won't necessarily tell the whole story. Next, add back the scripts in batches. Eventually, you want to find where the problem really is, regardless of the line number indicated by the error message alert.

To narrow down the problem spot, insert one or more alert dialog boxes into the script with a unique, brief message that you will recognize as reaching various stages (such as `alert("HERE-1")`). Start placing alert dialog boxes at the beginning of any groups of statements that execute and try the script again. Keep moving these dialog boxes deeper into the script (perhaps into other functions called by outer statements) until the error or crash occurs. You now know where to look for problems. See also an advanced tracing mechanism described later in this chapter.

Comment out statements

If the errors appear to be syntactical (as opposed to errors of evaluation), the error message may point to a code fragment several lines away from the problem. More often than not, the problem exists in a line somewhere above the one quoted in the error message. To find the offending line, begin commenting out lines one at a time (between reloading tests), starting with the line indicated in the error message. Keep doing this until the error message clears the area you're working on and points to some other problem below the original line (with the lines commented out, some value is likely to fail below). The most recent line you commented out is the one that has the beginning of your problem. Start looking there.

Check runtime expression evaluation

I've said many times throughout this book that one of the two most common problems scripters face is an expression that evaluates to something you don't expect (the other common problem is an incorrect object reference). In lieu of a debugger that would let you step through scripts one statement at a time while watching the values of variables and expressions, you have a few alternatives to displaying expression values while a script runs.

The simplest approaches to implement are an alert box and the status bar. Both the alert dialog box and status bar show you almost any kind of value, even if it is not a string or number. An alert dialog box can even display multiple-line values.

Because most expression evaluation problems come within function definitions, start such explorations from the top of the function. Every time you assign an object property to a variable or invoke a string, math, or date method, insert a line below that line with an `alert()` method or `window.status` assignment statement (`window.status = someValue`) that shows the contents of the new variable value. Do this one statement at a time, save, switch, and reload. Study the value that appears in the output device of choice to see if it's what you expect. If not, something is amiss in the previous line involving the expression(s) you used to achieve that value.

This process is excruciatingly tedious for debugging a long function, but it's absolutely essential for tracking down where a bum object reference or expression evaluation is tripping up your script. When a value comes back as being undefined or null, more than likely the problem is an object reference that is incomplete (for example, trying to access a frame without the `parent.frames[i]` reference), using the wrong name for an existing object (check case), or accessing a property or method that doesn't exist for that object.

When you need to check the value of an expression through a long sequence of script statements or over the lifetime of a repeat loop's execution, you are better off with a listing of values along the way. In the section "A Simple Trace Utility" later in this chapter, I show you how to capture trails of values through a script.

Debugging Tools

For a truly interactive debugger to work effectively with client-side JavaScript, the debugger must be highly integrated into the browser. This appears to have been a primary stumbling block in the availability of JavaScript debuggers. At best, we have some browser-specific debuggers, but this leaves other browsers completely out in the cold. The only browsers that have debuggers available for them are Internet Explorer for Windows, Netscape 4, and Mozilla.

WinIE Script Debugger

You can download the Microsoft Script Debugger and documentation from `http://msdn.microsoft.com/downloads/list/webdev.asp`. The debugger that is being used with IE6 is essentially the same debugger that has been around since IE4.

After you install the Script Debugger, you must tell your copy of IE to enable it. Go to the Tools ➪ Internet Options menu, click on the Advanced tab, and look for the "Disable script debugging" item in the Browsing category. Make sure that this box is unchecked. For good measure, quit and relaunch the browser. If script errors appear as usual, without an option letting you go into the debugger, help it along by choosing View ➪ Script Debugger ➪ Open.

When a runtime error occurs, you will be brought to the Debugger window, which will highlight the affected line. Script execution is paused, allowing you to inspect the value of variables and such. Open the Command window, and type any variable name (local or global) that should be active at the paused line. Press Enter, and the next line of the Command window shows the value of that variable.

You can also explicitly set breakpoints in the code, and single-step through the code. At each step, you can enter another variable name or property reference to see its current value.

Be prepared, however, for a less than satisfactory experience with this debugger. If your scripts are loaded from external `.js` files, the debugger does not track that code or line numbers reliably. In fact, it's not uncommon for the debugger to open with the completely wrong line of code highlighted, making single stepping impossible. Simply said, don't bring high expectations to the Script Debugger.

Perhaps Microsoft would rather you use the more sophisticated debugging facilities of their Visual Studio development software. It is a rather expensive solution if all you need is script debugging.

Mozilla's Venkman Debugger

When you install the Mozilla browser, you automatically receive its debugger, codenamed Venkman. This is a new component designed for the new Mozilla browser generation. You may also install it into Netscape 7 and other Mozilla-based browsers.

Access the debugger via the Tools->Web Development menu. With the debugger open, any script error brings the debugger window to the front, with execution paused at the problem line. If the code is from an external `.js` file, that file's source code is displayed, with the problem line highlighted. This debugger automatically tracks objects and their property values in one of the window's panes. You may also set breakpoints in code so that you can single-step through code and observe values at each step.

For more information about Venkman, visit `www.mozilla.org/projects/venkman`. To obtain the latest copy and pointers to documentation, visit `www.hacksrus.com/~glinda/venkman`.

Using the embeddable Evaluator

As soon as a page loads or after some scripts run, the window contains objects whose properties very likely reveal a lot about the environment at rest (that is, not while scripts are running). Those values are normally disguised from you, and the only way to guarantee successful access to view those values is through scripting within the same window or frame. That's where the embeddable Evaluator comes in handy.

As you probably recall from Chapter 13 and the many example sections of Parts III and IV of this book, the code within the standalone Evaluator provides two text boxes for entry of expressions (in the top box) and object references (the bottom box). Results of expression evaluation and object property dumps are displayed in the Results textarea between the two input boxes. A compact version of The Evaluator is contained by a separate library version called `evaluator.js` (located in the Chapter 45 folder of listings on the CD-ROM). As you embark on any substantial page development project, you should link in the library with the following tag at the top of your `head` section:

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

Be sure to either have a copy of the `evaluator.js` file in the same directory as the document under construction or specify a complete `file:` URL to the library file on your hard drive for the SRC attribute.

Immediately above the closing `body` tag of your document, include the following:

```
<script type="text/javascript">
printEvaluator();
</script>
```

If your page contains lots of positioned content, you'll need to put The Evaluator into its own positioned layer, out of the way of your primary content. For example:

```
<div style="position:absolute; top:800px; left:100px">
   <script type="text/javascript">
   printEvaluator();
   </script>
</div>
```

The `printEvaluator()` function draws a horizontal rule (`hr`) followed by the complete control panel of The Evaluator, including the codebase principal support for NN4+/Moz. From this control panel, you can reference any document object supported by the browser's object model or global variable. You can even invoke functions located in other scripts of the page by entering them into the top text box. Whatever references are available to other scripts on the page are also available to The Evaluator, including references to other frames of a frameset and even other windows (provided a reference to the newly opened window has been preserved as a global variable, as recommended in Chapter 16).

If you are debugging a page on multiple browsers, you can switch between the browsers and enter property references into The Evaluator on each browser and make sure all browsers return the same values. Or, you can verify that a DOM object and property are available on all browsers under test. If you are working on W3C DOM–compatible browsers, invoke the `walkChildNodes()` function of The Evaluator to make sure that modifications your scripts make to the node tree are achieving the desired goals. Experiment with direct manipulation of the page's objects and node tree by invoking DOM methods as you watch the results on the page.

You should be aware of only two small cautions when you embed The Evaluator into the page. First, The Evaluator declares its own one-letter lowercase global variable names (`a` through `z`) for use in experiments. Your own code should therefore avoid one-letter global variables (but local variables, such as the `i` counter of a `for` loop, are fine provided they are initialized inside a function with a `var` keyword). Second, while embedding The Evaluator at the bottom of the page should have the least impact on the rest of your HTML and scripts, any scripts that rely on the length of the `document.forms` array will end up including the form that is part of The Evaluator. You can always quickly turn off The Evaluator by commenting out the `printEvaluator()` statement in the bottom script to test your page on its own.

The embeddable Evaluator is without question the most valuable and frequently used debugging tool in my personal arsenal. It provides x-ray vision into the object model of the page at any resting point.

Emergency evaluation

Using The Evaluator assumes that you thought ahead of time that you want to view property values of a page. But what if you haven't yet embedded The Evaluator, and you encounter a state that you'd like to check out without disturbing the currently loaded page?

To the rescue comes the `javascript:` URL and the Location/Address box in your browser's toolbar. By invoking the `alert()` method through this URL, you can view the value of any property. For example, to find out the content of the cookie for the current document, enter the following into the Location/Address box in the browser:

```
javascript:alert(document.cookie)
```

Object methods or script functions can also be invoked this way, but you must be careful to prevent return values from causing the current page to be eliminated. If the method or function is known to return a value, you can display that value in an alert dialog box. The syntax for a function call is:

```
javascript:alert(myFunction("myParam1"))
```

And if you want to invoke a function that does not necessarily return a value, you should also protect the current page by using the void operator, as follows:

```
javascript:void myFunction("myParam1")
```

One more way to grab a "snapshot" of the current document tree, complete with all tags, is to use the following pseudo-URL in the Location/Address box:

```
javascript: "<textarea cols=120 rows=40>" + document.body.parentNode.innerHTML +
"</textarea>"
```

When you execute this script, the current page goes away, replaced by a page containing a textarea filled with the current HTML state of the entire document (except for the `<html>` tag). If your scripts perform document tree modification, and you want to observe the current tree state, this command acts as a cross-browser x-ray into the document. You should consider making a bookmark out of the above URL, and keep it handy.

A Simple Trace Utility

Single-stepping through running code with a JavaScript debugger is a valuable aid when you know where the problem is. But when the bug location eludes you, especially in a complex script, you may find it more efficient to follow a rapid trace of execution and viewing intermediate values along the way. The kinds of questions that this debugging technique addresses include:

- ✦ How many times is that loop executing?
- ✦ What are the values being retrieved each time through the loop?
- ✦ Why won't the while loop exit?
- ✦ Are comparison operators behaving as I'd planned in `if...else` constructions?
- ✦ What kind of value is a function returning?

A bonus feature of the embeddable Evaluator is a simple trace utility that lets you control where in your running code values can be recorded for viewing after the scripts run. The resulting report you get after running your script can answer questions like these and many more.

The trace() function

Listing 45-1 shows the `trace()` function that is built into the `evaluator.js` library file. By embedding the Evaluator into your page under construction, you can invoke the `trace()` function wherever you want to capture an interim value.

Listing 45-1: trace() function

```
function trace(flag, label, value) {
   if (flag) {
      var msg = "";
      if (trace.caller) {
         var funcName = trace.caller.toString();
         funcName = funcName.substring(9, funcName.indexOf(")") + 1);
         msg += "In " + funcName + ": ";
      }
      msg += label + "=" + value + "\n";
      document.forms["ev_evaluator"].ev_output.value += msg;
   }
}
```

The `trace()` function takes three parameters. The first, `flag`, is a Boolean value that determines whether the trace should proceed (I show you a shortcut for setting this flag later). The second parameter is a string used as a plain-language way for you to identify the value being traced. The value to be displayed is passed as the third parameter. Virtually any type of value or expression can be passed as the third parameter — which is precisely what you want in a debugging aid.

Only if the flag parameter is `true` does the bulk of the `trace()` function execute. The first task is to extract the name of the function from which the `trace()` function was called. By retrieving the rarely used `caller` property of a function, the script grabs a string copy of the entire function that has just called `trace()`. A quick extraction of a substring from the first line yields the name of the function. That information becomes part of the message text that records each trace. The message identifies the calling function followed by a colon; after that comes the label text passed as the second parameter plus an equal sign and the value parameter. The format of the output message adheres to the following syntax:

```
In <funcName>: <label>=<value>
```

The results of the trace — one line of output per invocation — are appended to the Results textarea in The Evaluator. It's a good idea to clear the textarea before running a script that has calls to `trace()` so that you can get a clean listing.

Preparing documents for trace.js

As you build your document and its scripts, you need to decide how granular you want tracing to be: global or function-by-function. This decision affects at what level you place the Boolean "switch" that turns tracing on and off.

You can place one such switch as the first statement in the first script of the page. For example, specify a clearly named variable and assign either false or zero to it so that its initial setting is off:

```
var TRACE = 0;
```

To turn debugging on at a later time, simply edit the value assigned to `TRACE` from zero to one:

```
var TRACE = 1;
```

Be sure to reload the page each time you edit this global value.

Alternatively, you can define a local `TRACE` Boolean variable in each function for which you intend to employ tracing. One advantage of using function-specific tracing is that the list of items to appear in the Results textarea will be limited to those of immediate interest to you, rather than all tracing calls throughout the document. You can turn each function's tracing facility on and off by editing the values assigned to the local `TRACE` variables.

Invoking trace()

All that's left now is to insert the one-line calls to `trace()` according to the following syntax:

```
trace(TRACE,"label",value);
```

By passing the current value of `TRACE` as a parameter, you let the library function handle the decision to accumulate and display the trace. The impact on your running code is kept to a one-line statement that is easy to remember. To demonstrate how to make the calls to `trace()`, Listing 45-2 shows a pair of related functions that convert a time in milliseconds to the string format "hh:mm". To help verify that values are being massaged correctly, the script inserts a few calls to `trace()`.

Listing 45-2: **Calling trace()**

```
function timeToString(input) {
   var TRACE = 1;
   trace(TRACE,"input",input);
   var rawTime = new Date(eval(input));
   trace(TRACE,"rawTime",rawTime);
   var hrs = twoDigitString(rawTime.getHours());
   var mins = twoDigitString(rawTime.getMinutes());
   trace(TRACE,"result", hrs + ":" + mins);
   return hrs + ":" + mins;
}

function twoDigitString(val) {
   var TRACE = 1;
   trace(TRACE,"val",val);
   return (val < 10) ? "0" + val : "" + val;
}
```

After running the script, the Results box in The Evaluator shows the following trace:

```
In  timeToString(input): input=976767964655
In  timeToString(input): rawTime=Wed Dec 13 20:26:04 GMT-0800 2000
In  twoDigitString(val): val=20
In  twoDigitString(val): val=26
In  timeToString(input): result=20:26
```

Having the name of the function in the trace is helpful in cases in which you might justifiably reuse variable names (for example, `i` loop counters). You can also see more clearly when one function in your script calls another.

One of the most valuable applications of the `trace()` function comes when your scripts accumulate HTML that gets written to other windows or frames, or replaces HTML segments of the current page. Because the source view may not display the precise HTML that you assembled, you can output it via the `trace()` function to the Results box. From there, you can copy the HTML and paste it into a fresh document to test in the browser by itself. You can also verify that the HTML content is being formatted the way that you want it.

Browser Crashes

Each new browser generation is less crash-prone than its predecessor, which is obviously good news for everyone. It seems that most crashes, if they occur, do so as the page loads. This can be the result of some ill-advised HTML, or something happening as the result of script statements that either run immediately as the page loads or in response to the `onload` event handler.

Finding the root of a crash problem is perhaps more time consuming because you must relaunch the browser each time (and in some cases, even reboot your computer). But the basic tactics are the same. Reduce the page's content to the barest minimum HTML by commenting out both scripts and all but `head` and `body` tags. Then begin enabling small chunks to test reloading of the page. Be suspicious of `meta` tags inserted by authoring tools. Their removal can sometimes clear up all crash problems. Eventually you will add something into the mix that will cause the crash. It means that you are close to finding the culprit.

Preventing Problems

Even with help of authoring and debugging tools, you probably want to avoid errors in the first place. I offer a number of suggestions that can help in this regard.

Getting structure right

Early problems in developing a page with scripts tend to be structural: knowing that your objects are displayed correctly on the page; making sure that your `<script>` tags are complete; and completing brace, parenthesis, and quoted pairs. I start writing my page by first getting down the HTML parts — including all form definitions. Because so much of a scripted page tends to rely on the placement and naming of interface elements, you will find it much easier to work with these items after you lay them out on the page. At that point, you can start filling in the JavaScript.

When you begin defining a function, repeat loop, or `if` construction, fill out the entire structure before entering any details. For example, when I define a function named `verifyData()`, I enter the entire structure for it:

```
function verifyData() {

}
```

I leave a blank line between the beginning of the function and the closing brace in anticipation of entering at least one line of code.

After I decide on a parameter to be passed and assign a variable to it, I may want to insert an `if` construction. Again, I fill in the basic structure:

```
function verifyData(form) {
   if (form.checkbox.checked) {

   }
}
```

This method automatically pushes the closing brace of the function lower, which is what I want — putting it securely at the end of the function where it belongs. It also ensures that I line up the closing brace of the `if` statement with that grouping. Additional statements in the `if` construction push down the two closing braces.

If you don't like typing or don't trust yourself to maintain this kind of discipline when you're in a hurry to test an idea, you should prepare a separate document that has templates for the common constructions: `<script>` tags, function, `if`, `if...else`, `for` loop, `while` loop, and conditional expressions. Then if your editor and operating system support it, drag and drop the necessary segments into your working script.

Build incrementally

The worst development tactic you can follow is to write tons of code before trying any of it. Error messages may point to so many lines away from the source of the problem that it could take hours to find the true source of difficulty. The save-switch-reload sequence is not painful, so the better strategy is to try your code every time you have written a complete thought — or even enough to test an intermediate result in an alert dialog box — to make sure that you're on the right track.

Test expression evaluation

Especially while you are learning the ins and outs of JavaScript, you may feel unsure about the results that a particular string, math, or date method yields on a value. The longer your scripted document gets, the more difficult it will be to test the evaluation of a statement. You're better off trying the expression in a more controlled, isolated environment, such as The Evaluator. By doing this kind of testing in the browser, you save a great deal of time experimenting by going back and forth between the source document and the browser.

Build function workbenches

A similar situation exists for building and testing functions, especially generalizable ones. Rather than test a function inside a complex scripted document, drop it into a skeletal document that contains the minimum number of user interface elements that you need to test the function. This task gets difficult when the function is closely tied to numerous objects in the real document, but it works wonders for making you think about generalizing functions for possible use in the future. Display the output of the function in a text or textarea object or include it in an alert dialog box.

Testing Your Masterpiece

If your background strictly involves designing HTML pages, you probably think of testing as determining your user's ability to navigate successfully around your site. But a JavaScript-enhanced page — especially if the user enters input into fields or implements Dynamic HTML techniques — requires a substantially greater amount of testing before you unleash it to the online masses.

A large part of good programming is anticipating what a user can do at any point and then being sure that your code covers that eventuality. With multiframe windows, for example, you need to see how unexpected reloading of a document affects the relationships between all the frames — especially if they depend on each other. Users will be able to click Reload at any time or suspend document loading in the middle of a download from the server. How do these activities affect your scripting? Do they cause script errors based on your current script organization?

The minute you enable a user to type an entry into a form, you also invite the user to enter the wrong kind of information into that form. If your script expects only a numeric value from a field, and the user (accidentally or intentionally) types a letter, is your script ready to handle that "bad" data? Or no data? Or a negative floating-point number?

Just because you, as author of the page, know the "proper" sequence to follow and the "right" kind of data to enter into forms, your users will not necessarily follow your instructions. In days gone by, such mistakes were relegated to "user error." Today, with an increasingly consumer-oriented Web audience, any such faults rest solely on the programmer — you.

If I sound as though I'm trying to scare you, I have succeeded. I was serious in the early chapters of this book when I said that writing JavaScript is *programming*. Users of your pages are expecting the same polish and smooth operation (no script errors and certainly no crashes) from your site as from the most professional software publisher on the planet. Don't let them or yourself down. Test your pages extensively on as many browsers and as many operating systems as you can and with as wide an audience as possible before putting the pages on the server for all to see.

✦ ✦ ✦

CHAPTER 46

Security and Netscape Signed Scripts

In This Chapter

Exploring browser security policies

Applying JavaScript to Navigator security mechanisms

Using Netscape signed scripts

The paranoia levels about potential threats to security and privacy on the Internet are at an all-time high. As more people rely on e-mail and Web site content for their daily lives and transactions, the fears will only increase for the foreseeable future (an indeterminate number of Web Weeks). As a jokester might say, though, "I may be paranoid, but how do I know someone really isn't out to get me?" The answer to that question is that you don't know, and such a person may be out there.

But Web software developers are doing their darnedest to put up roadblocks to those persons out to get you — hence, the many levels of security that pervade browsers. Unfortunately, these roadblocks also get in the way of scripters who have completely honest intentions. Designing a Web site around these barriers is one of the greatest challenges that many scripters face.

Battening Down the Hatches

When Navigator 2 first shipped to the world way back in the previous century (February 1996), it was the first browser released to include support for Java applets and scripting — two entirely different but often confused technologies. It didn't take long for clever programmers in the Internet community to find the ways in which one or the other technology provided inadvertent access to client computer information (such as reading file directories) and Web surfer activities (such as histories of where you've been on the Net and even the passwords you may have entered to access secure sites).

JavaScript, in particular, was the avenue that many of these programmers used to steal such information from Web site visitors' browsers. The sad part is that the same features that provide the access to the information were intentionally made a part of the initial language to aid scripters who would put those features to beneficial use in controlled environments, such as intranets. But out in the Wild Wide Web, a scripter could capture a visitor's e-mail address by having the site's home page surreptitiously send a message to the site's author without the visitor even knowing it.

Word of security breaches of this magnitude not only circulated throughout the Internet, but also reached both the trade and mainstream press. As if the security issues weren't bad enough on their own, the public relations nightmare compounded the sense of urgency to fix the problem. To that end, Netscape released two revised editions of Navigator 2. The final release of that generation of browser, Navigator 2.02, took care of the scriptable security issues by turning off some of the scripted capabilities that had been put into the original 2.0 version. No more capturing visitors' browser histories; no more local file directory listings; no more silent e-mail. Users could even turn off JavaScript support entirely if they so desired.

The bottom line on security is that scripts are prevented from performing automated processes that invade the private property of a Web author's page or a client's browser. Thus, any action that may be suspect, such as sending an e-mail message, requires an explicit action on the part of the user — clicking a Submit button, in this case — to carry it out. Security restrictions must also prevent a Web site from tracking your activity beyond the boundaries of that Web site.

When Worlds Collide

If a script tries to do something that is not allowed or is a potential personal security breach, the browser reports the situation to the user. For example, browsers display a warning to the user when a script in a subwindow tries to close the main window.

Another security error message often confuses scripters who don't understand the possible privacy invasions that can accrue from one window or frame having access to the URL information in another window or frame. In WinIE5+, for example, an ominous error message — "Access denied" — warns users of an attempt to access URL information from another frame if that URL is from a different Web site.

Despite the fact that a scripted Web site may have even loaded the foreign URL into the other frame, the security restrictions guard against unscrupulous usage of the ability to snoop in other windows and frames.

The Java Sandbox

Much of the security model for JavaScript is similar to that originally defined for Java applets. Applets had a potentially dangerous facility of executing Java code on the client machine. That is a far cry from the original deployment of the World Wide Web as a read-only publishing medium on the Internet. Here were mini-programs downloaded into a client computer that, if unchecked, could have the same access to the system as a local software program.

Access of this type would clearly be unacceptable. Imagine the dismay caused by someone clicking a link that said "Free Money," only to have the linked page download an applet that read or damaged local disk files unbeknownst to the user. In anticipation of pranksters, the designers of Java and the Java virtual machine built in a number of safeguards to prevent applets from gaining access to local machines. This mechanism is collectively referred to as the *sandbox*, a restricted area in which applets can operate. Applets cannot extend their reach outside of the sandbox to access local file systems and many sensitive system preferences. Any applet runs only while its containing page is still loaded in the browser. When the page goes away, so does the applet, without being saved to the local disk cache.

JavaScript adopted similar restrictions. The language provided no read or write access to local files beyond the highly regulated cookie file. Moreover, because JavaScript works more closely with the browser and its documents than applets typically do, the language had to build in extra restrictions to prevent browser-specific privacy invasions. For example, it was not possible for a script in one window to monitor the user's activity in another window, including the URL of the other window, if the page didn't come from the same server as the first window. Sometimes the restrictions on the JavaScript side are even more severe than in Java. For example, while a Java applet is permitted to access the network anytime after the applet is loaded, an applet is prevented from reaching out to the Net if the trigger for that transaction comes from JavaScript via LiveConnect (see Chapter 44). Only partial workarounds are available.

Neither the Java nor JavaScript security blankets were fully bug-free at the outset. Some holes were uncovered by the languages' creators and others in the community. To their credit, Sun, Netscape, and Microsoft are quick to plug any holes that are discovered. Although the plugs don't necessarily fix existing copies of insecure browsers out there, it means that a Bad Guy can't count on every browser to offer the same security hole for exploitation. That generally makes the effort not worth the bother.

Security Policies

Netscape has defined security mechanism under the term *policies*. This usage of the word mirrors that of institutions and governments: A policy defines the way potentially insecure or invasive requests are handled by the browser or scripting language. NN4+/Mozilla includes two different security policies: *same origin* and *signed* script policies. The same origin policy dates back to Navigator 2, although some additional rules have been added to that policy as Navigator has matured. The signed script policy started with NN4 and utilizes the state of the art in cryptographic signatures of executable code inside a browser, whether that code is a plug-in, a Java applet, or a JavaScript script. Because of the signed script facilities, NN4 was designed to allow scripts to have wider range of control over the browser's interior working parts, provided the user granted permission for such activity (more about this later in the chapter). These same facilities are built into today's Mozilla-based browsers. NN3 included a partially implemented prototype of another policy known as data tainting. Signed scripts supersede data tainting, so if you encounter any writings about data tainting, you can ignore them because the technology was abandoned long ago.

By and large, the same origin policy is in force inside IE3 and after. Precise details may not match up with NN one-for-one, but the most common features are identical. The signed script policy is implemented only in NN4+. While Microsoft offers digital signatures for some items that may be embedded within an HTML page (such as ActiveX controls and other components), scripts that are in an HTML page's source code or linked in as a `.js` library cannot be signed for IE. Although everything you read in this chapter about signed scripts applies only to NN4+, you should find the next couple of sections informative even if you develop solely for IE.

The Same Origin Policy

The "origin" of the same origin policy means the protocol and domain of a source document. If all of the source files currently loaded in the browser come from the same server and domain, scripts in any one part of the environment can poke around the other documents. Restrictions come into play when the script doing the poking and the document being poked come from different origins. The potential security and privacy breaches this kind of access can cause put this access out of bounds within the same origin policy.

An origin is not the complete URL of a document. Consider the two popular URLs for Netscape's Web sites:

```
http://home.netscape.com
http://devedge.netscape.com
```

The protocol for both sites is `http:`. Both sites also share the same domain name: `netscape.com`. But the sites run on two different servers: `home` and `devedge` (at least this is how the sites appear to browsers accessing them; the physical server arrangement may be quite different).

If a frameset contains documents from the same server at `netscape.com`, and all frames are using the same protocol, they have the same origin. Completely open and free access to information, such as `location` object properties, is available to scripts in any frame's document. But if one of those frames contains a document from the other server, their origins don't match. A script in a document from one server would display an "access disallowed" or "permission denied" error message if it tried to get the `location` property of that other document.

A similar problem occurs if you were creating a Web-based shopping service that displays the product catalog in one window and displays the order form from a secure server in another window. The order form, whose protocol might be `https:`, would not be granted access to the `location` object properties in a catalog page whose protocol is `http:`, even though both share the same server and domain name.

Setting the document.domain

When both pages in an origin-protected transaction are from the same domain (but different servers or protocols), you can instruct JavaScript to set the `document.domain` properties of both pages to the domain that they share. When this property is set to that domain, the pages are treated as if from the same origin. Making this adjustment is safe, because JavaScript doesn't allow setting the `document.domain` property to any domain other than the origin of the document making the setting. See the `document.domain` property entry in Chapter 18 for further details.

Origin checks

Scattered throughout the language reference chapters are notes about items that undergo what you now know to be origin checks. For the sake of convenience, I list them all here to help you get a better feeling for the kind of information that is protected. The general rule is that any object property or method that exposes a local file in a user's system or can trace Web surfing activity in another window or frame undergoes an origin check. Failure to satisfy the same origin rule yields an "access disallowed" or "permission denied" error message on the client's machine.

Window object checks

The document object models of windows and frames that don't share the same origin are not available to each other. Each separate origin window or frame is its own little world that has very little ability to communicate with another window or frame. IE sometimes takes this to the extreme, causing problems between a main window and a subwindow whose content is entirely dynamically generated from the main window's scripts.

Location object checks

All `location` object properties are restricted to same origin access. Of all same origin policy restrictions, this one seems to interfere with well-meaning page authors' plans when they want to provide a frame for users to navigate around the Web. Such access, however, would allow spying on your users.

Document object checks

A `document` object's properties are by necessity loaded with information about the content of that document. Just about every property other than the ones that specify color properties are off-limits if the origin of the target document is different from the one making the request:

```
anchors[]    lastModified
applets[]    length
cookie       links[]
domain       referrer
embeds       title
forms[]      URL
```

In addition, no normally modifiable `document` property can be modified if the origin check fails. This, of course, does not prevent you from using `document.write()` to write an entirely new page of content to the frame to replace a document from a different origin. But in IE4+ and W3C DOM browsers, scripts from one origin won't be able to modify (or even copy) partial content from a frame whose content comes from another origin.

NN4 layer object checks

Although most of a NN4 layer's content is protected by the restrictions that apply to the `document` object inside, a layer object also has a potentially revealing `src` property. This is essentially similar to the `location.href` property of a frame. Thus the `src` property requires an origin check before yielding its information.

Form object checks

Form data is generally protected by the restriction to a document's `forms[]` array. But should a script in another window or frame also know the name of the form, that, too, won't enable access unless both documents come from the same origin.

Applet object checks

The same goes for named Java applets. A script cannot retrieve information about the class filename unless both documents are from the same origin (although the applet can be from anywhere).

LiveConnect access from a Java applet to JavaScript is not an avenue to other windows and frames from other origins. Any calls from the applet to the objects and protected properties described here undergo origin checks when those objects are in other frames and windows. The applet assumes the origin of the document that contains the applet, not the applet codebase.

Image object checks

While image objects are accessible from other origins, their `src` and `lowsrc` properties are not. These URLs could reveal some or all the URL info about the document containing them.

Linked script library checks

To prevent a network-based script from hijacking a local script library file, NN4+ prevents a page from loading a `file:` protocol library in the `src` attribute of a `<script>` tag unless the main document also comes from a `file:` protocol source. If you are beginning to think that security engineers are a suspicious and paranoid lot, you are starting to get the idea. It's not easy to curb potential abuses of Bad Guys in a networked environment initially established for openness and free exchange of information among trusted individuals.

The Netscape Signed Script Policy

Just as there are excellent reasons to keep Web pages from poking around your computer and browser, there are equally good reasons to allow such access to a Web site you trust not to be a Bad Guy. To permit trusted access to the client machine and browser, Sun Microsystems and Netscape (in cooperation with other sources) developed a way for Web application authors to identify themselves officially as authors of the pages and to request permission of the user to access well-defined parts of the computer system and browser.

The technology is called object *signing*. In broad terms, object signing means that an author can electronically lock down a chunk of computer code (whether it be a Java applet, a plug-in, or a script) with the electronic equivalent of a wax seal stamped by the author's signet ring. At the receiving end, a user is informed that a sealed chunk of code is requesting some normally protected access to the computer or browser. The user can examine the "seal" to see who authored the code and the nature of access being requested. If the user trusts the author not to be a Bad Guy, the user grants permission for that code to execute; otherwise the code does not run at all. Additional checks take place before the code actually runs. That electronic "seal" contains an encrypted, reduced representation of the code as it was locked by the author. If the encrypted representation cannot be re-created at the client end (it takes only a fraction of a second to check), it means the code has been modified in transit and will not run.

In truth, nothing prevents an author from being a Bad Guy, including someone you may normally trust. The point of the object signing system, however, is that a trail leads back to the Bad Guy. An author cannot use this technology to sneak into your computer or browser without your explicit knowledge and permission.

Signed objects and scripts

A special version of the signed object technology is the one that lets scripts be locked down by their author and electronically signed. Virtually any kind of script in a document can be signed: a linked `.js` library, scripts in the document, event handlers, and JavaScript entities. As described later in this chapter, you must prepare your scripts for being signed, and then run the entire page through a special tool that attaches your electronic signature to the scripts within that page.

What you get with signed scripts

If you sign your scripts and the user grants your page permission to do its job, signed scripts open up your application to a long list of capabilities, some of which border on acting like genuine local applications. Because the designers of NN4+ know that signed scripts are available to authors, a huge number of properties and actions are exposed to authors.

The most obvious power you get with signed scripts is freedom from the restrictions of the same origin policy. All object properties and methods that perform origin checks for access in other frames and windows become available to your scripts without any special interaction with the user beyond the dialog box that requests the one-time permission for the page.

Some operations that normally display warnings about impending actions — sending a form to a `mailto:` URL or closing the main browser window under script control — lose those warning dialog boxes if the user grants the appropriate permission to a signed script. Object properties considered private information, such as individual URLs stored in the history object and browser preferences, are opened up, including the possibility of altering browser preferences.

Existing windows can have their chrome elements hidden. New windows can be set to be always raised or lowered, sized to very small sizes, or positioned offscreen. The `dragdrop` event of a window reveals its URL. All of these are powerful points of access, provided the user grants permission.

Again, however, I emphasize that these capabilities are accessible via Netscape's signed script policy only. Internet Explorer, at least through Version 6, does not support Netscape's signed script policy.

The Digital Certificate

Before you can sign a script or other object, you must apply for a digital certificate. A digital certificate (also called a digital `id`) is a small piece of software that gets downloaded and bound to the developer's Navigator browser on a particular computer. Each downloaded digital certificate is accessible from the Privacy & Security window of Navigator 7's Preferences, which are found by selecting Preferences from the Edit menu. Open the Privacy & Security category by double-clicking it, which reveals a list of subcategories. The Certificates category contains a button named Manage Certificates, which leads you to the Certificate Manager. The Certificate Manager displays a list of the certificates currently available on your system. If you have not yet applied for a certificate, the list is empty. When you sign a page with the certificate, information about the certificate is included in the file generated by the signing tool.

Possession of a certificate makes you what is known as a principal. If a user loads a page that has signed "stuff" in it, a security alert advises the user that a Web site is requesting enhanced privileges.

Certificates are issued by organizations established as certificate authorities. A certificate authority (known as a `ca` for short), or a certificate server authorized by a `ca`, registers applicants and issues certificates to individuals and software developers. When you register for a certificate, the `ca` queries you for identification information, which it verifies as best it can. The certificate that is issued to you identifies you as the holder of the certificate. Under the "Authorities" category of the Security Manager window are the certificate authorities loaded into the browser when you installed the browser. These are organizations that issue certificates. The `ca` of the organization that issues your certificate must be listed for you to sign scripts.

How to get a certificate

You must visit a certificate vendor to obtain your certificate. The cost ranges from about $20 to many hundreds of dollars depending on the vendor and the type of certificate you want to obtain. One vendor that is aware of the needs of Netscape object signing is Thawte Digital Certificate Services (`www.thawte.com`). This `ca` offers a certificate expressly for developers performing Netscape object signing. Verisign (`www.verisign.com`) is also recommended.

Because one of the foundations of a certificate is the identity of the certificate owner, registration requires submitting documentation that proves the identity of your organization. Each `ca` has different requirements, so check the latest information at the `ca`'s enrollment Web site. After the `ca` processes your application, the company sends you an e-mail message with a code number to pick up your certificate at the `ca`'s Web site. The act of picking up the certificate is actually downloading the certificate into your browser. Therefore, be sure you are using the Navigator browser on the computer with which you will use to sign your pages.

Activating the codebase principal

If you want to try out the capabilities available to signed scripts from a server without purchasing a certificate (or without going through the signing process described later in this chapter during script development and debugging), you can set up your copy of Navigator to accept what is called a *codebase* principal in place of a genuine certificate. A codebase principal means that the browser accepts the source file as a legitimate principal, although it contains no identification as to the owner or certificate.

You can experiment with properties and methods that normally require signed scripts without modifying the codebase principal preferences or signing scripts with a certificate. Each time you activate the Privilege Manager (as The Evaluator does when you check the "Use NN Codebase Security" checkbox), a security alert lets you elect to continue with the untrusted access to your browser or system. **Do not check the box in the alert dialog that remembers this choice** because it could leave your browser and system vulnerable to other unsigned scripts without your knowledge.

Depending on which version of Navigator you are running, if you set up your browser for codebase principals, you may not be able to verify a certificate that is presented to you when accessing someone else's Web site — even if it is a valid cryptographic certificate. Therefore, even though secure requests won't slip past you silently, your Navigator won't necessarily have the protective shield it normally does to identify certificate holders beyond the URL of the code. Enable codebase principals only on a copy of Navigator that doesn't venture beyond your development environment. To activate codebase principals for your copy of Navigator, follow these steps:

1. Quit Navigator.
2. Search your hard disk for a Navigator 4 support file named `prefs.js` or Navigator 6+ support file named `all.js`.
3. Edit the version-specific file in a text editor as follows:
    a. For NN4, add the following line to the end of the `prefs.js` file:

    ```
    user_pref("signed.applets.codebase_principal_support", true);
    ```

    b. For NN6+, change the codebase_principal preference in `all.js` from `false` to `true`:

    ```
    pref("signed.applets.codebase_principal_support", true);
    ```

4. Save the file.

To deactivate codebase principals, quit Navigator and then change the `true` setting of the affected line to `false`. Because NN/Moz rebuilds the preference file upon quitting, the entry will be in alphabetical order rather than at the end of the file where you first entered it. This preferences setting does not affect your ability to sign scripts with your certificate as described in the rest of this chapter.

Signing Scripts

The process of signing scripts entails some new concepts for even experienced JavaScript authors. You must use a separate signing tool program. You must also prepare the page that bears scripts so that the tool and the object signing facilities of the browser can do their jobs.

Signing tool

Download the latest version of Netscape's SignTool from links you find at `http://developer.netscape.com:80/software/signedobj/jarpack.html` (you find different versions for a variety of Windows and UNIX versions). This tool includes a utility program called a `JAR` Packager. A `JAR` file is a special kind of zipped file collection that has been designed to work with the Navigator security infrastructure. The letters of the name stand for Java ARchive, which is a file format standard developed primarily by Sun Microsystems in cooperation with Netscape and others.

A `JAR` file's extension is `.jar`, and when it contains signed script information, it holds at least one file, known as the *manifest*, or list of items zipped together in the file. Among the items in the manifest is certificate information and data (a hash value code) about the content of the signed items at the instant they were signed. In the case of a single page containing signed scripts, the `JAR` file contains only the certificate and hash values of the signed scripts within the document. If the document links in an external `.js` script library file, that library file is also packaged in the `JAR` file. Thus, a page with signed scripts occupies space on the server for the HTML file and its companion `JAR` file.

The SignTool program combines the `JAR` Packager with the script signing functions (originally a separate program called `zigbert.exe`). Follow links on the SignTool download page to the latest instructions on packaging and signing your finished files from the command line (there is no `gui` for this tool). But before you reach that point, you need to compose your pages in a way that the security mechanism can protect your scripts.

Preparing scripts for signing

Signifying which items in a page are script items that require signing is up to the page author. It is important to remember that if you want to sign even one script element in a document, *every* script in the document must be signed. By "document," I mean an object model document. Because the content of an NN4-only `<layer>` tag exists in its own document, you don't have to sign its scripts if they don't require it, nor communicate with the signed scripts in the main document.

The first concept you have to master is recognizing what a script is. For signing purposes, a script is more than just the set of statements between a `<script>` and `</script>` tag boundary. An event handler — even one that calls a function living in a `<script>` tag — is also a script that needs signing. So, too, is a JavaScript entity used to supply a value to an HTML tag attribute. Each one of these items is a script as far as script signing is concerned.

Your job is to mark up the file with special tag attributes that do two things: 1) help SignTool know what items to sign in a file; and 2) help the browser loading the signed document know what items to run through the hash routine again to compare against the values stored in the `JAR` file.

The archive attribute

The first attribute goes in the first `<script>` tag of the file, preferably near the very top of the document in the `<head>` portion. This attribute is the `archive` attribute, and its value is the name of the `JAR` file to be associated with the HTML file. For example:

```
<script type="text/javascript" archive="myArchive.jar" id="1">
```

You can add script statements to this tag or immediately end it with a `</script>` tag.

The SignTool utility uses the archive attribute to assign a name to its archive output file. After the signed page loads into the visitor's browser, the attribute points to the file containing signed script information. Having more than one JAR archive file associated with a signed page is possible. Typically, such a situation calls for a second JAR archive overseeing a confined portion of the page. That second archive file may even be embedded in the primary archive file, allowing a script segment signed by one principal to be combined with scripts signed by a different principal.

The id attribute

More perplexing to scripters coming to script signing for the first time is the id attribute. The id attribute is merely a label for each script. Each script must have a label that is unique among all labels specified for a JAR archive file.

As with the archive attribute, the id plays a dual role. When you run your page through SignTool, the utility scans the page for these id attributes. When SignTool encounters one, it calculates a hash value (something like a checksum) on the content of the script. For a <script> tag set, it is for the entire content of the tag set; for an event handler, it is for the event handler text. The hash value is associated with the id attribute label and stored inside the JAR file. After the document loads into the client's browser, the browser also scans for the id attributes and performs the same hash calculations on the script items. Then the browser can compare the id/hash value pairs against the list in the JAR file. If they match, then the file has arrived without being modified by a Bad Guy (or a dropped bit in the network).

Most examples show id attribute values to be numbers, but the attributes are actually strings. No sequence or relationship exists among id attribute values: you can use the names of your favorite cartoon show characters, as long as no two id attributes are given the same name. The only time the same id attribute value may appear in a document is if another JAR file is embedded within the main JAR file. Even so, I recommend avoiding reusing names inside the same HTML file, no matter how many JAR files are embedded.

With one exception, each script item in a document must have its own id attribute. The exception is a <script> tag that specifies a src attribute for an external .js file. That file is part of the JAR file, so the browser knows it's a signed script.

For other <script> tags, include the id attribute anywhere within the opening tag, as follows:

```
<script type="text/javascript" id="3">
   statements
</script>
```

For a function handler, the id attribute comes after the event handler inside the object tag, as follows:

```
<input type="button" value="Calculate" onclick="doCalc(this.form)" id="bart" />
```

And for a JavaScript entity, the id attribute must be specified in an empty <script> tag immediately before the tag that includes the entity for an attribute value, as follows:

```
<script id="20">
<input type="text" name="date" value=&{getToday()}; />
```

Listing 46-1 shows a skeletal structure of a document that references a single JAR file and includes five signed scripts: one external .js file and four script items in the document itself. The fetchFile() function invokes a function imported from access.js. Notice that the archive attribute appears in the very first <script> tag in the document. This also happens to be a tag that imports an external .js file, so that no id attribute is required. If there were

no external library file for this page, the `archive` attribute would be located in the main `<script>` tag, which also has the `id` attribute. I arbitrarily assigned increasing numbers as the `id` attribute values, but I could have used any identifiers. Notice, too, that each script has its own `id` value. Just because an event handler invokes a function in a `<script>` tag that has an `id` value doesn't mean a relationship exists between the `id` attribute values in the `<script>` tag and in the event handler that invokes a function there.

Listing 46-1: **Basic Signed Script Structure**

```
<html>
   <head>
      <title>Signed Scripts Testing</title>
      <script type="text/javascript" archive="myArchive.jar" src="access.js">
      </script>
      <script type="text/javascript" id="1">
      function fetchFile(form) {
         form.output.value = getFile();
      }
      function newRaisedWindow() {
         // statements for this function
      }
      </script>
   </head>
   <body>
      A Source Code Example Only
      <form>
         <textarea name="output" cols="60" rows="10" wrap="virtual">
         </textarea><br />
          <input type="button" value="Read File"
         onclick="this.form.output.value = ''; fetchFile(this.form);"
         id="2" /><br />
          <textarea name="input" cols="60" rows="10" wrap="virtual">
         </textarea><br />
          <input type="button" value="Save File"
         onclick="setFile(this.form.input.value);" id="3" />
         <p><input type="button" value="New Window&#201;"
            onclick="newRaisedWindow();" id="4" /></p>
      </form>
   </body>
</html>
```

Editing and moving signed scripts

The nature of the script signing process requires that even the slightest modification you make to a signed script source code requires re-signing the page. For this reason, enabling codebase principals while you create and debug your early code is a convenient alternative.

The rigid link between the hash value of a script element at both the signing and visitor loading times means that you must exercise care when shifting an HTML file that contains signed script elements between servers of differing operating systems. Windows, UNIX, and Macintosh have different ways of treating carriage returns. If you change the representation of an HTML source file when you move the source from, say, a Windows machine to a UNIX server, the

signature may no longer work. However, if you perform a purely binary transfer of the HTML files, every byte is the same, and the signature should work. This operating system-specific text representation affects only how files are stored on servers, not how various client platforms interpret the source code.

Accessing Protected Properties and Methods

For the browser to allow access to protected properties or methods, it must have its privileges enabled. Only the user can grant permission to enable privileges, but it is up to your code to request those privileges of the user.

Gaining privileges

NN4+ comes with some Java classes that allow signed scripts and other signed objects to display the privilege request alert windows, and then turn on the privileges if the user clicks the "OK" or "Grant" button. A lot of these classes show up in the `netscape.security` package, but scripters only work directly with one class and three of its methods:

```
netscape.security.PrivilegeManager.enablePrivilege(["targetName"])
netscape.security.PrivilegeManager.revertPrivilege(["targetName"])
netscape.security.PrivilegeManager.disablePrivilege(["targetName"])
```

The `enablePrivilege()` method is the one that displays the security alert for the user. In NN4, the specific target named as a parameter influenced the details of the security alert message; for NN6+, the security alert is generic (and far less intimidating).

If the user grants the privilege, script processing continues with the next statement. But if the user denies access, then processing stops, and the `PrivilegeManager` class throws a Java exception that gets displayed as a JavaScript error message. Later in this chapter I show you how to gracefully handle the user's denial of access.

Enabling a privilege in JavaScript is generally not as risky as enabling a Java applet. The latter can be more easily hijacked by an alien class to piggyback on the trusted applet's privileges. Even though the likelihood of such activity taking place in JavaScript is very low, turning privileges off after the statement that requires privileges is always a good idea. Use the `revertPrivilege()` method to temporarily turn off the privilege; another statement that enables the same privilege target will go right ahead without asking the user again. Disable privileges only when the script requiring privileged access won't be run again until the page reloads.

Specifying a target

Rather than opening blanket access to all protected capabilities in one blow, the Netscape security model defines narrow capabilities that are opened up when privileges are granted. Each set of capabilities is called a target. Netscape defines dozens of different targets, but not all of them are needed to access the kinds of methods and properties available to JavaScript. You will likely confine your targets to the nine discussed here.

Because NN4's security alerts provided (at times excruciating) detail about the nature of the privilege being requested by the Web site, targets had various risk levels and categories. These concerns are less of an issue in NN6+, but they are provided here for your more complete understanding of the mechanisms beneath the Privilege Manager.

Each target has associated with it a risk level (low, medium, or high) and two plain-language descriptions about the kinds of actions the target exposes to code. This information appears in the NN4 security privilege dialog box that faces a user the first time a particular signature requests privileges. All of the targets related to scripted access are medium or high risk, because they tend to open up local hard disk files and browser settings.

Netscape has produced two categories of targets: primitive and macro. A *primitive target* is the most limited target type. It usually confines itself to either reading or writing of a particular kind of data, such as a local file or browser preference. A *macro target* usually combines two or more primitive targets into a single target to simplify the user experience when your scripts require multiple kinds of access. For example, if your script must both read and write a local file, it could request privileges for each direction, but the user would be presented with a quick succession of two similar-looking security dialog boxes. Instead, you can use a macro target that combines both reading and writing into the privilege. The user sees one security dialog box, which, in NN4, explains that the request is for both read and write access to the local hard disk.

Likely targets for scripted access include a combination of primitive and macro targets. Table 46-1 shows the most common script-related targets and the information that appears in the security dialog box.

For each call to `netscape.security.PrivilegeManager.enablePrivilege()`, you specify a single target name as a string, as in

```
netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserRead");
```

This specification allows you to enable, revert, and disable individual privileges as required in your script.

Table 46-1: Scripting-Related Privilege Targets

Target Name		
Risk	***Short Description***	***Long Description***
`UniversalBrowserAccess`		
High	Reading or modifying browser data	Reading or modifying browser data that may be considered private, such as a list of Web sites visited or the contents of Web forms you may have filled in. Modifications may also include creating windows that look like they belong to another program or positioning windows anywhere on the screen.
`UniversalBrowserRead`		
Medium	Reading browser data	Access to browser data that may be considered private, such as a list of Web sites visited or the contents of Web page forms you may have filled in.
`UniversalBrowserWrite`		
High	Modifying the browser	Modifying the browser in a potentially dangerous way, such as creating windows that may look like they belong to another program or positioning windows anywhere on the screen.

Continued

Table 46-1 *(continued)*

Target Name		
Risk	***Short Description***	***Long Description***
`Universal FileAccess`		
High	Reading, modifying, or deleting any of your files	This form of access is typically required by a program such as a word processor or a debugger that needs to create, read, modify, or delete files on hard disks or other storage media connected to your computer.
`Universal FileRead`		
High	Reading files stored in your computer	Reading any files stored on hard disks or other storage media connected to your computer.
`Universal FileWrite`		
High	Modifying files stored in your computer	Modifying any files stored on hard disks or other storage media connected to your computer.
`Universal PreferencesRead`		
Medium	Reading preferences settings	Access to read the current settings of your preferences.
`Universal PreferencesWrite`		
High	Modifying preferences settings	Modifying the current settings of your preferences.
`Universal SendMail`		
Medium	Sending e-mail messages on your behalf	

Blending Privileges into Scripts

The implementation of signed scripts in Navigator protects scripters from many of the potential hazards that Java applet and plug-in developers must watch for. The chance that a privilege enabled in a script can be hijacked by code from a Bad Guy is very small. Still, exercising safe practices in case you someday work with other kinds of signed objects is good practice.

Keep the window small

Privilege safety is predicated on limiting exposure according to two techniques. The first technique is to enable only the level of privilege required for the protected access your scripts need. For example, if your script only needs to read a normally protected `document` object property, enable the `UniversalBrowserRead` target rather than the wider `UniversalBrowserAccess` macro target.

The second technique is to keep privileges enabled only as long as the scripts need them enabled. If a statement calls a function that invokes a protected property, enable the privilege for that property in the called function, not at the level of the calling statement. If a privilege is enabled inside a function, the browser automatically reverts the privilege at the end of the function. Even so, if the privilege isn't needed all the way to the end of the function, you should explicitly revert it after you are through with the privilege.

Think of the users

One other deployment concern focuses more on the user's experience with your signed page. You should recognize that the call to the Java `PrivilegeManager` class is a LiveConnect call from JavaScript in NN4. Because the Java virtual machine does not start up automatically when Navigator 4 does, a brief delay occurs the first time a LiveConnect call is made in a session (the status bar displays "Starting Java..."). Such a delay may interrupt the user flow through your page if, for example, a click of a button needs access to a privileged property. Therefore, consider gaining permission for protected access as the page loads. Execute an `enablePrivilege()` and `revertPrivilege()` method in the very beginning. If Java isn't yet loaded into the browser, the delay is added to the other loading delays for images and the rest of the page. Thereafter, when privileges are enabled again for a specific action, neither the security dialog box nor the startup delay get in the way for the user.

Also remember that users don't care for security dialog boxes to interrupt their navigation. If your page utilizes a couple of related primitive targets, at the outset enable the macro target that encompasses those primitive targets. The user gets one security dialog box covering all potential actions in the page. Then let your script enable and revert each primitive target as needed.

Example

To demonstrate signed scripts in action, I show a page that accesses a typical target that allows the script to open an always-raised new window. No error checking occurs for the user's denial of privilege in this example. Therefore, if you experiment with this page (either with codebase principals turned on or signing them yourself), you will see the JavaScript error that displays the Java exception. Error detection is covered later in the chapter.

Accessing a protected window property

Listing 46-2 is a small document that contains one button. The button calls a function that opens a new window with the NN-proprietary `alwaysRaised` parameter turned on. Setting protected `window.open()` parameters in NN4+ requires the `UniversalBrowserWrite` privilege target. Inside the function, the privilege is enabled only for the creation of the new window. For this simple example, I do not enable the privilege when the document loads.

Listing 46-2: **Creating an alwaysRaised Window**

```
<html>
   <head>
      <title>Simple Signed Script</title>
      <script type="text/javascript" archive="myJar.jar" id="1">
      function newRaisedWindow() {
         netscape.security.PrivilegeManager.enablePrivilege(
            "UniversalBrowserWrite");
         var newWindow =
            window.open("","","HEIGHT=100,WIDTH=300,alwaysRaised");
         netscape.security.PrivilegeManager.disablePrivilege(
            "UniversalBrowserWrite");
```

Continued

Listing 46-2 *(continued)*

```
         var newContent = "<html><body><b>It\'s good to be the King!<\/b>";
         newContent += "<form><center><input type='button' value='OK'";
         newContent +=
            "onclick='self.close()'><\/center><\/form><\/body><\/html>";
         newWindow.document.write(newContent);
         newWindow.document.close();
      }
      </script>
   </head>
   <body>
      <b>This button generates an always-raised new window.</b>
      <form>
         <input type="button" value="New 'Always Raised' Window"
         onclick="newRaisedWindow()" id="2" />
      </form>
   </body>
</html>
```

Listing 46-2 has two script items that need signing: the `<script>` tag and the event handler for the button. Also, the `archive` attribute points to the `JAR` file that contains the script signature. Note that this example file is not signed, and therefore does not include a companion `JAR` archive on the CD-ROM.

Handling Privilege Manager Errors

The change between the ways NN4 and NN6+ allows scripts to intercept errors causes no small problem if you need to serve both browser versions. The primary reason you want to handle errors is that when a user denies access to advanced privileges, the `PrivilegeManager` generates an error. Although the error is not destructive in any way, and it appears only in the JavaScript Console window (NN4.5+), accounting for such factors is good coding practice. Unfortunately, the mechanism that works for NN4 doesn't work in NN6+; the mechanism that works in NN6+ cannot even be placed in a page that loads into NN4 without generating syntax errors. So, if you plan on still supporting NN4, you'll need to serve up different pages for NN4 and NN6+.

For NN4, you can define an `onerror()` function that looks for the specific error message thrown by the `PrivilegeManager` class through LiveConnect. That function looks like the following:

```
function onerror(msg, URL, lineNum) {
   var errorMsg = msg;
   if (msg.indexOf("ForbiddenTargetException") != -1) {
      errorMsg = "You have elected not to grant privileges to this script.";
   }
   alert(errorMsg);
   return true;
}
```

Of course, you don't have to display any message, but it may be a good place to advise users about what they're missing by not granting privilege.

For NN6+, you can use the native `try...catch` exception handling, which means that the calls to the `enablePrivilege()` method of the `PrivilegeManager` class must be wrapped inside a `try` block. The function from Listing 46-2 is modified as follows:

```
function newRaisedWindow() {
   try {
      netscape.security.PrivilegeManager.enablePrivilege(
         "UniversalBrowserWrite");
   }
   catch(err) {
      alert("You have elected not to grant privileges to this script.");
      return;
   }
   var newWindow = window.open("","","HEIGHT=100,WidTH=300,alwaysRaised");
    netscape.security.PrivilegeManager.disablePrivilege("UniversalBrowserWrite")
   var newContent = "<html><body><b>It\'s good to be the King!</b>";
   newContent += "<form><center><input type='button' value='OK'";
   newContent += "onclick='self.close()' /></center></form></body></html>";
   newWindow.document.write(newContent);
   newWindow.document.close();
   return;
}
```

Signed Script Miscellany

In this last section of the chapter, I list some of the more esoteric issues surrounding signed scripts. Three in particular are: 1) how to allow unsigned scripts in other frames, windows, or layers to access signed scripts; 2) how to make sure your signed scripts are not stolen and reused; and 3) special notes about international text characters.

Exporting and importing signed scripts

JavaScript provides an escape route that lets you intentionally expose functions from signed scripts for access by unsigned pages. If such a function contains a trusted privilege without careful controls on how that privilege is used, a page that is not as well intentioned as yours could hijack the trust.

The command for exposing this function is export. The following example exports a function named `fileAccess()`:

```
export fileAccess;
```

A script in another window, frame, or layer can use the import command to bring that function into its own set of scripts:

```
import fileAccess;
```

Even though the function is now also a part of the second document, it executes within the context of the original document, whose signed script governs the privilege. For example, if you exported a function that did nothing but enable a file access privilege, a Bad Guy who studies your source code could write a page that imports that function into a page that now has unbridled file access.

If you wish to share functions from signed scripts in unsigned pages loaded into your own frames or layers, avoid exporting functions that enable privileges. Other kinds of functions, if hijacked, can't do the same kind of damage as a privileged function can.

Locking down your signed pages

Speaking of hijacking scripts, it would normally be possible for someone to download your HTML and JAR archive files and copy them to another site. When a visitor comes to that other site and loads your copied page and JAR file, your signature is still attached to the scripts. Although this may sound good from a copyright point of view, you may not want your signature to appear as coming from someone else's Web server. You can, however, employ a quick trick to ensure that your signed scripts work only on your server. By embedding the domain of the document in the code, you can branch execution so that scripts work only if the file comes from your server.

The following script segment demonstrates one way to employ this technique:

```
<script type="text/javascript" archive="myPage.jar" id="1">
if (document.URL.match(/^http:\/\/www.myDomain.com\//)) {
   privileges statements execute only from my server
}
</script>
```

Even though this branching code is visible in the HTML file, the hash value of your code is saved and signed in the archive. If someone modifies the HTML, the hash value that is recalculated when a visitor loads the page won't match the JAR file manifest, and the script signature fails.

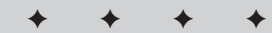
International characters

Although international characters are fine for HTML content, they should not be used in signed scripts. The problem is that international characters are often converted to other character sets for display. This conversion invalidates the signature, because the signed and recalculated hash values don't match. Therefore, do not put international characters in any signable script item. If you must include such a character, you can escape it or, in NN4, put such scripts in unsigned layers.

✦ ✦ ✦

CHAPTER 47

Cross-Browser Dynamic HTML Issues

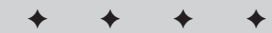


In This Chapter

Introducing Dynamic HTML

The common denominator of DHTML functionality across browsers

Upgrading to modern compatibility techniques



Version 4 browsers — NN4 and IE4 — were the first browsers to include World Wide Web technologies that gave page authors far more control over the display and interactive behavior of Web page content. Lumped together under the heading of Dynamic HTML (DHTML), these technologies dramatically extended the simple formatting of standard HTML that page authors had used for years. These days, scripters and designers coming to Web development for the first time take DHTML capabilities for granted; they are probably unaware that plain ol' HTML is little more than a specification to assign context to static text and images on a page.

A lot of what the user gets with DHTML had previously been accomplished only via Java applets and plug-ins, such as early versions of Shockwave (pre-Flash). Not that DHTML eliminates these technologies from the Web author's arsenal (DHTML doesn't do sound or video, for example), but because DHTML can accomplish much more of what authors look for in assembling page content and layout without the long downloads of applets or plug-in content, it becomes an attractive way for nonprogrammers to spice up Web applications.

Perhaps categorizing DHTML authors as "nonprogrammers" is not quite right. DHTML also adds significantly to the vocabulary required to incorporate dynamic content into pages. Suddenly HTML becomes a lot more programming than simply adding tags to existing content. And if you want to do dynamic positioning of elements, be prepared to put your JavaScript skills to use.

What Is DHTML?

You can practically find as many definitions of Dynamic HTML as there are people to ask. My definition covers a broad range, because DHTML is not really any one "thing." Instead it is an amalgam of several technologies, each of which has a standards effort in varying stages of readiness. The key technologies are as follows: Cascading Style Sheets (CSS); Document Object Model (DOM); and client-side scripting. It will help your authoring skills if you have a little historical perspective on how the Web arrived at DHTML.

For many years, the HTML standard was intended for the rendering of static content — not much more than an electronic version of a printed page. The most interactive part of a page was a form, which included buttons to click and text boxes to fill in. But for anything to change on the page, the content had to be served up again from the host computer.

Client-side scripting, as first implemented through JavaScript in NN2, opened the way for HTML pages to not only contain some "smarts," but also control individual pieces of content on the page without fetching a modified page from the server. At first, only form control elements were scriptable. Soon thereafter, images could be swapped, although the rectangular space for the image was fixed when the page loaded. More dynamism accrued to pages in NN4 by way of the layer, which acted like a borderless, transparent or opaque window that could contain its own HTML document content and be positioned anywhere on the page, including overlapping content on the main page or other layers. A layer's entire content could be modified without touching the rest of the page or other layers.

But the real breakthrough in dynamism came in IE4, whose rendering engine permitted any element to be modified, inserted, or removed on the fly, while the rest of the page reflowed its content quickly and automatically in response to the change. At the same time, an accepted standard for stylesheets (Cascading Style Sheets) opened the way for scripts to modify the look of content already on the page. Text could change colors when a cursor rolled atop it by either adjusting the stylesheet property associated with the text or changing the stylesheet rule that applies to the text.

Development activity at both Netscape and Microsoft eventually led to a standard for the Document Object Model as a way for scripts to control HTML content directly. Unfortunately, the browser makers frequently implemented first, and only then tried to establish their implementations as standards. Sometimes the implementations were not as complete as the standards became, leaving the browsers in states that only partially implement the standards, while paying homage to legacy implementations. Netscape used the occasion of developing an entirely new code base for what became the Mozilla browser family to try to sever some ties with the past. In many respects the Mozilla browser represents the state of the standard art as implemented so far. Newest versions of IE, on the other hand, must try to cater to both the legacy implementation and the standards, creating a massive DOM implementation with significant overlap in functionality with different syntaxes. Thus, the result of proprietary explorations and industry standards is a choice of modern browsers that permit a wide range of dynamic activity on content that reaches the browser. Browsers that had started life as sleepy renderers of a tiny HTML vocabulary have grown into powerful front ends for server applications, if not self-contained applications of a sort that execute entirely on the client computer.

Standards for CSS, DOM, and ECMA scripting have been well covered earlier in this book. The purpose of this chapter is to demonstrate approaches to accommodating the sometimes vast differences in specific implementations of these technologies (including browser-specific variations) to produce content that runs on as many DHTML-capable browsers as possible. Most of the problems, as you are well aware from Chapters 15 and 39, are caused by page authors trying to develop for essentially three different document object models: NN4, IE4+, and W3C DOM (as implemented in IE5+/NN6+/Mozilla/Safari).

Striving for Compatibility

With as many as three object models to support (you can, of course, elect to support only a subset of browsers if you like) you should look for ways to minimize your pain. If the NN4 object model is in your mix, you will very likely experience moments of sheer torture, as you try to get even the CSS-supported HTML to behave as it does in browsers of the other object models. Thankfully, the NN4 browser's installed base is quickly becoming a remnant of the past.

Two keys to survival are among the object models: knowing each DOM's limitations and finding common denominators.

In the area of DHTML limitations, NN4 is the clear winner. Compared to the automatic content reflowing of IE4+ and W3C DOM browsers, the NN4 object model is painfully static. For example, dynamically changing the color of a chunk of text in response to a rollover is a difficult task in NN4 requiring the careful positioning of a layer atop main page text; and making any inline modification to content (other than swapping an image of the same size) is completely out of the question. Between the IE4+ and W3C DOMs, the biggest differences fall more along operating system and browser brand lines. Microsoft takes advantage of the integration of the IE browser and the Windows operating system to such an extent that it can provide IE services that work only on Windows versions of IE. MacIE users are out of luck (for text filters, for instance), as are users of any other browser brand.

Looking for areas of commonality — or at least gaining a clear understanding of where the models diverge — can be a tedious, yet personally rewarding pursuit. For example, one of the biggest problems facing designers for all three DOMs is the way scripts must reference elements that are to be moved or hidden (something that all three object models can do). NN4 requires references that take the layer object structure into account; IE4+ has the Microsoft syntax of `document.all`, which provides a reference avenue to any element whose `id` attribute is set; and the W3C DOM (as implemented in IE5+/NN6+/Mozilla/Safari) users a finger-twisting (albeit now industry standard) `document.getElementById()` method to obtain a reference to any ID'd element.

As soon as your script has a valid reference to an element, the next step is to read or write some property, or invoke some method of that object that governs the element's position (and possibly other style) attributes. Here, again, the object models diverge, but not quite as severely. NN4 has a singular implementation that provides properties and methods of positioned elements (`layer` objects) directly; the IE4+ and W3C DOMs, on the other hand, work their positioning magic through the `style` property of a positioned element. In some cases the "last-dot" property names are identical across all three models (for example, `document.myLayer.zIndex`, `document.all.myLayer.style.zIndex`, and `document.getElementById("myLayer").style.zIndex`). Building a reference to reach that last dot, though, is where some of your hard work must go.

Each DOM also has its own event model. Whereas IE5+ overlaps its DOM features with both the IE4+ and to some extent the W3C DOM, the event models don't follow the same lines of implementation. As of WinIE6 and MacIE5, IE does not implement any of the W3C DOM event model, although Mozilla-based browsers do.

The bottom line, then, is letting your scripts decide how to perform actions based on the browser version is not a good idea. Instead, the scripts should be smart enough to act based on the capabilities of the browser that is currently running the script. As you see in the rest of this chapter, it is possible to develop fairly sophisticated DHTML into a page and make it work with all three DOMs without one iota of browser version detection.

Working Around Incompatibilities

To create DHTML for multiple DOMs, you must find ways to accommodate incompatible object references and occasionally incompatible property names. Scripting gives you several alternatives to working your way around these potential problems. Some of the approaches you can take are now passe, but they are described here partly for the sake of historical reference, but also because you will see many instances of these approaches taken in legacy DHTML applications from the days when authors had to worry about only two DOMs (NN4 and IE4). The real "meat" of this discussion comes later, when you learn more about object detection and custom APIs.

Old-fashioned compatibility tricks

In a simpler time (until late 2000), it was possible to write cross-browser DHTML applications that had to run on only two classes of browser: NN4 and IE4. Two approaches to writing code for these two DOMs grew in popularity: inline branching and platform equivalency. They are described here, not for you to apply, but for you to understand what the pioneers did, in case you encounter their code in your Web surfing.

Inline branching

The idea behind inline branching is that your scripts will use `if...else` decisions to execute one branch of code for one browser and another branch for the other browser. Before you can begin to write code that creates branches for each browser, you should define two global variables at the top of the page that act as Boolean flags for your `if...else` constructions later. Therefore, at the first opportunity for a `<script>` tag in a page, include the following code fragment to set flags named `isNav4` and `isIE4`:

```
var isNav4, isIE4;
if (parseInt(navigator.appVersion) == 4) {
   if (navigator.appName == "Netscape") {
      isNav4 = true;
   } else if (navigator.appVersion.indexOf("MSIE") != -1) {
      isIE4 = true;
   }
}
```

Version checking here is quite specific. First of all, it intentionally limits access to browsers whose versions come back as version 4. This code, written when the browsers were still at version 4, was remarkably prescient. My concern at the time was that DHTML was so volatile that it was unknown if future browser versions would be backward compatible with the code to be run inside branches governed by the two global variables. As it turned out, NN6 (whose `navigator.appVersion` reports 5) is not backward compatible with the layer structure of NN4, so that locking the NN4 branches to NN4 became a good thing. On the IE side, the `navigator.appVersion` property continues to report 4, even through IE6, which is backward compatible

with IE4. Thus, any branch dedicated to IE4 executes under this scheme and remains syntactically accurate.

Another aspect of the flag-setting script I should mention is that the example provides no escape route for browsers that aren't level 4 or aren't either Navigator or Internet Explorer (should there be a level 4 browser from another brand). In a production environment, I would either prefilter access to the page or redirect ill-equipped users to a page that explains why they can't view the page. In the structure of the preceding script, redirection would have to be made in two places, as follows:

```
var isNav4, isIE4;
if (parseInt(navigator.appVersion) == 4) {
   if (navigator.appName == "Netscape") {
      isNav4 = true;
   } else if (navigator.appVersion.indexOf("MSIE") != -1) {
      isIE4 = true;
   } else {
      location = "sorry.html";
   }
} else {
   location = "sorry.html";
}
```

Later in this chapter, I discuss the issue of designing DHTML pages that degrade gracefully in pre-DHTML browsers.

With the global variables defined in the document (and unsupported browsers redirected elsewhere), you can use them as condition values in branching statements that address an object according to the reference appropriate for each platform. For example, to change the visibility property of an object named `instructions`, you use the flags as follows:

```
if (isNav4) {
   document.instructions.visibility = "hidden";
} else {
   document.all.instructions.style.visibility = "hidden";
}
```

As the browser DOMs evolve, expand, and fragment, inline branching becomes increasingly less practical. With so many permutations of DOM according to browser brand, browser version, and operating system, you can drive yourself crazy trying to accommodate them all and maintain the code going forward. This approach also eliminates from consideration any non-NN or non-IE browser (such as Opera), which may have the capabilities needed to play your DHTML scripts. This approach also limits the possibility that future browsers with higher `navigator.appVersion` values can take advantage of your scripts.

Platform equivalency

Another technique attempts to limit the concern for the different ways each platform refers to a positionable element (because cross-browser DHTML is pretty much limited to the properties affecting positionable elements). If you examine the formats for each platform's object references, you see that all formats contain a reference to the `document` and to the object name or ID. The IE4+ DOM syntax also includes property words, such as `all` and `style`. If you assign these extra property names to variables for IE4 and leave those variables as empty strings for NN4, you can assemble an object reference for those two platforms in one statement.

To begin using this technique, set two global variables that store reference components for the scope (`all` in IE4) and the `style` object (`style` in IE4):

```
var range = "";
var styleObj = "";
if (parseInt(navigator.appVersion) == 4) {
   if (navigator.appVersion.indexOf("MSIE") != -1) {
      range = "all.";
      styleObj = ".style";
   }
}
```

From this point, you can assemble an object reference with the help of the JavaScript `eval()` function, as follows:

```
var instrux = eval("document." + range + "instructions" + styleObj);
instrux.visibility = "hidden";
```

Or, you can use the `eval()` function to handle the entire property assignment in one statement, as follows:

```
eval("document." + range + "instructions" + styleObj +
   ".visibility = 'hidden'");
```

If your page does not have a lot of objects that your scripts will be adjusting, you can use this platform equivalency approach to create global variables holding references to your positionable objects at load time (triggered by the `onload` event handler so that all objects exist and can be referenced by the `eval()` function). Then, use those variables for object references throughout the scripts.

Unfortunately, the platform equivalency methodology breaks down when an NN4 layer object is nested inside another layer. The platform equivalency formulas assume that each object is directly addressable from the outermost `document` object. If your objects have a variety of nested locations, you can use either the inline branching method described earlier, or batch-assign objects to global variables at load time using platform branching techniques along the lines of the following example:

```
var instrux;
function initObjectVars() {
   if (isNav4) {
      instrux = document.outerLayer.document.instructions;
   } else {
      instrux = document.all.instructions.style;
   }
}
```

As soon as the variable contains a valid reference to the object for the current platform, your scripts can treat the object without further concern for platform when addressing properties that have the same name in both platforms:

```
instrux.visibility = "hidden";
```

The nested layer situation is not the only potential problem for the platform equivalency approach. In fact, the W3C DOM format for referencing objects (using the `document.getElementById()` method) makes for some hair-raising string assembly and global variable assignment. Another truly negative aspect is the frequent usage of the `eval()` function. As mentioned in Chapter 34, this function is a performance speed thief.

Modern approaches to compatibility

Although in-line branching and platform equivalency were suitable for their generations, the profusion of browser versions calls for better approaches to simplifying authoring for multiple DOMs. "Browser sniffing" is no longer a workable strategy except in rare instances. Techniques more suitable for today — object detection and custom APIs — are not really new. But these techniques are the preferred way to build cross-browser scripts with an eye to compatibility both backward and forward.

Object detection

The subject of object detection has been mentioned in several places in earlier chapters of this book. The technique has been used for a long time to let a browser not equipped to handle image objects gracefully skip over image swapping script segments:

```
if (document.images) {
   // statements to work with image objects
}
```

If there is no `document.images` property for a browser, the condition evaluates to `undefined`, which the condition treats as being `false`.

But object detection has also been misused in the past, especially in the DHTML realm, to substitute for browser version detection. For example, if a browser supported the `document.all` collection, a global variable was set to indicate that the browser was IE4 or later; the existence of `document.layers` supposedly meant that the browser was NN4. While both of those assertions are true (as of the browsers released so far), it was a mistake to link a browser version with the existence of an object or property. Instead, object detection should be used only if your script statements will be addressing that object, just as the `document.images` condition does in the previous example.

To demonstrate this tactic, consider the need to assemble a reference to an object so that it is ready to have one of its DHTML properties adjusted. Each of the three DOMs has its own syntax for assembling such a reference, and each syntax relies on the existence of a particular object or property. The function shown in Listing 47-1 (not by itself, but included in Listing 47-2) lets you pass the name or ID of a positioned element (either in string form or object form) to receive back a valid reference to the object with which style-related properties are associated — all without resorting to the `eval()` function in any form:

Listing 47-1: **Using Object Detection to Assemble an Element Object Reference**

```
function getObject(obj) {
   var theObj;
   if (document.layers) {
      if (typeof obj == "string") {
         // just one layer deep
         return document.layers[obj];
      } else {
         // can be a nested layer
         return obj;
      }
   }
```

Continued

Listing 47-1 *(continued)*

```
   if (document.all) {
      if (typeof obj == "string") {
         return document.all(obj).style;
      } else {
         return obj.style;
      }
   }
   if (document.getElementById) {
      if (typeof obj == "string") {
         return document.getElementById(obj).style;
      } else {
         return obj.style;
      }
   }
   return null;
}
```

The primary object detection for each of the three sections of this function looks for the presence of categories of objects (`document.layers` and `document.all`) or a particular method (`document.getElementById()`), and then — this is the important part — the script uses those detected objects in the statements. The script doesn't know IE4 from NN7; it *does* know how to derive valid references for three different object models, and employs the syntax of the first one for which the associated object property or method is supported.

In practice, the order of the three sections should have no bearing on your scripts, but you should be aware of one subtlety: IE5+ can work with either of the last two sections, because those browsers detect `document.all` and `document.getElementById` as valid references. If you were to switch the position of the last two sections, IE5+ would be using W3C DOM terminology. The results, however, are the same: A valid reference to the `style` object associated with an element.

Custom APIs

Notions of object detection and simplifications of your scripts come together in the final approach to building cross-browser DHTML: Writing a custom API (Application Programming Interface). A JavaScript custom API is a library of functions you design to act as an intermediary between your scripts and other scriptable entities. Ideally, an API simplifies access to, or control of, other entities. In the context of designing a cross-browser DHTML page, an API can offer a single function that smoothes over the differences in object references and/or property names among several platforms. Your custom function provides a single access point that is consistent across all platforms. In essence, you are creating your own metavocabulary for methods and property settings.

The element object reference maker in Listing 47-1 is a good start for such an API, because all other functions for moving, hiding, showing, and changing the stacking order of a positionable element need a valid style-oriented reference to the element. Now look at a function from an API whose job is to alter the stacking order of a positionable element:

```
// set the z-order of an object
function setZIndex(obj, zOrder) {
   var theObj = getObject(obj);
   theObj.zIndex = zOrder;
}
```

Your main page script would use the ID of the positioned `div` element as the first parameter to this function, with an integer indicating the value that would be assigned to the element's stylesheet `z-Index` attribute:

```
setZIndex("myLayer", 100);
```

All of the branching for the various DOMs in this function is done in the `getObject()` function (Listing 47-1), which returns the valid reference for whichever of the three supported DOMs is running the script. All three DOMs, it turns out, have the same `zIndex` property representing the `z-Index` style attribute, so that no further branching is needed here.

As one more example, the next API function offers an interface to incompatible ways of adjusting the location of a positionable element. In this case, the act of moving an element has different syntax in different DOMs. One group (NN4 for layers) uses the `moveTo()` method; the rest support `left` and `top` properties of their `style` object:

```
// position an object at a specific pixel coordinate
function shiftTo(obj, x, y) {
   var theObj = getObject(obj);
   if (theObj.moveTo) {
      theObj.moveTo(x,y);
   } else if (typeof theObj.left != "undefined") {
      theObj.left = x + "px";
      theObj.top = y + "px";
   }
}
```

Notice one workaround, which, on the surface, isn't pretty: The second branch must perform an odd way of object detection. We're stuck with having to make a tradeoff when it comes to checking for the existence of a style property. If the page uses stylesheets defined in `<style>` tags (or imported into the page from external stylesheet files), the element affected by the rule does not yield the rule's property values through the element's `style` property. The property exists, but its value in this case (or until it is set by script) is an empty string. IE5+ provides a `currentStyle` property to give us the effective values, but that property is not (yet) a part of the DOM standard. It's true that you can derive the effective style property of an element (see Chapter 26), but even if you assign the stylesheet via the element's `style` attribute (in which case the style property values come through), detecting the presence of the property with the conditional expression

```
if (theObj.left)
```

is not practical here anyway. If the effective value of the `left` and `top` properties were an empty string (or zero for a numeric style property value), the conditional expression would evaluate to the equivalent of `false`, making it appear as though the property doesn't exist. To validate the existence of the property, the conditional expression verifies that the value of a named property has a type other than "undefined." It may seem like a long way to go to prove the existence of a property, but it works, even if the value is an empty string or zero.

It is important that both branches perform object detection. Although it is unlikely (but, as we learned from the transition between NN4 and NN6, not impossible), if a future browser should completely alter its vocabulary, omitting the objects being detected here, the function ends gracefully, without generating script errors.

An API is usually best deployed as an external `.js` file. One such API file is described later in this chapter. Bear in mind, however, that a lengthy API gets downloaded to the browser, no matter how much or how little of it your main scripts use. Blindly linking in a big library just to use a few of its functions is a mistake. You serve your users better if you create a subset of the API, and link the subset to the page (or drop the few functions directly into the page's scripts if the combination is not reused on a lot of pages).

Handling non-DHTML browsers

An important question to ask yourself as you embark on a DHTML-enhanced page is how you intend to treat visitors whose browsers aren't up to the task. In many respects the problem is similar to the problem of treating nonscriptable browsers when your page relies on scripting (see Chapter 13).

The moment your page uses DHTML to position an element, you must remember that non-DHTML browsers display the content according to traditional HTML rendering rules. No elements are allowed to overlap. Any block-level tag is rendered at the left margin of the page, unless some other non-DHTML alignment (center or right) is at work. This goes for elements that you design to be DHTML-positioned to sit offscreen (perhaps with a clickable tab) until called by the user. An element defined as being hidden or not displayed in DHTML will be visible. In most cases, your carefully designed DHTML page will look terrible.

However, a page that does not use too radical a layout strategy may still be usable in non-DHTML browsers. You should always check your DHTML-enabled page in an older browser to see how it looks. Perhaps there isn't too much you need to do to degrade the DHTML so that the page is acceptable in older browsers.

The ultimate responsibility for deciding your compatibility strategy with older browsers rests with you and your perceptions about your page visitors. If they are in need of vital information from your site and that information is readable in non-DHTML browsers, that may be enough. Otherwise, you must provide a separate content path for both levels of browsers, much as you may be doing for scriptable versus nonscriptable browsers.

A DHTML API Example

Now it's time to get to a real DHTML API that you can use and build upon for your own applications. Listing 47-2 contains the API code, which is most likely to be deployed as an external `.js` library file. In fact, this API is used as-is in a map puzzle game application in Chapter 56. You can see there how it is used to control element positioning, dragging, and layering for the three DOM families. The code in Listing 47-2 is longer than most listings in this book, so for your convenience, I interlace commentary amid the long listing.

No global variables are needed for this API. Because all browser branching is performed via object detection, there is no need for browser version detection. Instead, the library starts with the `getObject()` function shown earlier in this chapter. Virtually every other function in this library makes a trip to `getObject()` to convert the name of the object passed as a parameter to an object reference whose positionable (or other style-related property) can be adjusted.

Listing 47-2: **The Custom API (DHTMLapi.js)**

```
// convert object name string or object reference
// into a valid object reference ready for style change
function getObject(obj) {
   var theObj;
   if (document.layers) {
      if (typeof obj == "string") {
         return document.layers[obj];
      } else {
         return obj;
      }
   }
   if (document.all) {
      if (typeof obj == "string") {
         return document.all(obj).style;
      } else {
         return obj.style;
      }
   }
   if (document.getElementById) {
      if (typeof obj == "string") {
         return document.getElementById(obj).style;
      } else {
         return obj.style;
      }
   }
   return null
}
```

A pair of functions handles all motion of positionable elements. The first function, `shiftTo()` takes three parameters: the ID of the object being moved, and the horizontal and vertical pixel coordinates of the top-left corner of the element. The assumption is that the main page script that invokes this library function performs the calculation of the coordinates. You see that code in Chapter 56. Branches inside this function handle the NN4 `layer.moveTo()` method or the setting of `style` properties for other DOMs. In these other browsers, moving the element requires adjusting two positional properties, left and top. Even though the adjustments are made in separate statements, the action on the screen does not follow the action statement-by-statement. Between screen buffering and quick execution, the repositioning appears as a single shift.

```
// position an object at a specific pixel coordinate
function shiftTo(obj, x, y) {
   var theObj = getObject(obj);
   if (theObj.moveTo) {
      theObj.moveTo(x,y);
   } else if (typeof theObj.left != "undefined") {
      theObj.left = x + "px";
      theObj.top = y + "px";
   }
}
```

The `shiftBy()` function mimics NN4's `layer.moveBy()` method. The second and third parameters represent the number of pixels that the object should be moved on the page. A positive number means to the right or down; a negative number means to the left or up; a value of zero means no change to the axis. For NN4, the script uses the `layer.moveBy()` method. But for the rest, the passed values are added to the `left` and `top` properties. Notice that because these properties return strings that include the units for the measurements, the incremental values are added to integer extractions from the current settings. And because the units being used here are the default (pixels), no units have to be assigned with the new values (although they could without penalty).

```
// move an object by x and/or y pixels
function shiftBy(obj, deltaX, deltaY) {
   var theObj = getObject(obj);
   if (theObj.moveBy) {
      theObj.moveBy(deltaX, deltaY);
   } else if (typeof theObj.left != "undefined") {
      theObj.left = parseInt(theObj.left) + deltaX + "px";
      theObj.top = parseInt(theObj.top) + deltaY + "px";
   }
}
```

Both platforms use the same property name for setting the stacking order of positionable thingies. Therefore, the `setZIndex()` function does little more than convert the object reference and assign the incoming value to the `zIndex` property.

```
// set the z-order of an object
function setZIndex(obj, zOrder) {
   var theObj = getObject(obj);
   theObj.zIndex = zOrder;
}
```

NN4 and browsers with `style` objects have their own way of referring to the background color. The `setBGColor()` function applies the correct syntax based on whichever property is detected in the object.

```
// set the background color of an object
function setBGColor(obj, color) {
   var theObj = getObject(obj);
   if (theObj.bgColor) {
      theObj.bgColor = color;
   } else if (typeof theObj.backgroundColor != "undefined") {
      theObj.backgroundColor = color;
   }
}
```

Allowable values for the `visibility` property are very unprogrammatic in my opinion. I expect a Boolean value rather than strings. To accede to reality while making the process of showing and hiding elements more logical to me, I created API functions called `show()` and `hide()`.

```
// set the visibility of an object to visible
function show(obj) {
   var theObj = getObject(obj);
   theObj.visibility = "visible";
}

// set the visibility of an object to hidden
function hide(obj) {
   var theObj = getObject(obj);
   theObj.visibility = "hidden";
}
```

Although the `left` and `top` properties of NN4 layers do not include unit values, it is still safe to use `parseInt()` on the values returned from the properties, whether they be retrieved in NN4 or browsers that have `style` objects (whose properties return units). The need for these API functions came from the way the map puzzle application in Chapter 56 works. For a couple of operations, it calculates the destination for an object with respect to the position of one of the other positioned elements. These functions return the values needed for the main program's calculation. This is also an example of how you may need to embellish the API for your own application.

```
// retrieve the x coordinate of a positionable object
function getObjectLeft(obj)  {
   var theObj = getObject(obj);
   return parseInt(theObj.left);
}

// retrieve the y coordinate of a positionable object
function getObjectTop(obj)  {
   var theObj = getObject(obj);
   return parseInt(theObj.top);
}
```

The previous API is generalizable enough to be used as a library with any cross-platform DHTML application using positioning. The API can even be used with a platform-specific page. It is more efficient, however, to use a browser's native objects, properties, and methods if you know for sure that users will have only one brand of browser.

✦ ✦ ✦

CHAPTER 48

Internet Explorer Behaviors

In This Chapter

Introducing IE behaviors

Understanding the structure of behavior XML files

Exploring behavior samples

Internet Explorer 5 for Windows was the first browser to deploy what Microsoft calls *behaviors*. Microsoft and others have proposed the behaviors concept to the W3C, and it could some day become one of the W3C standard recommendations. Such a standard might not be implemented exactly the way Microsoft currently implements behaviors, but most of the concepts are the same, and the syntax being discussed so far is similar. While there is no guarantee that the W3C will adopt behaviors as a standard, you will see that the concept seems to be a natural extension to the work that has already been adopted for both CSS and XML.

The W3C effort is called Behavioral Extensions to CSS. For the latest document describing the work of the participants of the standards discussions, visit `http://www.w3.org/TR/becss`.

Stylesheets for Scripts

You can best visualize what a behavior is in terms of the way you use stylesheets. Consider a stylesheet rule whose selector is a tag or a class name. The idea behind the stylesheet is that one rule, which can define dozens of rendering characteristics of a chunk of HTML content, can be applied to perhaps dozens, if not hundreds, of elements within the document. A corporation may design a series of rules for the way its Web documents will look throughout the Web site. If the designer decides to alter the font family or color for, say, `h1` elements, that change is made in one place (the external stylesheet file), and the impact is felt immediately across the entire site. Any page that includes an `h1` element renders the header with the newly modified style.

Imagine now that instead of visual styles associated with an element, you want to define a *behavioral style* for a particular group of elements. A behavioral style is the way an element responds to predominantly user interaction with the element. For example, if the design specifications for a Web site indicate that all links should have their text colored a certain way when at rest, but on mouse rollovers, the text color changes to a more contrasting color, the font weight increases to bold, and the text becomes underlined. Those modifications require scripts to change the style properties of the element in response to the mouse action of the user. The scripts that fire in response to specific user actions (events) are written in an external file known as a behavior, and a behavior is associated with an element, class, or tag through the same CSS syntax that you use for other style properties.

A behavior, of course, assumes that its scripts can work with whatever HTML element is associated with the behavior. Just as it would be illogical to associate the `tableLayout` style property with an element that wasn't a `table`, so, too, would it be illogical to associate a behavior, whose scripts employed `table` object properties and methods, to a `p` element. Even so, a well-designed behavior can obtain details about the element being manipulated through the element object's properties. The better you are at writing generalizable JavaScript functions, the more successful you will be in implementing behaviors.

Embedding Behavior Components

IE treats each behavior as a component, or add-on building block for the browser. IE5+ comes equipped with a handful of behaviors built into the browser (the so-called default behaviors, which happen to rely on specific XML elements embedded in a document). Behaviors that you create most likely exist as separate files on the server, just like external `.css` and `.js` files do. The file extension for a behavior file is `.htc` (standing for HTML Component).

Linking in a behavior component

To associate a behavior with any single element, class of elements, or tag as the page loads, use CSS rule syntax and the IE-specific `behavior` attribute. The basic syntax is as follows:

```
selector {behavior:url(componentReference)}
```

As with any stylesheet rule, you can combine multiple rule properties, delimiting them with semicolons. The format of the *componentReference* depends on whether you are using one of the IE default behaviors or a behavior you've written to an external file. For default behaviors, the reference is in the format:

```
#default#componentName
```

For example, if you want to associate the `download` behavior with any element of class `downloads` use:

```
.downloads {behavior:url(#default#download)}
```

Relative or absolute URIs to external `.htc` files can also be specified. For example, if your site contains a directory named `behaviors` and a file named `hilite.htc`, the stylesheet rule from the root directory is:

```
.hiliters {behavior:url(behaviors/hilite.htc)}
```

As with all Cascading Style Sheet rules, behaviors can be specified in a `style` element of the page, in the `style` attribute of an individual element, or in a rule defined inside an imported `.css` file.

Enabling and disabling behaviors

In Chapter 15, you can find details of WinIE5+ methods for all HTML elements that let scripts manage the association of a behavior with an element after the page has loaded. Invoking the `addBehavior()` method on an element assigns an external `.htc` file to that element. When you no longer need that behavior associated with the element, invoke the `removeBehavior()` method.

Component Structure

An .htc behavior file is a text file consisting of script statements inside a <script> tag set and some special XML tags that WinIE5+ knows how to parse. You create .htc files in the same kind of plain text editor that you use for external .js or .css files.

Script statements

Unlike external .js files, an .htc behavior file includes <script> tags, which surround any JavaScript (or VBScript, if you like) statements that control the behavior. Because a behavior most typically is written to control one or more aspects of the HTML element to which it is connected, statements tend to operate only on the associated object element. A special reference — element — is used to refer to the element object itself (much like the way the this keyword in a custom object's method self-refers to the object associated with the method).

If your behavior will be modifying either the content or style of the element, use the element reference as a foundation to the reference to one of that element object's properties or methods. For example, if a statement in a behavior needs to set the style.visibility property so that the element hides itself, the statement in the behavior script is:

```
element.style.visibility = "hidden";
```

Any valid reference from the point of view of the element object is fair game, including references to the element's parentElement, even though the parent element is not explicitly associated with the behavior.

Variable scope

Except for the special element reference, script content of a behavior is completely self-contained. You can define global variables in the behavior that are accessible to any script statement in the behavior. But a global variable in a behavior does not become a global variable for the main document's scripts to use. You can expose variables so that scripts outside of the behavior can get to them (as described later in the chapter), but this exposure is not automatic.

Most of the script content of a behavior consists of functions that usually interact in some fashion with the associated element (via the element's properties and/or methods). Local variables in functions have the same scope and operate just like they do in regular script functions. Global variables you define in a behavior, if any, are usually there for the purpose of preserving values between separate invocations of the functions.

Assigning event handlers

Functions in a behavior are triggered from outside the behavior through two means: event handlers and direct invocation of functions declared as public (described in the next section). Event handler binding is performed in a way that is not used elsewhere in the IE4+ DOM. Each event type (for example, onmouseover, onkeypress) requires its own special XML tag at the top of the behavior file. The format for the event handler tag is as follows:

```
<public:attach event="eventName" onevent="behaviorFunctionName()" />
```

As the behavior loads, the `public:attach` tag instructs the browser to expose to the "public" (that is, the world outside of the behavior) an event type (whose name always begins with the "on" prefix in the IE4+ event model); whenever an event of that type reaches the behavior's element, the function (defined within the behavior file) is invoked. In XML terminology, the `public:` part of the tag is known as a *namespace*, and IE includes a built-in parser for the `public` namespace. Notice, too, the XML syntax at the end of the tag that allows a single set of angle brackets to act as a start and end tag set (there is no content for this tag, just the attributes and their values).

To demonstrate, imagine that a behavior has a function named `underlineIt()`, which sets the `element.style.textDecoration` property to underline. To get the element to display the underline decoration as the user rolls the mouse atop the element, bind this function to the element's `onmouseover` event handler as follows:

```
<public:attach event="onmouseover" onevent="underlineIt()" />
```

If you compare the wording of the opening part of the tag, you may recognize a connection to the IE4+ event model's `attachEvent()` method of all HTML elements (see Chapter 15). You can have as many event binding tags as your element needs. To invoke multiple functions in response to a single event type, simply add the subsequent function invocation statements to the `onevent` attribute, separating the calls by semicolons (the same as with regular JavaScript statement delimiters).

Exposing properties and methods

XML tags with the `public:` namespace are also used (with different attributes) to expose a behavior's global variables as properties of the element and a behavior's functions as methods of the element. The syntax for both types of "public" announcements is as follows:

```
<public:property name="globalVarName" />
<public:method name="functionName" />
```

Values for both items are string versions of references to the variable and function (no parentheses). Again, you can define as many properties and methods for a behavior as you need.

As soon as a property and/or method is made public in a behavior, scripts from outside the behavior can access those items as if they were properties or methods of the element associated with the behavior:

```
document.all.elementID.behaviorProperty
document.all.elementID.behaviorMethod()
```

If you associate a behavior with a stylesheet class selector, and several document elements share that class name, each one of those elements gains the public properties and methods of that behavior, accessible through references to the individual elements. That's because a behavior's scripts are written to read or modify properties of whatever element receives a bound event or is referenced along the way to the public property or method.

Behavior Examples

The two following examples are intentionally simple to help you grasp the concepts of behaviors if they are new to you. The first example interacts with multiple elements strictly through event binding; the second example exposes a property and method that the main page's scripts access to good effect.

Example 1: Element dragging behavior

This book contains several examples of how to script a page to let a user drag an element around the browser window (Chapters 39 and 56 in particular). In all those examples, the dragging code and event handling was embedded in some fashion into the page's scripts. The first example of a behavior, however, drives home the notion of separating an element's behavior from its content (just as a CSS2 style sheet separates an element's appearance from its content).

Imagine that it's your job to design a page that employs three draggable elements. Two of the elements are images, while the third is a panel layer that also includes a form. If you haven't scripted DHTML before, this may sound like a daunting task at first, one rife with the possibility of including multiple versions of the same scripts to accommodate different kinds of draggable elements.

Now imagine that to the rescue comes a scripter who has built a behavior that takes care of all of the dragging scripting for you. All you do is assign that behavior by way of one attribute of each draggable element's stylesheet rule. Absolutely no other scripting is required on the main page to achieve the element dragging.

Listing 48-1 shows the behavior file (`drag.htc`) that controls basic dragging of a positionable element on the page. You may recognize some of the code as an IE4+ version of the cross-browser dragging code used elsewhere in this book (for a blow-by-blow account of these functions, see the description of the map puzzle game in Chapter 56). The names of the three operative functions and the basic way they do their jobs are identical to the other dragging scripts. Event binding, however, follows the behavior format through the XML tags. All interaction with the outside world occurs through the "public" event handlers.

Listing 48-1: **An Element Dragging Behavior**

```
<public:attach event="onmousedown" onevent="engage()" />
<public:attach event="onmousemove" onevent="dragIt()" />
<public:attach event="onmouseup"   onevent="release()" />
<public:attach event="onmouseover" onevent="setCursor()" />
<public:attach event="onmouseout"  onevent="release();restoreCursor()" />

<script type="text/javascript">
// global declarations
var offsetX = 0;
var offsetY = 0;
var selectedObj;
var oldZ, oldCursor;

// initialize drag action on mousedown
function engage() {
   selectedObj = (element == event.srcElement) ? element : null;
   if (selectedObj) {
      offsetX = event.offsetX - element.document.body.scrollLeft;
      offsetY = event.offsetY - element.document.body.scrollTop;
      oldZ = element.runtimeStyle.zIndex;
      element.style.zIndex = 10000;
      event.returnValue = false;
   }
}
```

Continued

Listing 48-1 *(continued)*

```
// move element on mousemove
function dragIt() {
   if (selectedObj) {
      selectedObj.style.pixelLeft = event.clientX - offsetX;
      selectedObj.style.pixelTop = event.clientY - offsetY;
      event.cancelBubble = true;
      event.returnValue = false;
   }
}

// restore state on mouseup
function release() {
   if (selectedObj) {
      selectedObj.style.zIndex = oldZ;
   }
   selectedObj = null;
}

// make cursor look draggable on mouseover
function setCursor() {
   oldCursor = element.runtimeStyle.cursor;
   element.style.cursor = "hand";
}

// restore cursor on mouseout
function restoreCursor() {
   element.style.cursor = oldCursor;
}
</script>
```

Notice a subtlety in Listing 48-1 that is implied by the element-specific scope of a behavior. Two statements in the `engage()` function need to reference scroll-related properties of the `document.body` object. Because the only connection between the behavior and the document is via the `element` reference, that reference is used along with the `document` property (a property of every HTML element object in IE4+, as shown in Chapter 15). From there, the `body` object and the required properties can be accessed.

Listing 48-2 is a simple page that contains three elements that are associated with the `drag.htc` behavior through a stylesheet rule definition (for the `draggable` class). The document is incredibly uncomplicated. Even the `drag.htc` file isn't very big. But together they produce a far more interesting page for the user than a couple of static images and a form.

Listing 48-2: **Three Draggable Elements Using the Behavior**

```
<html>
   <head>
      <title>IE5+ Behavior Demo (Dragging)</title>
      <style type="text/css">
      .draggable {position:absolute; behavior:url(drag.htc)}
      #img1 {left:150px; top:150px}
      #img2 {left:170px; top:170px}
```

```
      #txt1 {left:190px; top:190px; background-color:aqua; width:150px;
         height:50px; text-align:center}
      </style>
   </head>
   <body>
      <h1>IE5+ Behavior Demo (Dragging)</h1>
      <hr />
      <img alt="image" class="draggable" id="img1" src="cpu1.gif" /> <img
      alt="image" class="draggable" id="img2" src="desk3.gif" />
      <div class="draggable" id="txt1">
         A form inside a div element.
         <form>
            <input type="button" value="Does Nothing" />
         </form>
      </div>
   </body>
</html>
```

Obviously, the dragging example here is very rudimentary. It isn't clear from the sample code what the user gets from the page, other than the joy of moving things around. If you were designing an application that genuinely benefits from draggable objects (for example, the map puzzle in Chapter 56), you can easily enhance the behavior to perform actions, such as snapping a dragged element into place when it is within a few pixels of its proper destination. For such an implementation, the behavior can be given some extra global variables, akin to the values assigned to the state objects in Chapter 56, including the pixel coordinates of the ideal destination for a dragged element. An `onload` event handler for the page can fire a public `init()` function in each element's behavior to assign those coordinate values. Any event that can bubble (such as mouse events) does so from the behavior to the target. Therefore, you can extend the event action of the behavior by adding a handler for the same event to the element outside of the behavior.

Example 2: Text rollover behavior

In the second example, you see how a behavior exposes a global variable and function as a public property and method, respectively. The demonstration reinforces the notion that even if a single behavior file is associated with multiple elements (for example, the elements share the same class, and the behavior is assigned to the class), each behavior maintains its own variable values, independent of the other elements and their behaviors.

The nature of this behavior is to set the `color` style property of the associated element to either a default color (red) or to another color that has been passed into the behavior via one of its public methods. The color setting is preserved in one of the behavior's global variables, and that variable is exposed as a public property.

Listing 48-3 shows the `.htc` behavior file's content. Only two events are bound to this behavior: `onmouseover` and `onmouseout` — the typical rollover events. The `onmouseover` event invokes the `makeHot()` function, while the `onmouseout` event invokes the `makeNormal()` function. Before the `makeHot()` function makes any changes to the `color` and `fontWeight` style properties of the element, existing settings are preserved in (non-public) global variables in the behavior. This allows the `makeNormal()` function to restore the original settings, regardless of what document styles may be applied to the element in a variety of pages. That's something to keep in mind when you design behaviors: they can be deployed in pages controlled by any number of stylesheets. Don't assume any basic style setting; instead, use the `currentStyle` property to read and preserve the effective property values before touching them with your behavior's modification scripts.

Neither of the event handler functions are exposed as public methods. This was a conscious decision for a couple of reasons. The most important reason is that both functions rely on being triggered by a known event occurring on the element. If either function were invoked externally, the event object would contain none of the desired information. Another reason behind this is from a common programming style for components that protects inner workings, while exposing only those methods and properties that are "safe" for others to invoke. For this code, the public method does little more than set a property. It's an important property, to be sure, and one of the protected functions relies on it. But by allowing the public method little room to do any damage other than execution of the behavior, the design makes the behavior component that more robust.

Assigning a color value to the public property and passing one as a parameter to the public method accomplishes the same result in this code. As you will see, the property gets used in the demonstration page to retrieve the current value of the global variable. In a production behavior component, the programmer would probably choose to expose this value strictly as a read/write property or expose two methods, one for getting and one for setting the value. The choice would be at the whim of the programmer's style and would likely not be both. Using a method, however, especially for setting a value, creates a framework in which the programmer can also perform validation of the incoming value before assigning it to the global variable (something the example here does not do).

Listing 48-3: **Rollover Behavior (makeHot.htc)**

```
 <public:attach event="onmouseover" onevent="makeHot()" />
<public:attach event="onmouseout" onevent="makeNormal()" />
<public:property name="hotColor" />
<public:method name="setHotColor" />
<script type="text/javascript">
var oldColor, oldWeight;
var hotColor = "red";

function setHotColor(color) {
   hotColor = color;
}

function makeHot() {
   if (event.srcElement == element) {
      oldColor = element.currentStyle.color;
      oldWeight = element.currentStyle.fontWeight;
      element.style.color = hotColor;
      element.style.fontWeight = "bold";
   }
}

function makeNormal() {
   if (event.srcElement == element) {
      element.style.color = oldColor;
      element.style.fontWeight = oldWeight;
   }
}
</script>
```

To put the public information and the behavior, itself, to work, a demonstration page includes three spans within a paragraph that are associated with the behavior. Listing 48-4 shows the code for the demo page.

In addition to the text with rollover spans, the page contains two `select` controls, which let you assign a separate color to each of the three elements associated with the behavior. The first `select` element lets you choose one of the three elements. Making that choice invokes the `readColor()` function in the same page. This is the function that reads the `hotColor` public property of the chosen span. That color value is used to select the color name for display in the second `select` element. If you make a choice in the list of colors, the `applyVals()` function invokes the public `setHotColor()` method of the element currently selected from the list of elements. Rolling the mouse over that element now highlights in the newly selected color, while the other elements maintain their current settings.

Listing 48-4: **Applying the Rollover Behavior**

```
<html>
   <head>
      <title>IE5+ Behavior Demo (Styles)</title>
      <style type="text/css">
      .hotStuff {font-weight:bold; behavior:url(makeHot.htc)}
      </style>
      <script type="text/javascript">
      function readColor(choice) {
         var currColor = document.all(choice.value).hotColor
         var colorList = choice.form.color
         for (var i = 0; i < colorList.options.length; i++) {
            if (colorList.options[i].value == currColor) {
               colorList.selectedIndex = i
               break
            }
         }
      }
      function applyVals(form) {
         var elem = form.elem.value
         document.all(elem).setHotColor(form.color.value)
      }
      </script>
   </head>
   <body>
      <h1>IE5+ Behavior Demo (Styles)</h1>
      <hr />
      <form>
         Choose Hilited Element: <select name="elem"
         onchange="readColor(this)">
            <option value="elem1">First</option>
            <option value="elem2">Second</option>
            <option value="elem3">Third</option>
         </select> Choose Hilite Color: <select name="color"
         onchange="applyVals(this.form)">
            <option value="red" selected="selected">Red</option>
            <option value="blue">Blue</option>
            <option value="green">Green</option>
```

Continued

Listing 48-4 *(continued)*

```
        </select>
      </form>
      <p>Lorem ipsum dolor sit amet, <span id="elem1"
         class="hotStuff">consectetaur</span> adipisicing elit, sed do eiusmod
         tempor incididunt ut <span id="elem2" class="hotStuff">labore et
         dolore magna aliqua</span>. Ut enim adminim veniam, quis nostrud
         exercitation ullamco laboris <span id="elem3" class="hotStuff">nisi ut
         aliquip ex ea commodo consequat</span>.</p>
   </body>
</html>
```

Behaviors are not the solution for every scripting requirement. As demonstrated here, they work very well for generic style manipulation, but you are certainly not limited to that sphere. By having a reference back to the element associated with the behavior, and then to the document that contains the element, a behavior's scripts can have free run over the page — provided the actions are either generic among any page or generic among a design template that is used to build an entire Web site or application.

Even if you don't elect to use behaviors now (perhaps because you must support browsers other than WinIE5+), they may be in your future. Behaviors are fun to think about and also instill good programming practice in the art of creating reusable, generalizable code.

For More Information

In addition to the address of W3C activity on behaviors, Microsoft devotes many pages of its developer site to behaviors. Here are some useful pointers.

Overview:

`http://msdn.microsoft.com/workshop/author/behaviors/overview.asp`

Using DHTML Behaviors:

`http://msdn.microsoft.com/workshop/author/behaviors/howto/using.asp`

Default Behaviors Reference:

`http://msdn.microsoft.com/workshop/author/behaviors/reference/reference.asp`

IE5.5 Element Behaviors (an extension to the original behaviors):

`http://msdn.microsoft.com/workshop/author/behaviors/overview/elementb_ovw.asp`

Each of these locations ends with yet more links to related pages at the Microsoft Developer Network (MSDN) Web site.

✦ ✦ ✦

CHAPTER 49

Application: Tables and Calendars

In This Chapter

Accommodating older browsers

Scripted tables

Date calculations

Working with HTML tables is a lot of fun, especially if, like me, you are not a born graphics designer. By adding a few tags to your page, you can make your columnar data look more organized, professional, and appealing. Having this power under scripting control is even more exciting, because it means that in response to a user action or other variable information (such as the current date or time), a script can do things to the table as the table is being built. In IE4+ and W3C DOMs, scripts can modify the content and structure of a table even after the page has loaded, allowing the page to almost "dance."

You have three options when designing scripted tables for your pages, although only two are backward compatible with non-DHTML browsers:

- Static tables
- Dynamic tables
- Dynamic HTML tables

The design path you choose is determined by whether you need to dynamically update some or all fields of a table (data inside `<td>...</td>` tags) and which browser levels you need to support. To highlight the differences among the three styles, this chapter traces the implementation of a monthly calendar display in all three formats.

About the Calendars

Because the emphasis here is on the way tables are scripted and displayed, I quickly pass over structural issues of the calendar versions described in the following sections. The first two examples are backward compatible to all but the first-generation scriptable browsers. The final example, however, is a much more modern affair, utilizing table-related DOM objects and methods to simplify the code. It requires WinIE4+ (unfortunately, a bug in MacIE causes problems with the amount of `table` object modification the script does) or a W3C-compatible browser.

All three calendars follow similar (if not over-simplified) rules for displaying calendar data. English names of the months are coded into the script, so that they can be plugged into the calendar heading as needed. To make some of the other calendar calculations work (such as figuring out which day of the week is the first day of a given month in a given year), I define a method for my month objects. The method returns the JavaScript date object value for the day of the week of a month's first date. Virtually everything I do to implement the month objects is adapted from the custom objects discussion of Chapter 27.

Static Tables

The issue of updating the contents of a table's fields is tied to the nature of an HTML document being loaded and fixed in the browser's memory. Recall that for early browsers, you can modify precious few elements of a document and its objects after the document has loaded. That case certainly applies for typical data points inside a table's `<td>` tag pair. After a document loads — even if JavaScript has written part of the page — none of its content (except for text and textarea field contents and a few limited form element properties) can be modified without a complete reload.

Listing 49-1 contains the static version of a monthly calendar. The scripted table assembly begins in the Body portion of the document. Figure 49-1 shows the results.

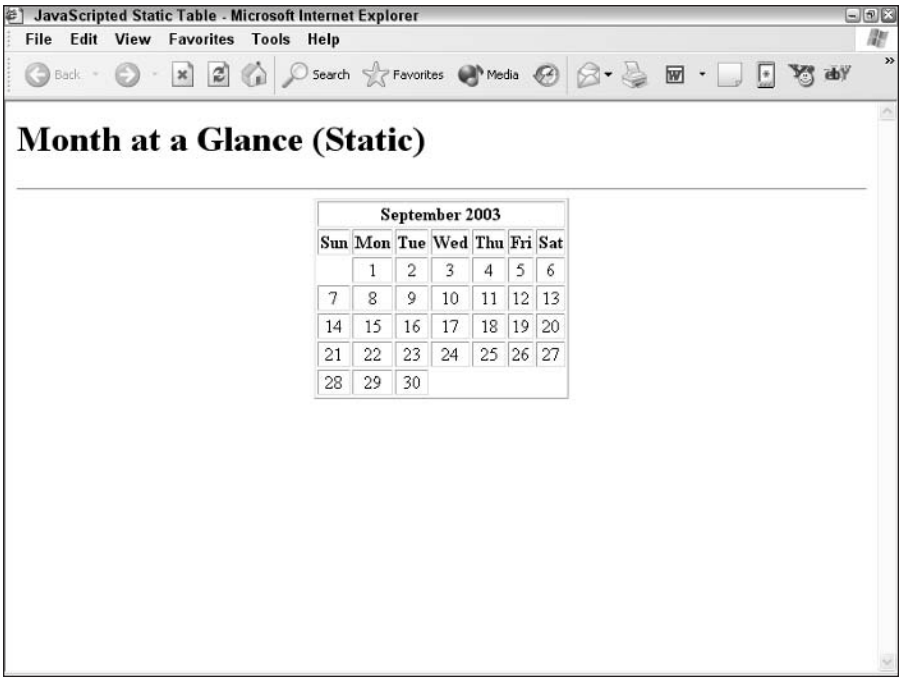


Figure 49-1: The static table calendar generated by Listing 49-1.

Listing 49-1: **A Static Table Generated by JavaScript**

```
<html>
   <head>
      <title>JavaScripted Static Table</title>
      <script type="text/javascript">
      // function becomes a method for each month object
      function getFirstDay(theYear, theMonth){
         var firstDate = new Date(theYear,theMonth,1);
         return firstDate.getDay() + 1;
      }
      // number of days in the month
      function getMonthLen(theYear, theMonth) {
         var oneHour = 1000 * 60 * 60;
         var oneDay = oneHour * 24;
         var thisMonth = new Date(theYear, theMonth, 1);
         var nextMonth = new Date(theYear, theMonth + 1, 1);
         var len = Math.ceil((nextMonth.getTime() -
            thisMonth.getTime() - oneHour)/oneDay);
         return len;
      }
      // correct for Y2K anomalies
      function getY2KYear(today) {
         var yr = today.getYear();
         return ((yr < 1900) ? yr + 1900 : yr);
      }
      // create array of month names
      theMonths = new Array("January", "February", "March", "April", "May",
         "June", "July", "August", "September", "October", "November",
         "December");
      </script>
   </head>
   <body>
      <h1>Month at a Glance (Static)</h1>
      <hr />
      <script type="text/javascript">
      // initialize some variables for later
      var today = new Date();
      var theYear = getY2KYear(today);
      var theMonth = today.getMonth(); // for index into our array

      // which is the first day of this month?
      var firstDay = getFirstDay(theYear, theMonth);
      // total number of <TD>...</TD> tags needed in for loop below
      var howMany = getMonthLen(theYear, theMonth) + firstDay;

      // start assembling HTML for table
      var content = "<center><table border='1'>";
      // month and year display at top of calendar
      content += "<tr><th colspan='7'>" + theMonths[theMonth] + " " + theYear +
         "<\/th><\/tr>";
```

Continued

Listing 49-1 *(continued)*

```
      // days of the week at head of each column
      content += "<tr><th>Sun<\/th><th>Mon<\/th><th>Tue<\/th><th>Wed<\/th>";
      content += "<th>Thu<\/th><th>Fri<\/th><th>Sat<\/th><\/tr>";
      content += "<tr>";

      // populate calendar
      for (var i = 1; i < howMany; i++) {
         if (i < firstDay) {
            // 'empty' boxes prior to first day
            content += "<td><\/td>";
         } else {
            // enter date number
            content += "<td align='center'>" + (i - firstDay + 1) + "<\/td>";
         }
         // start new row after each week
         if (i % 7 == 0 &&  i != howMany) {
            content += "<\/tr><tr>";
         }
      }
      content += "<\/table><\/center>";

      // blast entire table's HTML to the document
      document.write(content);
      </script>
   </body>
</html>
```

In this page, a little bit of the HTML — the `<h1>` heading and `<hr>` divider — is unscripted. The rest of the page consists entirely of the table definition, all of which is constructed in JavaScript. Most of the work for assembling the calendar's data points occurs inside of the `for` loop. Because not every month starts on a Sunday, the script determines the day of the week on which the current month starts. For all fields prior to that day, the `for` loop writes empty `<td></td>` tags as placeholders. After the numbered days of the month begin, the `for` loop writes the date number inside the `<td>...</td>` tags. Whatever the script puts inside the tag pair is written to the page as flat HTML. Under script control like that in the example, however, the script can designate what goes into each data point — rather than writing fixed HTML for each month's calendar.

The important point to note in this example is that although the content of the page may change automatically over time (without having to redo any HTML for the next month), after the page is written, its contents cannot be changed. If you want to add controls or links that are to display another month or year, you have to rewrite the entire page. This can be accomplished by passing the desired month and year as a search string for the current page's URL and then assigning the combination to the `location.href` property. You also have to add script statements to the page that look for a URL search string, extract the passed values, and use those values to generate the calendar while the page loads (see Chapter 17 for examples of how to accomplish this feat). But to bring a calendar such as this even more to life (while avoiding page reloading between views), you can implement it as a dynamic table.

Dynamic Tables

The only way to make data points of a table dynamically updatable in backward-compatible browsers is to turn those data points into text (or `textarea`) objects. The approach to this implementation is different from the static table because it involves the combination of *immediate* and *deferred* scripting. Immediate scripting facilitates the building of the table framework, complete with fields for every modifiable location in the table. Deferred scripting enables users to make choices from other interface elements, causing a new set of variable data to appear in the table's fields.

Listing 49-2 turns the preceding static calendar into a dynamic one by including controls that enable the user to select a month and year to display in the table. As testament to the support for absolute backward compatibility, a button triggers the redrawing of the calendar contents, rather than `onchange` event handlers in the `select` elements; a bug in NN2 for Windows caused that event not to work for the `select` object. Don't worry, the last example you see a bit later in the chapter does away with backward compatibility in favor of a sleeker design.

Form controls aside, the look of this version is quite different from the static calendar. Compare the appearance of the dynamic version shown in Figure 49-2 against the static version in Figure 49-1.

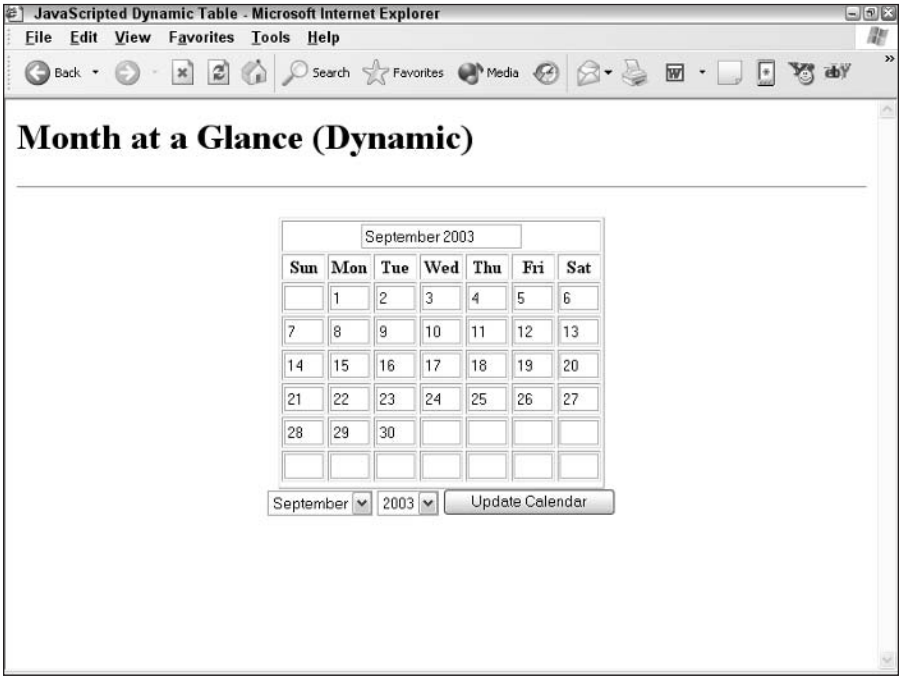


Figure 49-2: Dynamic calendar generated by Listing 49-2.

Listing 49-2: **A Dynamic Calendar Table**

```
<html>
   <head>
      <title>JavaScripted Dynamic Table</title>
      <script type="text/javascript">
      // function becomes a method for each month object
      function getFirstDay(theYear, theMonth){
         var firstDate = new Date(theYear,theMonth,1);
         return firstDate.getDay();
      }
      // number of days in the month
      function getMonthLen(theYear, theMonth) {
         var oneHour = 1000 * 60 * 60;
         var oneDay = oneHour * 24;
         var thisMonth = new Date(theYear, theMonth, 1);
         var nextMonth = new Date(theYear, theMonth + 1, 1);
         var len = Math.ceil((nextMonth.getTime() -
            thisMonth.getTime() - oneHour)/oneDay);
         return len;
      }
      // correct for Y2K anomalies
      function getY2KYear(today) {
         var yr = today.getYear();
         return ((yr < 100) ? yr + 1900 : yr);
      }
      // create array of month names
      theMonths = new Array("January", "February", "March", "April", "May",
         "June", "July", "August", "September", "October", "November",
         "December");      // deferred function to fill fields of table
      function populateFields(form) {
         // initialize variables for later from user selections
         var theMonth = form.chooseMonth.selectedIndex;
         var theYear =
            form.chooseYear.options[form.chooseYear.selectedIndex].value;
         // initialize date-dependent variables

         // which is the first day of this month?
         var firstDay = getFirstDay(theYear, theMonth);
         // total number of <TD>...<\/TD> tags needed in for loop below
         var howMany = getMonthLen(theYear, theMonth);

         // set month and year in top field
         form.oneMonth.value = theMonths[theMonth] + " " + theYear;
         // fill fields of table
         for (var i = 0; i < 42; i++) {
            if (i < firstDay || i >= (howMany + firstDay)) {
               // before and after actual dates, empty fields
               // address fields by name and [index] number
               form.oneDay[i].value = "";
            } else {
               // enter date values
               form.oneDay[i].value = i - firstDay + 1;
            }
        }
      }
      </script>
   </head>
```

```
   <body>
      <h1>Month at a Glance (Dynamic)</h1>
      <hr />
      <script type="text/javascript">
      // initialize variable with HTML for each day's field
      // all will have same name, so we can access via index value
      // empty event handler prevents
      // reverse-loading bug in some platforms
      var oneField = "<input type='text' name='oneDay' size='2' onfocus=''>";
      // start assembling HTML for raw table
      var content = "<form><center><table border='1'>";
      // field for month and year display at top of calendar
      content += "<tr><th colspan='7'><input type='text'
         name='oneMonth'><\/th><\/tr>";
      // days of the week at head of each column
      content += "<tr><th>Sun<\/th><th>Mon<\/th><th>Tue<\/th><th>Wed<\/th>";
      content += "<th>Thu<\/th><th>Fri<\/th><th>Sat<\/th><\/tr>";
      content += "<tr>";

      // layout 6 rows of fields for worst-case month
      for (var i = 1; i < 43; i++) {
         content += "<td align='middle'>" + oneField + "<\/td>";
         if (i % 7 == 0) {
            content += "<\/tr><tr>";
         }
      }

      content += "<\/table>";
      // blast empty table to the document
      document.write(content);
      </script>
      <select name="chooseMonth">
         <option value="January" selected="selected">January</option>
         <option value="February">February</option>
         <option value="March">March</option>
         <option value="April">April</option>
         <option value="May">May</option>
         <option value="June">June</option>
         <option value="July">July</option>
         <option value="August">August</option>
         <option value="September">September</option>
         <option value="October">October</option>
         <option value="November">November</option>
         <option value="December">December</option>
      </select> <select name="chooseYear">
         <option value="2003" selected="selected">2003</option>
         <option value="2004">2004</option>
         <option value="2005">2005</option>
         <option value="2006">2006</option>
         <option value="2007">2007</option>
         <option value="2008">2008</option>
         <option value="2009">2009</option>
         <option value="2010">2010</option>
      </select>
         <br /><input type="button" name="updater" value="Update Calendar"
         onclick="populateFields(this.form)" />
      </form>  <!--start tag written by script -->
   </body>
</html>
```

When you first load Listing 49-2, it creates an empty table along with numerous text objects. An `onload` event handler in the Body definition also could easily set the necessary items to load the current month.

From a cosmetic point of view, the dynamic calendar may not be as pleasing as the static one in Figure 49-1. Several factors contribute to this appearance.

From a structural point of view, creating a table that can accommodate any possible layout of days and dates that a calendar may require is essential. That means a basic calendar consisting of six rows of fields. For many months, the last row remains completely empty. But because the table definition must be fixed when the page loads, this layout cannot change on the fly.

The more obvious cosmetic comparison comes from the font and alignment of data in text objects. Except for capabilities of browsers capable of using stylesheets, you're stuck with what the browser presents in both categories. In the static version, you can define different font sizes and colors for various fields, if you want (such as coloring the entry for today's date). Not so in text objects in a backward-compatible program.

This cosmetic disadvantage, however, is a boon to functionality and interactivity on the page. Instead of the user being stuck with an unchanging calendar month, this version includes pop-up menus from which the user can select a month and year of choice. Clicking the Update Calendar button refills the calendar fields with data from the selected month.

One more disadvantage to this dynamic table surfaces, however: All text objects can be edited by the user. For many applications, this capability may not be a big deal. But if you're creating a table-based application that encourages users to enter values in some fields, be prepared (in other words, have event handlers in place) to either handle calculations based on changes to any field or to alert users that the fields cannot be changed (and restore the correct value).

Hybrids

It will probably be the rare scripted table that is entirely dynamic. In fact, the one in Figure 49-2 is a hybrid of static and dynamic table definitions. The days of the week at the top of each column are hard-wired into the table as static elements. If your table design can accommodate both styles, implement your tables that way. The fewer the number of text objects defined for a page, the better the performance for rendering the page, and the less confusion for the page's users.

Dynamic HTML Tables

If you have the luxury of developing for IE4+ or NN6+/W3C, you have all the resources of the `table` and related element objects, as described in Chapter 37. The resulting application will appear to be much more polished, because not only does your content flow inside a table (which you can style to your heart's delight), but the content is dynamic within the table.

Listing 49-3 blends the calendar calculations from the earlier two calendar versions with the powers of IE4+ and W3C DOMs. A change to a requested calendar month or year instantly redraws the body of the table, without disturbing the rest of the page (see Figure 49-3).

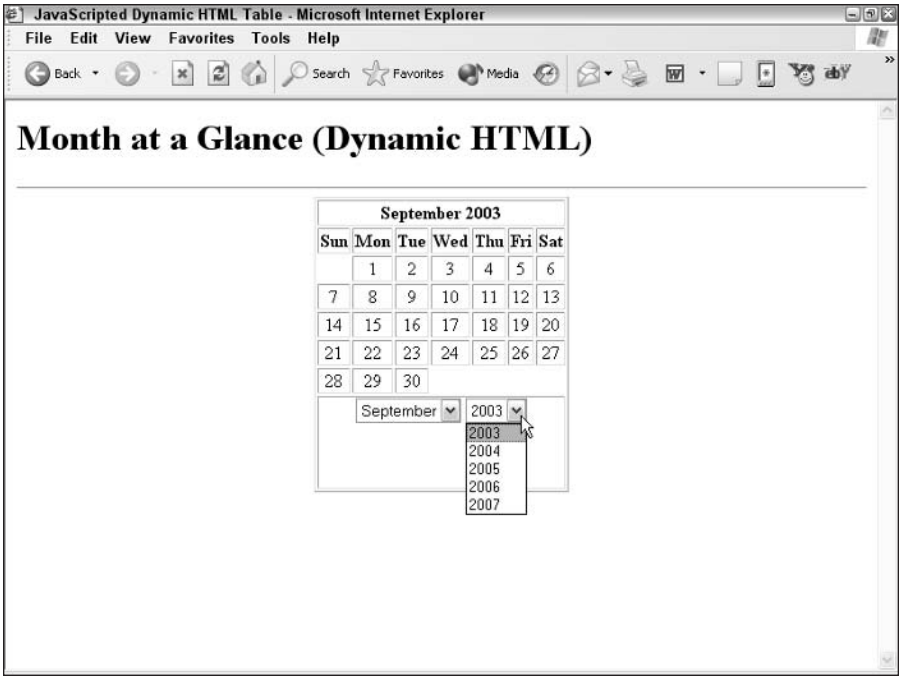


Figure 49-3: DHTML table.

Basic date calculations are identical to the other two versions. Because this page has to be used with more modern browsers, it can use the shortcut array creation syntax for the month names array. Also, the way the table must be constructed each time is very different from two previous versions. In this version, the script creates new table rows, creates new cells for those rows, and then populates those cells with the date numbers. Repeat loop logic is quite different, relying on a combination of `while` and `for` loops to get the job done.

Other features made possible by more modern browsers include automatic population of the list of available years. This page will never go out of style (unless browsers in 2050 no longer use JavaScript). There is also more automation in the triggers of the function that populates the table.

Listing 49-3: **Dynamic HTML Calendar**

```
<html>
   <head>
      <title>JavaScripted Dynamic HTML Table</title>
      <style type="text/css">
      td, th {text-align:center}
      </style>
      <script type="text/javascript">
      /*******************
```

Continued

Listing 49-3 *(continued)*

```
UTILITY FUNCTIONS
********************/
// day of week of month's first day
function getFirstDay(theYear, theMonth){
   var firstDate = new Date(theYear,theMonth,1);
   return firstDate.getDay();
}
// number of days in the month
function getMonthLen(theYear, theMonth) {
   var oneHour = 1000 * 60 * 60;
   var oneDay = oneHour * 24;
   var thisMonth = new Date(theYear, theMonth, 1);
   var nextMonth = new Date(theYear, theMonth + 1, 1);
   var len = Math.ceil((nextMonth.getTime() -
      thisMonth.getTime() - oneHour)/oneDay);
   return len;
}
// create array of English month names
var theMonths = ["January","February","March","April","May","June",
   "July","August","September","October","November","December"];
// return IE4+ or W3C DOM reference for an ID
function getObject(obj) {
   var theObj;
   if (document.all) {
      if (typeof obj == "string") {
         return document.all(obj);
      } else {
         return obj.style;
      }
   }
   if (document.getElementById) {
      if (typeof obj == "string") {
         return document.getElementById(obj);
      } else {
         return obj.style;
      }
   }
   return null;
}

/************************
DRAW CALENDAR CONTENTS
************************/
// clear and re-populate table based on form's selections
function populateTable(form) {
   var theMonth = form.chooseMonth.selectedIndex;
   var theYear = parseInt(
      form.chooseYear.options[form.chooseYear.selectedIndex].text);
   // initialize date-dependent variables
   var firstDay = getFirstDay(theYear, theMonth);
   var howMany = getMonthLen(theYear, theMonth);
```

```
      // fill in month/year in table header
      getObject("tableHeader").innerHTML = theMonths[theMonth] +
         " " + theYear;

      // initialize vars for table creation
      var dayCounter = 1;
      var TBody = getObject("tableBody");
      // clear any existing rows
      while (TBody.rows.length > 0) {
         TBody.deleteRow(0);
      }
      var newR, newC;
      var done=false;
      while (!done) {
         // create new row at end
         newR = TBody.insertRow(TBody.rows.length);
         for (var i = 0; i < 7; i++) {
            // create new cell at end of row
            newC = newR.insertCell(newR.cells.length);
            if (TBody.rows.length == 1 && i < firstDay) {
               // no content for boxes before first day
               newC.innerHTML = "";
               continue;
            }
            if (dayCounter == howMany) {
               // no more rows after this one
               done = true;
            }
            // plug in date (or empty for boxes after last day)
            newC.innerHTML = (dayCounter <= howMany) ?
               dayCounter++ : "";
         }
      }
   }

   /*******************
   INITIALIZATIONS
   *******************/
   // create dynamic list of year choices
   function fillYears() {
      var today = new Date();
      var thisYear = today.getFullYear();
      var yearChooser = document.dateChooser.chooseYear;
      for (i = thisYear; i < thisYear + 5; i++) {
         yearChooser.options[yearChooser.options.length] = new Option(i, i);
      }
      setCurrMonth(today);
   }
   // set month choice to current month
   function setCurrMonth(today) {
      document.dateChooser.chooseMonth.selectedIndex = today.getMonth();
   }
   </script>
</head>
```

Continued

Listing 49-3 *(continued)*

```
   <body onload="fillYears(); populateTable(document.dateChooser)">
      <h1>Month at a Glance (Dynamic HTML)</h1>
      <hr />
      <table id="calendarTable" border="1" align="center">
         <tr>
            <th id="tableHeader" colspan="7"></th>
         </tr>
         <tr>
            <th>Sun</th>
            <th>Mon</th>
            <th>Tue</th>
            <th>Wed</th>
            <th>Thu</th>
            <th>Fri</th>
            <th>Sat</th>
         </tr>
         <tbody id="tableBody"></tbody>
         <tr>
            <td colspan="7">
               <form name="dateChooser">
                  <select name="chooseMonth"
                  onchange="populateTable(this.form)">
                     <option selected="selected">January</option>
                     <option>February</option>
                     <option>March</option>
                     <option>April</option>
                     <option>May</option>
                     <option>June</option>
                     <option>July</option>
                     <option>August</option>
                     <option>September</option>
                     <option>October</option>
                     <option>November</option>
                     <option>December</option>
                  </select> <select name="chooseYear"
                  onchange="populateTable(this.form)">
                  </select>
               </form>
               <br />
               <br />
            </td>
         </tr>
      </table>
   </body>
</html>
```

Further Thoughts

The best deployment of an interactive calendar requires the kind of Dynamic HTML currently available in IE4+ and NN6+/W3C DOMs. Moreover, the cells in those DOMs can receive mouse events so that a user can click a cell and it will highlight perhaps in a different color or display some related, but otherwise hidden, information.

A logical application for such a dynamic calendar would be in a pop-up window or frame that lets a user select a date for entry into a form date field. It eliminates typing in a specific date format, thereby ensuring a valid date entry every time. Without DHTML, you can create a static version of the calendar that renders the numbers in the calendar cells as HTML links. Those links can use a `javascript:` URL to invoke a function call that sets a date field in the main form.

✦ ✦ ✦

CHAPTER 50

Application: A Lookup Table

In This Chapter

Serverless data collection lookup

Data-entry validation

One of the first ideas that intrigued me about JavaScript was the notion of delivering CGI-like functionality along with an HTML document. On the Web, numerous, small data collections currently require CGI scripting and a back-end database engine to drive them. Of course, not everyone who has information to share has access to the server environment (or the expertise) to implement such a solution. JavaScript provides that power.

A Serverless Database

Before you get too carried away with the idea of letting JavaScript take the place of your SQL database, you need to recognize several limitations that prevent JavaScript from being a universal solution. First, any database that you embed into an HTML document is read-only. Although you can script an interface and lookup routines for the user, no provisions are available for writing revised information back to the server, if that is your intention.

A second consideration is the size of the data collection. Unlike databases residing on servers, the entire JavaScript database (or subset you define for inclusion into a single HTML document) must be downloaded to the user's browser before the user can work with the data. As a point of reference, think about image files. At 56.6 Kbps through a dial-up connection, how large an image file would you tolerate downloading? Whatever that limit may be (anywhere from 10 to 35K, depending on your patience) is what your database size limit should be. For many special-purpose collections, this is plenty of space, assuming one byte per character. Unlike what happens when the user downloads an embedded image file, the user doesn't see special status bar messages about your database: To the browser, these messages are all part of the HTML coming in with the document.

The kind of data I'm talking about here is obviously text data. That's not to say you can't let your JavaScript-enhanced document act as a front end to data files of other types on your server. The data in your embedded lookup table can be URLs to images that get swapped into the page as needed.

The Database

As I was thinking about writing a demonstration of a serverless database, I encountered a small article in the *Wall Street Journal* that related information I had always suspected. The Social Security numbers assigned to virtually every U.S. citizen are partially coded to indicate the state in which you registered for your Social Security number. This information often reveals the state in which you were born (another study indicates that two-thirds of U.S. citizens live their entire lives in the same state). The first three digits of the nine-digit number comprise this code.

When the numbering system was first established, each state was assigned a block of three-digit numbers. Therefore, if the first three digits fall within a certain range, the Social Security Administration has you listed as being registered in the corresponding state or territory. I thought this would be an interesting demonstration for a couple of reasons: first, the database is not that large, so it can be easily embedded into an HTML document without making the document too big to download, even on slow Internet connections; second, it offers some challenges to data-entry validation, as you see in a moment.

Before young people from populous states write to tell me that their numbers are not part of the database, let me emphasize that I am well aware that several states have been assigned number blocks not reflected in the database. This example is only a demonstration of scripting techniques, not an official Social Security Administration page.

The Implementation Plan

For this demonstration, all I started with was a printed table of data. I figured that the user interface for this application would probably be very plain: a text field in which the user can enter a three-digit number, a clickable button to initiate the search, and a text field to show the results of the lookup. Figure 50-1 shows the page. Pretty simple by any standards.

Given that user interface (I almost always start a design from the *interface* — how my page's users will experience the information presented on the page), I next planned the internals. I needed the equivalent of two tables: one for the numeric ranges, and one for the state names. Because most of the numeric ranges are contiguous, I could get by with a table of the high number of each range. This meant that the script would have to trap elsewhere for the occasional numbers that fall outside of the table's ranges — the job of data validation.

Because the two tables were so closely related to each other, I had the option of creating two separate arrays, so that any given index value would correspond to both the numeric and state name entries in both tables (*parallel arrays*, I call them). The other option was to create a two-dimensional array (see Chapter 30), in which each array entry has data points for both the number and state name. For purposes of demonstration to first-time database builders, I decided to stay with two parallel arrays. This method makes visualizing how the lookup process works with two separate arrays a little easier.

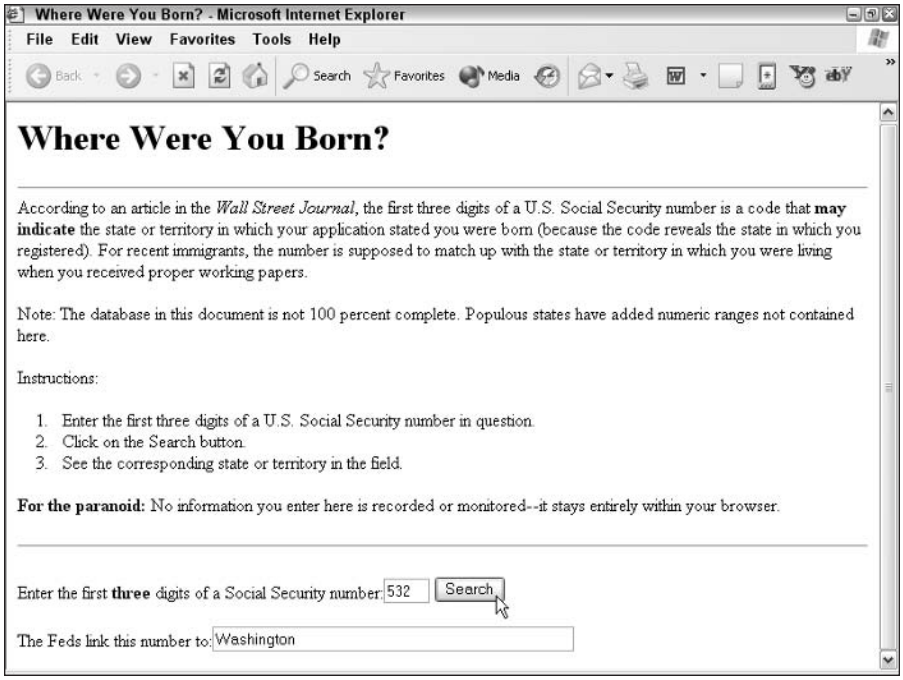


Figure 50-1: The Social Security number lookup page.

The Code

The HTML document starts normally through the definition of the document title:

```
<html>
   <head>
      <title>Where Were You Born?</title>
```

In this application, you will see that I place utility function definitions close to the top of the script sections and put any action-oriented scripts (functions acting in response to event handlers) closer to the bottom of the script sections. My preference is to have all dependencies resolved before the script needs them. This philosophy carries over from the logic that dictates putting as many scripts in the Head as possible, so that even if the user (or network) should interrupt downloading of a page before every line of HTML reaches the browser, any user interface element relying on scripts will have those scripts loaded and ready to go. The order of functions in this example is not critical, because as long as they all reside in the Head section, they are defined and loaded by the time the field and button appear at the bottom of the page. But after I develop a style, I find it easier to stick with it — one less matter to worry about while scripting a complex application.

After creating an array (named `ssn`) with 57 empty slots, the script populates all 57 data points of the array, starting with the first entry going into the slot numbered 0. These data numbers correspond to the top end of each range in the 57-entry table. For example, any number greater than 3 but less than or equal to 7 falls into the range of the second data entry of the array (`ssn[1]`).

```
<script type="text/javascript">
// create array listing all the top end of each numeric range
var ssn = new Array(57);
ssn[0] = 3;
ssn[1] = 7;
ssn[2] = 9;
ssn[3] = 34;
ssn[4] = 39;
ssn[5] = 49;
ssn[6] = 134;
ssn[7] = 158;
ssn[8] = 211;
ssn[9] = 220;
ssn[10] = 222;
ssn[11] = 231;
ssn[12] = 236;
ssn[13] = 246;
ssn[14] = 251;
ssn[15] = 260;
ssn[16] = 267;
ssn[17] = 302;
ssn[18] = 317;
ssn[19] = 361;
ssn[20] = 386;
ssn[21] = 399;
ssn[22] = 407;
ssn[23] = 415;
ssn[24] = 424;
ssn[25] = 428;
ssn[26] = 432;
ssn[27] = 439;
ssn[28] = 448;
ssn[29] = 467;
ssn[30] = 477;
ssn[31] = 485;
ssn[32] = 500;
ssn[33] = 502;
ssn[34] = 504;
ssn[35] = 508;
ssn[36] = 515;
ssn[37] = 517;
ssn[38] = 519;
ssn[39] = 520;
ssn[40] = 524;
ssn[41] = 525;
ssn[42] = 527;
ssn[43] = 529;
ssn[44] = 530;
ssn[45] = 539;
ssn[46] = 544;
ssn[47] = 573;
ssn[48] = 574;
```

```
db[db.length] = new dbRecord(false, "Low-Sodium","foods.htm#losodium",2,
   "A healthy treat...");
db[db.length] = new dbRecord(true,
   "<font color=red size=+2>Pickles</font>","",0,"");
db[db.length] = new dbRecord(true, "Cucumber","foods.htm#cucumber",1,
   "What\\'s new in cukes...");
db[db.length] = new dbRecord(false, "Dill", "foods.htm#dill",2,
   "Pucker up...");
db[db.length] = new dbRecord(false, "Fresh", "foods.htm#fresh",2,
   "You\\'d prefer stale?");
db[db.length] = new dbRecord(false,"Sour", "foods.htm#sour",2,
   "For sweeties...");
// add more records to complete your outline
// ** END AUTHOR-ADJUSTABLE SPECIFICATIONS **//
```

Each record consists of five items. The first item is a Boolean value, which denotes whether the item is a mother item (that is, the next item in the list is nested one level deeper). The HTML that displays in the outline comes next. This text can be multiple-word strings, or any HTML that you like (some users have assigned `<img>` tags to show pictures instead of text). For the third item, you can insert any valid URL, whether it be to a separate site, an anchor in another document (as shown here), or even a `javascript:` URL to execute another function. If you don't want an entry in the outline to be a link — just plain, flat text — leave this third item as an empty string, as I do here for the topmost items in both categories. The fourth item is a number representing how many nested levels the item has. And finally, the last item is a string containing the text that appears in the status bar when the user rolls the mouse over the item in the outline. Because of a quirk in the way the status bar responds to quoted characters, any string literal character (normally preceded with a backslash) requires two backslashes (one to alert the browser of the other).

Be sure to keep the items for the `db` array in the same top-to-bottom order as you'd expect to see in a fully expanded outline. Notice that the index values of the array are automatically inserted for you: The `length` property of an array is always one more than the highest index. By inserting references to the `db.length` property in the brackets, you instruct JavaScript to "walk the ladder" upward from zero. If you move things around the outline, however, don't forget to adjust the indentation levels if they are affected by the content changes.

The bottom of the array creation section marks the end of the code that you need to modify after you deploy the outliner. The rest of the JavaScript code works silently for you, but if you intend to perform further customizations to the outliner, understanding how it all works will help.

On to the constructor function for the `dbRecord` entries. This function is the classic JavaScript way to build a custom object (see Chapter 33):

```
// object constructor for each outline entry
function dbRecord(mother,display,URL,indent,statusMsg){
   this.mother = mother;   // is this item a parent?
   this.display = display; // text to display
   this.URL = URL;         // link tied to text; if empty string, item
                           // appears as straight text
   this.indent = indent;   // how many levels nested?
   this.statusMsg = statusMsg;  // descriptive text for status bar
   return this;
}
```

To preload all the images into the browser's image cache, you create new `Image` objects for each and assign the filenames to their `src` properties. Notice that these statements are not in functions, but rather execute as the page loads:

```
// pre-load all images into cache
var fillerImg = new Image(1,1);
fillerImg.src = "filler.gif";
var collapsedImg = new Image(widgetWidth,widgetHeight);
collapsedImg.src = collapsedWidget;
var expandedImg = new Image(widgetWidth,widgetHeight);
expandedImg.src = expandedWidget;
var endpointImg = new Image(widgetWidth,widgetHeight);
endpointImg.src = endpointWidget;
```

Cookie storage

To preserve the binary digit string between redraws of the outline, this script must save the string to a place that won't be overwritten or emptied during the document reload. The `document.cookie` fills that requirement nicely. Excerpting and adapting parts of Bill Dortch's cookie functions (see Chapter 18), this script simplifies the writing of a cookie that disappears when the user quits the browser:

```
// ** functions that get and set persistent cookie data **
// set cookie data
var mycookie = document.cookie;
function setCurrState(setting) {
   mycookie = document.cookie = "currState=" + escape(setting);
}
// retrieve cookie data
function getCurrState() {
   var label = "currState=";
   var labelLen = label.length;
   var cLen = mycookie.length;
   var i = 0;
   while (i < cLen) {
      var j = i + labelLen;
      if (mycookie.substring(i,j) == label) {
         var cEnd = mycookie.indexOf(";",j);
         if (cEnd ==     -1) {
            cEnd = mycookie.length;
         }
         return unescape(mycookie.substring(j,cEnd));
      }
      i++;
   }
   return "";
}
```

A global variable is used to act as a speedy intermediary between the actual browser cookie and the functions here that need to access cookie data. The `setCurrState()` function contains a construction that you don't see much in this book, but is quite valid. Notice the three-piece assignment statement. Evaluation of this statement works from right to left. The rightmost expression concatenates a cookie label and the value passed in as a parameter to the function. Note, too, that the value is passed through the `escape()` function to properly URL-encode the data for the sake of data integrity (so that spaces and odd punctuation don't mess up the mechanism). The concatenated value is assigned to the `document.cookie` property. With the value safely dropped into the cookie (it may be just one of several name/value pairs for this domain), the value of the `document.cookie` property (which includes all name/value pairs for the domain) is assigned to the `mycookie` global variable.

Retrieving information from the cookie still requires a bit of parsing to be on the safe side. If other cookie writing were to come from the current server path, more than one cookie would be available to the current document. Parsing the entire cookie for just the portion that corresponds to the `currState` labeled cookie ensures that the script gets only the data previously saved to that label.

The focal point

The `toggle()` function, which is pivotal in this outline scheme, receives as a parameter the index number of the `db` array element whose content the user just clicked. The purpose of this function is to grab a copy of the current outline state from the cookie, alter the binary representation of the clicked item, and feed the revised binary number back to the cookie (where it governs the display of the outline after the document reloads):

```
// **function that updates persistent storage of state**
// toggles an outline mother entry, storing new value in the cookie
function toggle(n) {
   var newString = "";
   var currState = getCurrState();
   var expanded = currState.charAt(n); // of clicked item
   newString += currState.substring(0,n);
   newString += expanded ^ 1; // Bitwise XOR clicked item
   newString += currState.substring(n+1,currState.length);
   setCurrState(newString); // write new state back to cookie
}
```

To make this happen, you must extract two pieces of information before any processing: the current state from the cookie and the current setting of the clicked item. The latter is saved in a local variable named `expanded` because its `0` or `1` value represents the expanded state of that particular entry in the outline.

With those information morsels in hand, the script starts building the new binary string that gets written back to the cookie. The new string consists of three pieces: the front part of the existing string up to (but not including) the digit representing the clicked item, the changed entry, and the rest of the original string.

Changing the setting of the clicked item from a `0` to a `1`, or vice versa, is necessary. Although I can implement this task a few different ways (for example, using a conditional expression or an `if...else` construction), I thought I'd exercise an operator that otherwise gets little use: the bitwise XOR operator (^). Because the values involved here are `0` and `1`, performing an XOR operation with the value of `1` inverts the original value:

```
0 ^ 1 = 1
1 ^ 1 = 0
```

Okay, perhaps using an XOR operator is showing off. But the experience forced me to understand a JavaScript power that may come in handy for the future.

Selecting a widget image for an entry

At this point, the script starts defining functions to help the script statements in the Body write the HTML for the new version of the outline. The `getGIF()` function determines which of the three widget image files needs to be specified for a particular entry in the outline. The function receives the index value to the `db` array of entries created earlier in the script. As the Body script assembles the HTML for the outline, it calls this function once for each item in the outline. In return, the function provides a reference to one of three `Image` objects created earlier:

```
// **functions used in assembling updated outline**
// returns the proper GIF file name for each entry's control
function getGIF(n, currState) {
   var mom = db[n].mother;  // is entry a parent?
   var expanded = currState.charAt(n); // of clicked item
   if (!mom) {
      return endpointWidget;
   } else {
      if (expanded == 1) {
         return expandedWidget;
      }
   }
   return collapsedWidget;
}
```

The decision process for this function first tries to eliminate any item that ends a mother–daughter chain. Any item that is as deeply nested as it can be (which means the item is not a mother) automatically gets the `endpointWidget` image.

Now you're left with trying to figure out whether the item in the display should get an expanded or collapsed icon. The holder of this information is the cookie. Thus, the script extracts the binary setting for the entry under scrutiny. If the cookie shows that entry to be a `1`, it means that the item has nested items showing and that it should get the `expandedWidget` image; otherwise, it should get the `collapsedWidget` image. Notice that you're returning *references* to the `Image` objects, not the names of the image files.

A similar excursion through each item determines what status message is assigned to the `onmouseover` event handler for each of the widget images. The decision tree is identical to that of the `getGIF()` function:

```
// returns the proper status line text based on the icon style
function getGIFStatus(n, currState) {
   var mom = db[n].mother;  // is entry a parent
   var expanded = currState.charAt(n); // of rolled item
   if (!mom) {
      return "No further items";
   } else {
      if (expanded == 1) {
         return "Click to collapse nested items";
      }
   }
   return "Click to expand nested items";
}
```

Initialize the cookie

The final task of the script running in the Head is to initialize the cookie if it's empty. Using the length of the `db` array as a counter, you build a string of `0`s, with one `0` for each item in the outline:

```
// initialize 'current state' storage field
if (getCurrState() == "" || getCurrState().length != (db.length)) {
   initState = "";
   for (i = 0; i < db.length; i++) {
      initState += "0";
   }
   setCurrState(initState);
}
</script>
</head>
```

Each of those 0s in the parameter to the setCurrState() function corresponds to a collapsed setting for an entry in the outline. In other words, the first time the outline appears, all items are in the collapsed mode. If you modify the outline for your own use by creating your own db array of data, the initial state of the cookie will be set for you automatically based on the length of the db array.

Writing the outline

At last we reach the document Body, where the outline is assembled and written to the page. Script statements here are immediate, meaning that they execute while the page loads. I have you begin by initializing some variables that you will need in a moment. The most important variable is newOutline, which will be used to accumulate the contents of the outline for eventual writing to the page:

```
<body>
   <script type="text/javascript" language="JavaScript1.1">
   // build new outline based on the values of the cookie
   // and data points in the outline data array.
   // This fires each time the user clicks on a control,
   // because the HREF for each one reloads the current document.
   var newOutline = "";
   var prevIndentDisplayed = 0;
   var showMyDaughter = 0;
   var currState = getCurrState(); // get whole state string
   document.write("<center><h3>Composition of Selected Foods</h3>
      <hr /></center>");
```

The following section is the real beef of this script: the part that assembles the HTML for the outline that displays as the document loads. In other words, this part must read the current state data from the cookie and assemble widget images and text links according to the map of expanded and collapsed items in the cookie data. These activities take place within a for loop that cycles through every item in the database. Each value of the i index refers to one listing in the db array. Trace the logic of one entry:

```
   // cycle through each entry in the outline array
   for (var i = 0; i < db.length; i++) {
      var theGIF = getGIF(i, currState);   // get the image
      var theGIFStatus = getGIFStatus(i, currState);  // get the status msg
      var currIndent = db[i].indent;   // get the indent level
      var expanded = currState.charAt(i);    // current state

      // display entry only if it meets one of three criteria
      if (currIndent == 0 || currIndent <= prevIndentDisplayed ||
         (showMyDaughter == 1 && (currIndent - prevIndentDisplayed == 1))) {
         newOutline += "<img src=\"filler.gif\" height='1' width='" +
            (indentPixels * currIndent) + "' />";
         newOutline += "<a href=\"javascript:history.go(0)\" " +
            "onmouseover=\"window.status='" + theGIFStatus +
            "';return true;\" onclick=\"toggle(" + i + ");return " +
            (theGIF != endpointWidget) + "\">";
         newOutline += "<img src=\"" + theGIF + "\" height='" +
            widgetHeight + "' width='" + widgetWidth +
            "' border='0' /></a>";
         if (db[i].URL == "" || db[i].URL == null) {
            newOutline += " " + db[i].display + "<br />";   // no link
         } else {
            newOutline += " <a href=\"" + db[i].URL + "\" target=\"" +
```

```
                displayTarget + "\" onmouseover=\"window.status=\'" +
                db[i].statusMsg + "\';return true;\">" + db[i].display +
                "</a><br />";
            }
            prevIndentDisplayed = currIndent;
            showMyDaughter = expanded;
            if (db.length > 25) {
               document.write(newOutline);
               newOutline = "";
            }
         }
      }
```

First, you call upon two previously defined functions to grab the widget image object and corresponding `onmouseover` message for the status bar. Two more variables contain the indent property for the item (that is, how many steps indented the item will appear in the outline structure) and the current expanded state, based on the cookie's entry for that item.

Not every entry in the outline database is displayed. For instance, a nested item whose mother is collapsed won't need to be displayed. To find out if an entry should be displayed, the script performs a number of tests on some of its values. An item can be displayed if any of the following conditions are met:

- The item is a topmost item, with an indentation factor of 0.
- The item is at the same or smaller indentation level as the previous item displayed.
- The previous item was tagged as being expanded, and the current item is indented from the previous item by one level.

Over the next few statements, the script pieces together the HTML for the outline entry, starting with the width necessary for the transparent filler image (based on the number of pixels specified for indentations near the top of the script). Next comes the link definition that wraps around the widget image. The following concepts apply to each link:

- The `href` attribute is the `javascript:` URL to invoke the `history.go()` method.
- The `onmouseover` event handler is set to adjust the status message to the previously retrieved message (notice the `return true` statement to make the setting take effect).
- The `onclick` event handler is set to call the `toggle()` function, passing the number of the item within the outline database. An `onclick` event handler is carried out before the browser responds to the click of the link by navigating to the URL. Therefore, the `toggle()` function changes the setting of the cookie a fraction of a second before the browser refreshes the document (which relies on that new cookie setting). But click events on widgets that have no children do not need to hit the `toggle()` function. Therefore, the content of the `return` statement is influenced by whether or not the widget image is an endpoint image.

In the next statement, the `newOutline` string accumulation continues with the `<img>` tag specifications for the widget art. Specifying the `height` and `width` attributes for the image is important, partly to help the browser lay out the page more quickly, partly to avoid pesky performance inconsistencies.

Next comes a decision about whether to display the item text as a link or as plain text. The script inspects the `db[i].URL` property to see if it is empty. If so, that means no URL is specified for a link, and the item should be built as plain text.

If a URL is specified for the item, the script instead constructs a link around the text. In this HTML assembly process, numerous calls to properties of the `db` array fetch properties of the entry for the URL, the status bar message, and the text to display. Notice, too, that the link sets the target of the link to the frame name assigned to `displayTarget` near the top of the script.

As you near the end of the loop, two variable values, `prevIndentDisplayed` and `showMyDaughter`, are updated with settings from the current traversal through the loop. These values influence the display of nested items for the next entry's journey through the loop.

But before looping around again, the script inspects whether the outline is longer than 25 entries. If so, the script writes the outline entries that have accumulated so far, resetting the `newOutline` variable to empty for the next time through the loop. The reasoning behind this last routine is to help long outlines start to display their goods faster. I have seen Web site authors use this outline for literally hundreds of entries. At that quantity, the usually fast JavaScript begins to bog down a bit. By writing lines from a big outline to the page early, the user gets visual feedback that something is happening.

Once outside the loop, the script writes whatever last items may have accumulated in the `newOutline` variable. For outlines with less than 25 items, the whole outline is written in one push; for longer outlines, the value is empty at this point, because the intermediate writings have completed the job.

All that's left is to close up standard tags to finish the document definition:

```
      document.write(newOutline);
      </script>
   </body>
</html>
```

Notice that the `document.write()` statement here is not followed by `document.close()`. Because this content is being written as the page loads, the output stream is closed at the end of the page's HTML.

Customization possibilities

Although this DHTML-free outliner is not the fanciest to be found on the Web, it is, nevertheless, quite popular probably due to its ease of customizability and backward compatibility to all but the earliest browsers (you can find the very original version at my Web site). Other page authors have pushed and pulled on this code to tailor it to a variety of special needs.

Alternative displays

At the root of almost all significant customization jobs lie modifications to the `dbRecord` object constructor near the beginning of the page and the HTML assembly portion in the Body. They work hand in hand. For example, one user wants different links in the outline to load pages into different targets. Most links are to load content into another frame of the same frameset, while others are to replace the frameset entirely. In the version provided previously, one target is assumed, and it is set as a global variable. But if you need to provide different targets for each item, you can add a new property (perhaps named `target`) to the `dbRecord` constructor, and assign the string name of the target (for example, `"Frame2"`, `"_top"`) to the property for each item. Then, in the HTML accumulation portion, assign the value of `db[i].target` to that `target` attribute (watching out for the necessary pairings of quote symbols, as shown in other attribute assignments).

Another request asked that the text associated with the plus/minus images be clickable, not to navigate to another page, but to expand and collapse the nested content. All the pieces for this variation are already in place. By performing minor reconstructive surgery on the HTML accumulator script, you can add a branch that looks for the `db[i].mother` property. If it's `true`, don't write the closing `</a>` tag after the widget. Instead, branch to write the `db[i].display` text without its own URL link, and write the widget's `</a>` tag after the text. Now the widget and text share the same link as the widget originally had.

Cookie-free zones

Not everyone likes to develop with cookies. That's not a problem for this outliner, even though the previous example uses them liberally. The data that preserves the state of the outline is nothing but a string of `1`s and `0`s. If you are using a frameset, that string can be preserved as a global variable in the framesetting document.

To minimize the changes needed to the existing code, you can continue to use the same functions — `setCurrState()` and `getCurrState()` — as the interfaces to the reading and writing of the state. Begin by defining a global variable in the Head portion of the framesetting document, initializing it as an empty string:

```
<script type="text/javascript">
outlineState = "";
</script>
```

Now you can modify the two functions in the outliner page as follows:

```
// ** functions that get and set state data **
// set cookie data
var mycookie = document.cookie;
function setCurrState(setting) {
   mycookie = parent.outlineState = setting;
}
// retrieve cookie data
function getCurrState() {
   return parent.outlineState;
}
```

Notice that there is no need for the label that has to be assigned to a cookie. The variable name keeps this data separate from the rest of the script space.

The only downside to not using a cookie is that the outline state is not preserved if the frameset goes away. If the user revisits the frameset in the same session, the outline state will be reinitialized at its beginning state.

Expanding/collapsing all at once

If you have an extensive outline, you may want to provide a shortcut to the user to expand everything at once or close up the entire outline. Because the string of `1`s and `0`s maintains the state of the outline, you can use the `db` array to help you create a new state string, and then apply it to the page. Here are two functions that do the job:

```
function expandAll() {
   expState = "";
   for (i = 1; i < db.length; i++) {
      expState += (db[i].mother) ? "1" : "0";
   }
```

```
   setCurrState(expState);
   history.go(0);
}

function collapseAll() {
   collState = "";
   for (i = 1; i < db.length; i++) {
      collState += "0";
   }
   setCurrState(collState);
   history.go(0);
}
```

All you need are a couple of buttons to invoke these functions, and you're in business.

Reducing server access

Through the lifetime of this outliner application, it has seen wildly different behaviors of the various browsers with regard to how much the browser reaches out to the server for each redisplay of the outline. Although the `history.go(0)` type of reloading is supposed to be the least onerous, some browsers seem to read the entire file from scratch. This approach is still faster than having a CGI script completely reconfigure a page, but for an extensive outline and a slow Internet connection, the results can be objectionable.

One possible solution is to avoid reloading the page at all. Instead, place all of the code for the outliner management and creation in the framesetting document. Code that currently writes the outline as the page loads can be encapsulated in a function that writes to the frame designated as the outline frame (don't forget the `document.close()` for this writing!). Function calls from the outliner (to `toggle()`, for instance) have to be modified so that the reference is to the function in the parent frame (`parent.toggle(`*n*`)`).

Distributing the code around frames may not be as convenient as keeping it all together, but user experience should weigh more heavily than programmer expedience. This practice also opens the possibility for putting all of the outliner code, except for the calls to the constructor functions, in an external `.js` library. You can then put multiple outline contents into multiple `.js` libraries and load the pairs that you need into a frameset.

Using `document.write()` to another frame may still not avoid server access entirely. It is not uncommon for the application of any image file—including those that have been precached—to check the cached version against the modification date of the file on the server. This activity is much faster than downloading the image again, but if you see network activity even after shifting the outliner's scripts to the frameset, at least you understand what's happening. A version of the application directed from the parent window is contained on the CD-ROM.

Multiple outlines

The example in this chapter assumes that a site will be using only one outline-style table of contents. You can, of course, have multiple outlines for different sections of a Web site or application. But if the outlines all share the same cookie data, the state of the most recent outline will be applied to the next one that loads. Items will be magically opened. And if the number of items between the two outlines is different, the cookie data can get a bit messy.

To solve this problem, assign a different cookie label for each outline. That prevents one outline's state from stepping on another.

Cascading Style Sheet Version

The advent of Cascading Style Sheets (CSS) brought a number of intriguing possibilities for an application, such as the outliner. Not only can stylesheets be used to control the look of the items in the outline, but additional properties make it possible to hide and show elements, including inserting or removing elements from the rendered content. Alas, not all of these features work in NN4, so the version under discussion in this section resorts to redrawing the outline for NN4. But for IE4+ and NN6+/W3C DOMs, the response is very fast, and no page reloading is necessary. One of the goals, too, in this application was to reuse as much of the code from earlier versions as possible. Note that this version does not work (or work correctly) with browsers prior to NN4 or IE4.

CSS implementation plan

Many of the compromises in this version resulted from quirky behavior of NN4 with some types of elements and stylesheets. I chose to render the outline content as a series of nested `div` elements. If this were being implemented strictly in more well-behaved browsers, stylesheet control over `ul` and `li` elements would be even more convenient because those elements already have an indentation scheme built into them. With so much HTML code needed to generate the `div` elements and their contents, I decide to trade the cleverness of multidimensional array storage of outline content for the better performance of straight HTML. Each row of content in the outline is set in its own `<div>` block tag set. Any row that had children nested inside contains those items as a nested block.

Stylesheets afforded the design a handy behavior. Hiding and showing blocks via the CSS-Positioning `visibility` property (see Chapter 26) is not an apt solution here, because hiding an item does not remove it from the page rendering. Therefore, unless the page included a ton of positioning code to overlap hidden items with visible items (which would have worked in NN4, but at the price of substantial increases in code and inflexibility), the outline would not cinch up if a branch is collapsed. To the rescue comes the `display` property of a style. One value of this property (`none`) not only hides the block, but it temporarily removes it from the rendering order of the page. Any items rendered below it that are visible (that is, whose `display` property is set to `block`) scoot up to render after the previous visible item.

Setting the `display` property has slightly different results in NN4 and more modern DOMs. In NN4, you can set the property after the block has been rendered on the page, but its appearance does not change; in both the IE4+ and NN6+/W3C DOMs, the change is immediate, with the rest of the page reflowing to adjust to the change in the block's visibility and presence. Therefore, for NN4, the page still needs to reload itself and remember the state of the outline between reloads (via the same cookie mechanism used for the earlier version) so that the page can set the property value as the page loads. And except for only a couple of places in the code, both the IE4+ and NN6+/W3C DOMs share positioning code.

The CSS version uses the same cookie value (a sequence of `1` and `0` values) to represent the visible or hidden state of each item as in the old version. To convey the change of state, however, the function called by the click of an icon widget must pass the index values of the child items affected by the expansion or collapse of a node. This means that more of the HTML — in this case, the parameters of the functions — has to be hard-wired to the structure of the outline, as you see shortly. Less of this would be necessary if NN4's implementation of CSS offered the same level of scriptable introspection into HTML elements as IE4's implementation:

We'd be able to employ the style property inheritance behavior to simplify the way blocks are shown and hidden. Because the two classes of browsers supported in this example are so different in this regard, the scripting reflects the lowest common denominator for controlling the toggle of expanded and collapsed states.

The CSS code

By putting so much of the content directly into HTML, the scripting component of the CSS outliner version is significantly smaller than the older version. Where possible, I stayed with the same function and variable naming schemes of the previous version.

At the top of the document, I define three styles for the amount of indentation required by the three indentation levels of my sample outline. If the outline were to go to more levels, I would add styles accordingly:

```
<html>
  <head>
     <title></title>
     <style type="text/css">
     div.indent0 {margin-left:0}
     div.indent1 {margin-left:10}
     div.indent2 {margin-left:20}
     </style>
```

Scripting begins by setting some global variables. Browser-specific branching comes into play later, but in an effort to stamp out explicit version detection, the code here relies on object detection to set the requisite flags. Only browsers capable of the CSS style scripting needed here have a `document.styleSheets` property, so flags are set for the two supported browser classes. These flags are set here primarily as a convenience for writing branching code later. Rather than constantly retesting for the presence of the property, the global flags are shorter and marginally faster. Two more variables hold their respective browser class state values, with the NN4 version maintaining a copy of the cookie as a variable for performance reasons:

```
     <script type="text/javascript">
     // global variables
     var isNN4, isCSS, CSScurrState, NN4Cookie = document.cookie;
     if (document.styleSheets) {
        isCSS = true;
        isNN4 = false;
     } else {
        isCSS = false;
        isNN4 = true;
     }
```

To each of the cookie storage functions from the original version, I add a branch to handle the storage and retrieval of state data for CSS browsers, simply setting and getting the global variable. This may seem to be more indirect than is necessary, but it is essential to allow the reuse of many functions in other parts of the code so that those areas don't have to worry about browser platform. Notice that the label for this outline's cookie is slightly different from that of the earlier version. This difference allows you to open both outliners with NN4 in the same session and not worry about one cookie value overlapping with the other:

```
     // ** functions that get and set persistent data **
     // set persistent data
     function setCurrState(setting) {
        if (isNN4) {
           NN4Cookie = document.cookie = "currState2=" + escape(setting);
        } else {
```

```
            // for CSS, data is saved as a global variable instead of cookie
            CSScurrState = setting;
         }
      }

      // retrieve persistent data
      function getCurrState() {
         if (isCSS) {
            // for CSS, data is in global var instead of cookie
            return CSScurrState;
         }
         var label = "currState2=";
         var labelLen = label.length;
         var cLen = NN4Cookie.length;
         var i = 0;
         while (i < cLen) {
            var j = i + labelLen;
            if (NN4Cookie.substring(i,j) == label) {
               var cEnd = NN4Cookie.indexOf(";",j);
               if (cEnd ==   -1) {
                  cEnd = NN4Cookie.length;
               }
               return unescape(NN4Cookie.substring(j,cEnd));
            }
            i++;
         }
         return "";
      }
```

The `toggle()` function is called by the `onclick` event handler of the links surrounding the widget icon art in the outline. A variable number of parameters are passed to this function, so that the parameters are extracted and analyzed via the `arguments` property of the function. Both browsers with only a few small browser-specific branches use a great deal of the code. Inside the large `for` loop, a CSS branch dynamically changes the setting of the `style.display` property. For NN4, the page is reloaded after all changes to the cookie version of the state are saved. After the NN4 version goes off to reload the page, the CSS version swaps the image of the toggled widget. As a final touch, the window is given focus so that WinIE browsers lose the dotted rectangle around the clicked image:

```
      // **function that updates persistent storage of state**
      // toggles an outline mother entry, storing new value
      function toggle() {
         var newString = "";
         var expanded, n;
         // get all <DIV> tag objects in IE4/W3C DOMs
         if (document.all) {
            var allDivs = document.all.tags("div");
         } else if (document.getElementsByTagName) {
            var allDivs = document.getElementsByTagName("div");
         }
         var currState = getCurrState() // of whole outline
         // assemble new state string based on parameters passed from link
         for (var i = 0; i < arguments.length; i++) {
            n = arguments[i];
            expanded = currState.charAt(n); // of clicked item
            newString += currState.substring(0,n);
            newString += expanded ^ 1; // Bitwise XOR clicked item
            newString += currState.substring(n+1,currState.length);
```

```
            currState = newString;
            newString = "";
            if (isCSS) {
               // dynamically change display style without reloading
               if (expanded == "0") {
                  allDivs[n].style.display = "block";
               } else {
                  allDivs[n].style.display = "none";
               }
            }
         }
         setCurrState(currState); // write new state back to cookie
         if (isNN4) {
            location.reload();
         }
         // swap images in CSS versions
         var img = document.images["widget" + (arguments[0]-1)];
         img.src = (img.src.indexOf("plus.gif") != -1) ?
            "minus.gif" : "plus.gif";
         window.focus();
      }
```

A prerequisite for loading the page to begin with is setting the initial value of the state. This is the only part of the script that must be hard-wired based on the structure of the outline — string assigned to `initState` will be different with each outline. The goal here is to set each block assigned to the `indent0` style class to `1` while all others are set to `0`. These settings allow the first display of the outline to show all the root nodes, with all other items collapsed:

```
      // initialize 'current state' storage field
      if (!getCurrState()) {
         // must be hard-wired to outline structure with "1" for
         // each indent0 class item, "0" for all others
         initState = "1000010000";
         setCurrState(initState);
      }
```

With the initial outline state saved in the preceding code, the following statements execute at load time to write a `<style>` tag set for NN4. This tag sets the `display` property of all collapsed blocks to `none`. As you see in the HTML coming up, blocks are assigned `id` attributes with the letter "a" followed by a sequence number starting with zero:

```
      // for Navigator 4, set display style for flagged IDs to 'none'
      // each time the page (re)loads
      if (isNN4) {
         document.write("<style type='text/css'>");
         var visState = getCurrState();
         for (var i = 0; i < visState.length; i++) {
            if (visState.charAt(i) == "0") {
               document.write("#a" + i + " {display:none}\n");
            }
         }
         document.write("</style>");
      }
```

Initial settings of the `display` property for IE4+ can be done programmatically only after the document loads (the tags must exist before their properties can be adjusted). The following `init()` function is called from the `onload` event handler. Each browser class has a different set of initialization tasks. Both branches rely on the current state setting, so that value is

retrieved just once. In the CSS branch, the `style.display` properties for hidden blocks are set to `none`. For NN4, on the other hand, the `style.display` properties are set as the page reloads, but this loop swaps the widget image for expanded blocks to the `minus.gif` version:

```
      // for CSS, initialize flagged tags to style display = "none"
      // for NN4, set affected images to minus.gif
      function init() {
         var visState = getCurrState();
         if (isCSS) {
            for (var i = 0; i < visState.length; i++) {
               if (visState.charAt(i) == "0") {
                  if (document.all) {
                     document.all("a" + i).style.display = "none";
                  } else if (document.getElementsByTagName) {
                     document.getElementById("a" + i).style.display = "none";
                  }
               }
            }
         } else if (isNN4) {
            for (i = 0; i < visState.length; i++) {
               if (visState.charAt(i) == "1") {
                  if (i+1 < visState.length && visState.charAt(i+1) == "1") {
                     if (document.images["widget" + i]) {
                        document.images["widget" + i].src = "minus.gif";
                     }
                  }
               }
            }
         }
      }
      </script>
   </head>
   <body onload="init()">
      <center>
         <h3>Composition of Selected Foods</h3>
         <hr />
      </center>
```

Now begins the HTML that defines the content of the outline. For readability, I have formatted the `<div>` tag sets to follow the indentation of the outline data (this listing looks much better if you open the file from the CD-ROM in your text editor with word wrap turned off). Each tag includes a `class` attribute pointing to a class defined in the first `<style>` tag of the page. Each tag also includes an `id` attribute whose name begins with the letter "a" and a sequential serial number, starting with zero. Navigator uses the `id` attributes to help it assign `display` property settings during each reload.

Like the older version of the outliner, each entry includes an image (surrounded by a clickable link) and a text entry (which may or may not be a link to a document). The link around the image includes a `javascript:` URL for the `href` attribute. When a link is for a widget that is a mother item, the parameters to the `toggle()` function are the serial numbers of the immediate children IDs whose display properties are to be adjusted in the `toggle()` function. These passed items only need to be in the immediate children, because any of their children inherit the `display` property of their parents. For example, the first widget toggles items 1 and 2 (ids `a1` and `a2`). Item 2 happens to be a parent to items 3 and 4. But when the `display` property of item 2 is set to `none`, none of its children (items 3 and 4) are displayed, no matter how their display properties are set.

`img` elements associated with each toggled `div` are named along similar lines, with the name starting with "widget" and the same serial number as the containing `div`. If you look at the end of the `toggle()` function again, you'll see that the name for the `img` element is derived from the first parameter received by the `toggle()` function. That first parameter will always be one number higher than the serial number for the widget image to swap. To help you visualize the numbering scheme used within the example, the numbered identifiers and methods that relay associated numbers are shown in boldface:

```
<div class="indent0" id="a0">
   <a href="javascript:toggle(1,2)"
   onmouseover="status='Click to expand/collapse nested items';
      return true"
    onmouseout="status='';return true"><img alt="image" name="widget0"
   src="plus.gif" height="12" width="12" border="0" />
   </a> Peas<br />
   <div class="indent1" id="a1">
      <a href="javascript:void(0)"
      onmouseover="status='No further items';return true"
      onmouseout="status='';return true"><img alt="image" src="end.gif"
      height="12" width="12" border="0" /></a> <a
      href="foods.htm#boiled" target="Frame2">Boiled</a><br />
   </div>
   <div class="indent1" id="a2">
      <a href="javascript:toggle(3,4)"
      onmouseover="status='Click to expand/collapse nested items';
         return true"
       onmouseout="status='';return true"><img alt="image" name="widget2"
      src="plus.gif" height="12" width="12" border="0" /></a> <a
      href="foods.htm#canned" target="Frame2">Canned</a><br />
      <div class="indent2" id="a3">
         <a href="javascript:void(0)"
         onmouseover="status='No further items';return true"
         onmouseout="status='';return true"><img alt="image"
         src="end.gif" height="12" width="12" border="0" /></a> <a
         href="foods.htm#alaska" target="Frame2">Alaska</a><br />
      </div>
      <div class="indent2" id="a4">
         <a href="javascript:void(0)"
         onmouseover="status='No further items';return true"
         onmouseout="status='';return true"><img alt="image"
         src="end.gif" height="12" width="12" border="0" /></a> <a
         href="foods.htm#losodium" target="Frame2">Low-Sodium</a><br />
      </div>
   </div>
</div>
<div class="indent0" id="a5">
   <a href="javascript:toggle(6)"
   onmouseover="status='Click to expand/collapse nested items';
      return true"
    onmouseout="status='';return true"><img alt="image" name="widget5"
   src="plus.gif" height="12" width="12" border="0" />
   </a> Pickles<br />
   <div class="indent1" id="a6">
      <a href="javascript:toggle(7,8,9)"
      onmouseover="status='Click to expand/collapse nested items';
         return true"
       onmouseout="status='';return true"><img alt="image" name="widget6"
```

```
            src="plus.gif" height="12" width="12" border="0" /></a> <a
            href="foods.htm#cucumber" target="Frame2">Cucumber</a><br />
            <div class="indent2" id="a7">
               <a href="javascript:void(0)"
               onmouseover="status='Click to expand nested items';return true"
               onmouseout="status='';return true"><img alt="image"
               src="end.gif" height="12" width="12" border="0" /></a> <a
               href="foods.htm#dill" target="Frame2">Dill</a><br />
            </div>
            <div class="indent2" id="a8">
               <a href="javascript:void(0)"
               onmouseover="status='No further items';return true"
               onmouseout="status='';return true"><img alt="image"
               src="end.gif" height="12" width="12" border="0" /></a> <a
               href="foods.htm#fresh" target="Frame2">Fresh</a><br />
            </div>
            <div class="indent2" id="a9">
               <a href="javascript:void(0)"
               onmouseover="status='No further items';return true"
               onmouseout="status='';return true"><img alt="image"
               src="end.gif" height="12" width="12" border="0" /></a> <a
               href="foods.htm#sour" target="Frame2">Sour</a><br />
            </div>
         </div>
      </div>
   </body>
</html>
```

The CSS version (for the identical outline content) is a slightly smaller file size than the older, compatible one, but not so big a difference as to influence your choice. Browser compatibility should be your number one criterion. Ease of modification for changing content and improved user experience for browsers following the CSS branch are tied in second.

A Futuristic (XML) Outline

Only a handful of browsers — WinIE5+ and Mozilla-based browsers — are capable of working directly with XML data, but I believe this capability will grow more important as the concept of "Web services" becomes more ingrained in the mainstream internet. Therefore, I've modernized the many generations of scriptable outliner described earlier in this chapter to use the `XMLHttpRequest` object described in Chapter 41. The data for this outline version is set apart in a more easily maintainable XML data file. At the same time, I've "prettied up" the hierarchical interface of the outliner display.

Birth of an XML specification

Collapsible outlines provide convenient ways to organize hierarchical information all around us. You'd be hard-pressed to find a more active proponent of the outline than Dave Winer, founder of UserLand Software, Inc. (`http://www.userland.com`). Dave is a veteran software developer, as well as author and outspoken Web publisher. His `www.scripting.com` Web site is a popular destination if you want to find out the latest Internet and computing technology "buzz."

As an outgrowth of development for his company's Web tools, Dave looked to the XML structure to assist in representing outline content in a shareable, easily parseable format. The result is a specification called Outline Processor Markup Language, or OPML for short. You can read all about the formal specification at `http://www.opml.org/spec`. Like virtually all XML, OPML is intended to be written by software, not humans (although humans input the data via a user-friendly front-end provided by the software). Even so, the format of an OPML outline is extremely readable by humans, and, with little more trouble than writing basic HTML tags manually, you can represent an outline in this format yourself.

A plain OPML file, saved as an `.xml` file, can be viewed through the native XML parsers of IE5+ and NN6+/Moz. These parsers automatically render XML tags in the same hierarchical fashion as OPML encourages outlines to be structured. But such rendering is under strict control of the browser, unless you also get involved with XML stylesheets (the XSL and XSLT standards), at which point, browser implementation incompatibilities can make the going tough.

I liked the OPML data format when I first saw it, and I think it's a convenient way to convey an outline's data to the client, at which point JavaScript and the browser's DOM can take over to provide interesting visuals for the content and interaction with the content. Thus was born this last example of the chapter, in which the outliner's data is delivered not in the form of scripted arrays or hard-wired HTML `div` elements. Instead, the data arrives in its native XML (OPML) format from a separate file. Scripts take over to do the rest.

OPML outliner prep

The appearance of widgets and text for the new outliner has changed to more closely emulate the kinds of outline presentations that you see in some Windows programs (see Figure 52-2). For demonstration purposes, the same frameset structure and outline content from earlier examples are used for the OPML version so that you can more easily see the differences in implementations and grasp new concepts presented here. For example, the comparison of how the outline data is delivered in the form of JavaScript objects (the first example) and OPML is enlightening.

As you recall, a custom object constructor function generated one JavaScript object for each outline entry. The properties of the object are completely under your control, so that you can add properties (such as the target of an entry's link) whose values influence the way the entry is rendered and the way it behaves. OPML has a similar extensibility feature. Each outline entry is nothing more than a tag. An entry that does not have any nested child nodes can use the XML shortcut of combining a start and end tag inside one set of angle brackets:

```
<tagName attribute="value" ... />
```

And any entry that has nested nodes contains the nested nodes between its start and end tags, as shown here with the actual tag names used in OPML (indentation is optional, but increases readability):

```
<outline text="text">
   <outline text="text" />
   <outline text="text" />
</outline>
```

If you want to associate more information about an entry, simply add an attribute. For example, if an entry is to behave as a link, you can convey that information with an attribute whose name you determine. When it comes time for your scripts to render the content in HTML, the scripts access the attribute values and generate the associated HTML for the attributes (you see an example of this in the code).

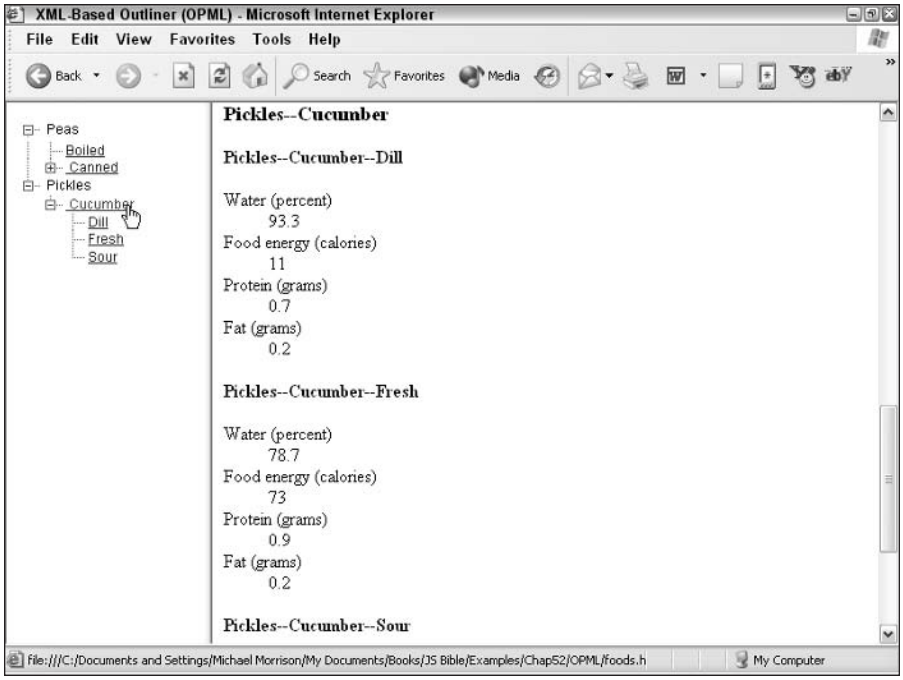


Figure 52-2: OPML-based outliner style.

The true beauty of the OPML structure (and XML in general) is that the parent–child relationships are automatically implied by the element containment. Unlike the JavaScript custom object in the first example, the author does not have to specify how many levels deep an entry is, or whether it has any child nodes: The XML containment hierarchy describes all of that information. Suddenly, all of the W3C DOM gobbledygook about nodes, child nodes, and attributes become your friend, as your scripts convert the element hierarchy into a renderable hierarchy of your design.

The XML and HTML code

Because our focus is so tight on the outliner content, the first code to view is that of the XML document, named `outlineData.xml`:

```
<?xml version="1.0" ?>
<opml version="1.0">
   <head>
      <title>A Modern Outline</title>
      <dateCreated>Thu, 13 Nov 2003 02:40:00 GMT</dateCreated>
      <dateModified>Fri, 19 Dec 2003 19:35:00 GMT</dateModified>
      <ownerName>Danny Goodman</ownerName>
      <ownerEmail>dannyg@dannyg.com</ownerEmail>
      <expansionState></expansionState>
      <vertScrollState>1</vertScrollState>
      <windowTop></windowTop>
      <windowLeft></windowLeft>
      <windowBottom></windowBottom>
      <windowRight></windowRight>
```

```
   </head>
   <body>
      <outline text="Peas">
         <outline text="Boiled" uri="foods.htm#boiled"/>
         <outline text="Canned" uri="foods.htm#canned">
            <outline text="Alaska" uri="foods.htm#alaska"/>
            <outline text="Low-Sodium" uri="foods.htm#losodium"/>
         </outline>
      </outline>
      <outline text="Pickles">
         <outline text="Cucumber" uri="foods.htm#cucumber">
            <outline text="Dill" uri="foods.htm#dill"/>
            <outline text="Fresh" uri="foods.htm#fresh"/>
            <outline text="Sour" uri="foods.htm#sour"/>
         </outline>
      </outline>
   </body>
</opml>
```

This file is textbook OPML version 1.0 form. Notice that the OPML syntax reuses element names that are found in all HTML files (for example, `head`, `title`, `body`). This element name duplication makes it essential to isolate the XML from any HTML rendering area (such as an `iframe`), which would automatically add its own `body` element and thus destroy the OPML node tree.

The HTML body of the `toc6.html` file is sparse to say the least:

```
   <body onload="init('outlineData.xml')">
   <div id="content"></div>
   </body>
```

The only other HTML delivered in the document body is an empty `div` element, which is used as the container for the outline HTML that the scripts generate as a result of the `onload` event handler's invocation of the `init()` function.

Setting the scripted stage

All scripts for this page are in the Head (although they could also be linked in from an external `.js` file). First on the docket is establishing several global variables that get used a lot within the rest of the code and make it easy to customize important visible properties, especially widget art. Due to the art choices made for this version, there are separate versions for items that appear as first, middle, and end items for different nesting states:

```
<script type="text/javascript">
// global variables
// art files and sizes for widget styles and spacers
// (all images must have same height/width)
var collapsedWidget = "oplus.gif";
var collapsedWidgetStart = "oplusStart.gif";
var collapsedWidgetEnd = "oplusEnd.gif";
var expandedWidget = "ominus.gif";
var expandedWidgetStart = "ominusStart.gif";
var expandedWidgetEnd = "ominusEnd.gif";
var nodeWidget = "onode.gif";
var nodeWidgetEnd = "onodeEnd.gif";
var emptySpace = "oempty.gif";
var chainSpace = "ochain.gif";
```

```
var widgetWidth = "20";
var widgetHeight = "16";
var currState = "";
var displayTarget = "Frame2";
```

The `init()` function, invoked by the `onload` event handler, starts the content creation in motion. The basic sequence is to first load the external XML data into a scripted document object global variable (named `xDoc`). If the data loads successfully, a reference to the Body portion of the outline data is retrieved so that many other functions are able to dive into the outliner hierarchy. The reference to the OPML `body` element is passed to the `makeHTML()` function, which returns the entire outline HTML to be assigned to the `innerHTML` property of the empty `div` element delivered with the document:

```
// initialize first time
function init(outlineDataURL) {
   loadXMLDoc(outlineDataURL);
   if (xDoc) {
      // demo how to get outline head elements
      var hdr = xDoc.getElementsByTagName("head")[0];
      // get outline body elements for iteration and conversion to HTML
      var ol = xDoc.getElementsByTagName("body")[0];
      // wrap whole outline HTML in a span
      var olHTML = "<span id='renderedOL'>" + makeHTML(ol) + "</span>";
      // throw HTML into 'content' div for display
      document.getElementById("content").innerHTML = olHTML;
      initExpand();
   }
}
```

Loading the XML file is performed in the `loadXMLDoc()` function. The function also handles alerts to users due to a missing XML file, connection problems, or lack of support for the feature:

```
// XML document
var xDoc;

// retrieve XML document as document object
function loadXMLDoc(url) {
   var req;
   if (window.XMLHttpRequest) {
      req = new XMLHttpRequest();
      req.open("GET", url, false);
      req.send(null);
   } else if (window.ActiveXObject) {
      req = new ActiveXObject("Microsoft.XMLHTTP");
      if (req) {
         req.open("GET", url, false);
         req.send();
      }
   }
   if (req) {
      if (req.status == 200) {
         xDoc = req.responseXML;
         if (xDoc && typeof xDoc.childNodes != "undefined" &&
            xDoc.childNodes.length == 0) {
            xDoc = null;
         }
```

```
      } else {
         alert("There was a problem retrieving the XML data:\n" +
         req.statusText);
      }
   } else {
      alert("Sorry, this browser isn\'t equipped to read XML data.");
   }
}
```

Accumulating the HTML

From the `init()` function, a call to the `makeHTML()` function starts the most complex actions of the scripts on this page. This function walks the node hierarchy of the outline's `body` elements, deciphering which ones are containers and which ones are end points.

Two global variables are used to keep track of how far the node walk progresses because this function calls itself from time to time to handle nested branches of the node tree. Because a reflexive call to a function starts out with new values for local variables, the globals operate as pointers to let statements in the function know which node is being accessed. The numbers get applied to an `id` attribute assigned to the `div` elements holding the content.

One of the fine points of the design of this outline is the way space to the left of each entry is assembled. Unlike the earlier outlines in this chapter, this one displays vertical dotted lines connecting nodes at the same level. There isn't a vertical line for every clickable node appearing above the item, because a clickable node may have no additional siblings, meaning that the space is blank. To see what I mean, open the OPML example, and expand the Peas and Canned nodes (or see Figure 52-2). The Canned node is the end of the second "column," so the space beneath the icon is blank. That's what some of the code in `makeHTML()` named "prefix" is dealing with: Accumulating just the right combination of dotted line (`chain.gif`) and blank (`empty.gif`) images in sequence before the outline entry.

Another frequent construction throughout this function is a three-level conditional expression. This construction is used to determine whether the image just to the left of the item's text should be a start, middle, or end version of the image. The differences among them are subtle (having to do with how the vertical dotted line extends above or below the widgets). All of these decisions are made from information revealed by the inherent structure of the OPML element nesting. The listing in the book looks longer than it truly is because so many long or deeply nested lines must be wrapped to the next line. Viewing the actual file in your text editor should calm your fears a bit.

```
// counters for reflexive calls to makeHTML()
var currID = 0;
var blockID = 0;
// generate HTML for outline
function makeHTML(outlineID, ol, prefix) {
   var output = "";
   var nestCount, link, nestPrefix;
   prefix = (prefix) ? prefix : "";
   for (var i = 0; i < ol.childNodes.length ; i++) {
      nestCount = ol.childNodes[i].childNodes.length;
      output += "<div class='row' id='line" + currID++ + "'>\n";
      if (nestCount > 0) {
         output += prefix;
         output += "<img id='widget" + (currID-1) + "' src='" +
```

```
            ((i== ol.childNodes.length-1) ? collapsedWidgetEnd : (blockID==0) ?
            collapsedWidgetStart : collapsedWidget);
         output += "' height=" + widgetHeight + " width=" + widgetWidth;
         output += " title='Click to expand/collapse nested items.'
            onclick='toggle(this," + blockID + ")'>";
         link =  (ol.childNodes[i].getAttribute("uri")) ?
            ol.childNodes[i].getAttribute("uri") : "";
         if (link) {
            output += " <a href='" + link + "'
               class='itemTitle' title='" +
               link + "' target='" + displayTarget + "'>";
         } else {
            output += " <a class='itemTitle' title='" + link + "'>";
         }
         output += " " + ol.childNodes[i].getAttribute("text") + "</a>";
         currState += calcBlockState(outlineID, currID-1);
         output += "<span class='OLBlock' blocknum='" + blockID +
            "' id='OLBlock" + blockID++ + "'>";
         nestPrefix = prefix;
         nestPrefix += (i == ol.childNodes.length - 1) ?
            "<img src='" + emptySpace + "' height=16 width=20>" :
            "<img src='" + chainSpace + "' height=16 width=20>";
         output += makeHTML(outlineID, ol.childNodes[i], nestPrefix);
         output += "</span></div>\n";
      } else {
         output += prefix;
         output += "<img id='widget" + (currID-1) + "' src='" +
            ((i == ol.childNodes.length - 1) ? nodeWidgetEnd : nodeWidget);
         output += "' height=" + widgetHeight + " width=" + widgetWidth + ">";
         link =  (ol.childNodes[i].getAttribute("uri")) ?
            ol.childNodes[i].getAttribute("uri") : "";
         if (link) {
            output += " <a href='" + link + "'
               class='itemTitle' title='" +
               link + "' target='" + displayTarget + "'>";
         } else {
            output += " <a class='itemTitle' title='" + link + "'>";
         }
         output += ol.childNodes[i].getAttribute("text") + "</a>";
         output += "</div>\n";
      }
   }
   return output;
}
```

As with the HTML assembly code of the first outliner, if you were to add attributes to `outline` elements in an OPML outline (for example, a URL for an icon to display in front of the text), it is in `makeHTML()` that the values would be read and applied to the HTML being created.

The only other function invoked by the `makeHTML()` function is `calcBlockState()`. This function looks into one of the OPML outline's `head` elements, called `expansionstate`. This element's values can be set to a comma-delimited list of numbers corresponding to nodes that are to be shown expanded when the outline is first displayed. The `calcBlockState()` function is invoked for each parent element. The element's location is compared against values in the `expansionstate` element, if there are any, and returns the appropriate `1` or `0` value for the state string being assembled for the rendered outline:

```
// apply default expansion state from outline's header
// info to the expanded state for one element to help
// initialize currState variable
function calcBlockState(outlineID, n) {
   var ol = document.getElementById(outlineID).getElementsByTagName("body")[0];
   var outlineLen = ol.getElementsByTagName("outline").length;
   // get OPML expansionState data
   var expandElem = document.getElementById(
      outlineID).getElementsByTagName("expansionState")[0];
   var expandedData = (expandElem.childNodes.length) ?
      expandElem.firstChild.nodeValue.split(",") : null;
   if (expandedData) {
      for (var j = 0; j < expandedData.length; j++) {
         if (n == expandedData[j] - 1) {
            return "1";
         }
      }
   }
   return "0";
}
```

The final act of the initialization process is a call to the `initExpand()` function. This function loops through the `currState` global variable (whose value was written in `makeHTML()` with the help of `calcBlockState()`) and sets the `display` property to `block` for any element designed to be expanded at the outset. HTML element construction in `makeHTML()` is performed in such a way that each parent `div` has a `span` nested directly inside of it; and inside that `span` are all the child nodes. The `display` property of the `span` determines whether all of those children are seen or not.

```
// expand items set in expansionState XML tag, if any
function initExpand(outlineID) {
   for (var i = 0; i < currState.length; i++) {
      if (currState.charAt(i) == 1) {
         document.getElementById("OLBlock" + i).style.display = "block";
      }
   }
}
```

By the time the `initExpand()` function has run — a lot of setup code that executes pretty quickly — the rendered outline is in a steady state. Users can now expand or collapse portions by clicking the widget icons.

Toggling node expansion

All of the widget images in the outline have `onclick` event handlers assigned to them. The handlers invoke the `toggle()` function, passing parameters consisting of a reference to the `img` element object receiving the event and the serial number of the `span` block nested just inside the `div` that holds the image. With these two pieces of information, the `toggle()` function sets in motion the act of inverting the expanded/collapsed state of the element and the plus or minus version of the icon image. The `blockNum` parameter corresponds to the position within the `currState` string of `1`s and `0`s holding the flag for the expanded state of the block. With the current value retrieved from `currState`, the value is inverted through `swapState()`. Then, based on the new setting, the `display` property of the block is set accordingly, and widget art is changed through two special-purpose functions:

```
// toggle an outline mother entry, storing new state value;
// invoked by onclick event handlers of widget image elements
function toggle(img, blockNum) {
   var newString = "";
   var expanded, n;
   // modify state string based on parameters from img
   expanded = currState.charAt(blockNum);
   currState = swapState(currState, expanded, blockNum);
   // dynamically change display style
   if (expanded == "0") {
      document.getElementById("OLBlock" + blockNum).style.display = "block";
      img.src = getExpandedWidgetState(img.src);
   } else {
      document.getElementById("OLBlock" + blockNum).style.display = "none";
      img.src = getCollapsedWidgetState(img.src);
   }
}
```

Swapping the state of the `currState` variable utilizes the same XOR operator employed by the first outliner in this chapter. The entire `currState` string is passed as a parameter. The indicated digit is segregated and inverted, and the string is reassembled before being returned to the calling statement in `toggle()`:

```
// invert state
function swapState(currState, currVal, n) {
   var newState = currState.substring(0,n);
   newState += currVal ^ 1; // Bitwise XOR item n
   newState += currState.substring(n+1,currState.length);
   return newState;
}
```

As mentioned earlier, each of the clickable widget icons (plus and minus) can be one of three states, depending on whether the widget is at the start, middle, or end of a vertical-dotted chain. The two image swapping functions find out (by inspecting the URLs of the images currently occupying the `img` element) which version is currently in place so that, for instance, a starting plus (collapsed) widget is replaced with a starting minus (expanded) widget. This is a case of going the extra mile for the sake of an improved user interface:

```
// retrieve matching version of 'minus' images
function getExpandedWidgetState(imgURL) {
   if (imgURL.indexOf("Start") != -1) {
      return expandedWidgetStart;
   }
   if (imgURL.indexOf("End") != -1) {
      return expandedWidgetEnd;
   }
   return expandedWidget;
}

// retrieve matching version of 'plus' images
function getCollapsedWidgetState(imgURL) {
   if (imgURL.indexOf("Start") != -1) {
      return collapsedWidgetStart;
   }
   if (imgURL.indexOf("End") != -1) {
      return collapsedWidgetEnd;
   }
      return collapsedWidget;
}
```

Wrap up

There's no question that the amount and complexity of the code involved for the OPML version of the outliner are significant. The "pain" associated with developing an application such as this is all up front. After that, the outline content is easily modifiable in the OPML format (or perhaps by some future editor that produces OPML output).

Even if you don't plan to implement an OPML outline, the explanation of how this example works should drive home the importance of designing data structures that assist not only the visual design but also the scripting that you use to manipulate the visual design.

Further Thoughts

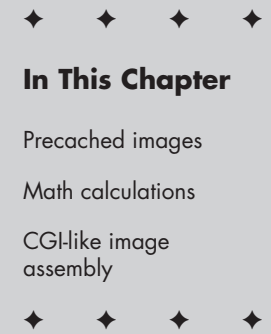
The advent of CSS and element positioning has prompted numerous JavaScripters to develop another kind of hierarchical system of pop-up or drop-down menus. You can find examples of this interface at many of the JavaScript source Web sites listed in Appendix D. Making these kinds of menus work well in NN4, IE4+, and NN6+/W3C DOM browsers is a lot of hard work, and if you can adopt code already ironed out by others, then all the better.

Most of the code you find, however, will require a fair amount of tweaking to blend the functionality into the visual design that you have or are planning for your Web site. Finding two implementations on the Web that look or behave the same way is rare. As long as you're aware of what you'll be getting yourself into, you are encouraged to check out these interface elements. By hiding menu choices except when needed, valuable screen real estate is preserved for more important, static content.

✦ ✦ ✦

CHAPTER 53

Application: Calculations and Graphics



When the scripting world had its first pre-release peeks at JavaScript (while Netscape was still calling the language LiveScript), the notion of creating interactive HTML-based calculators captured the imaginations of many page authors. Somewhere on the World Wide Web, you can find probably every kind of special-purpose calculation normally done by scientific calculators and personal computer programs — leaving only weather-modeling calculations to the supercomputers of the world.

In the search for my calculator gift to the JavaScript universe, I looked around for something more graphical. Numbers, by themselves, are pretty boring; so any way the math could be enlivened was fine by me. Having been an electronics hobbyist since I was a kid, I recalled the color-coding of electronic resistor components. The values of these gizmos aren't printed in plain numbers anywhere. You have to know the code and the meaning of the location of the colored bands to arrive at the value of each one. I thought that this calculator would be fun to play with, even if you don't know what a resistor is.

The Calculation

To give you an appreciation for the calculation that goes into determining a resistor's value, here is the way the system works. Three closely spaced color bands determine the resistance value in ohms. The first (leftmost) band is the tens digit; the second (middle) band is the ones digit. Each color has a number from 0 through 9 assigned to it (black = 0, brown = 1, and so on). Therefore, if the first band is brown and the second band is black, the number you start off with is 10. The third band is a multiplier. Each color determines the power of ten by which you multiply the first digits. For example, the red color corresponds to a multiplier of 102, so that 10×102 equals 1,000 ohms.

A fourth band, if present, indicates the tolerance of the component — how far, plus or minus, the resistance measurement can fluctuate due to variations in the manufacturing process. Gold means a tolerance of plus-or-minus 5 percent; silver means plus-or-minus 10 percent; and no band means a 20 percent tolerance. A pinch of extra space typically appears between the main group of three-color bands and the one tolerance band.

User Interface Ideas

The quick-and-dirty, non-graphical approach for a user interface was to use a single frame with four `select` elements defined as pop-up menus (one for each of the four color bands on a resistor), a button to trigger calculation, and a field to show the numeric results.

How dull.

It occurred to me that if I design the art carefully, I can have JavaScript assemble an updated image of the resistor consisting of different slices of art: static images for the resistor's left and right ends, and variable slivers of color bands for the middle. Rather than use the brute force method of creating an image for every possible combination of colors (3,600 images total!), a far more efficient approach is to have one image file for each color (12 colors plus 1 empty) and enable JavaScript to call them from the server, as needed, in the proper order. If not for client-side JavaScript, a CGI script on the server would have to handle this kind of intelligence and user interaction. But with this system, any dumb server can dish up the image files as called by the JavaScript script.

The first generation of this resistor calculator used two frames, primarily because I needed a second frame to update the calculator's art dynamically while keeping the pop-up color menus stationary. Images couldn't be swapped back in those frontier days, so the lower frame had to be redrawn for each color choice. Fortunately, modern browsers enabled me to update individual image objects in a loaded document without any document reloading. Moreover, with all the images precached in memory, page users experience no (or virtually no) delay in making a change from one value to another.

The design for the new version is a simple, single-document interface (see Figure 53-1). Four pop-up menus let you match colors of a resistor, whereas the `onchange` event handler in each menu automatically triggers an image and calculation update. To hold the art together on the page, a table border surrounds the images on the page, whereas the numeric value of the component appears in a text field.

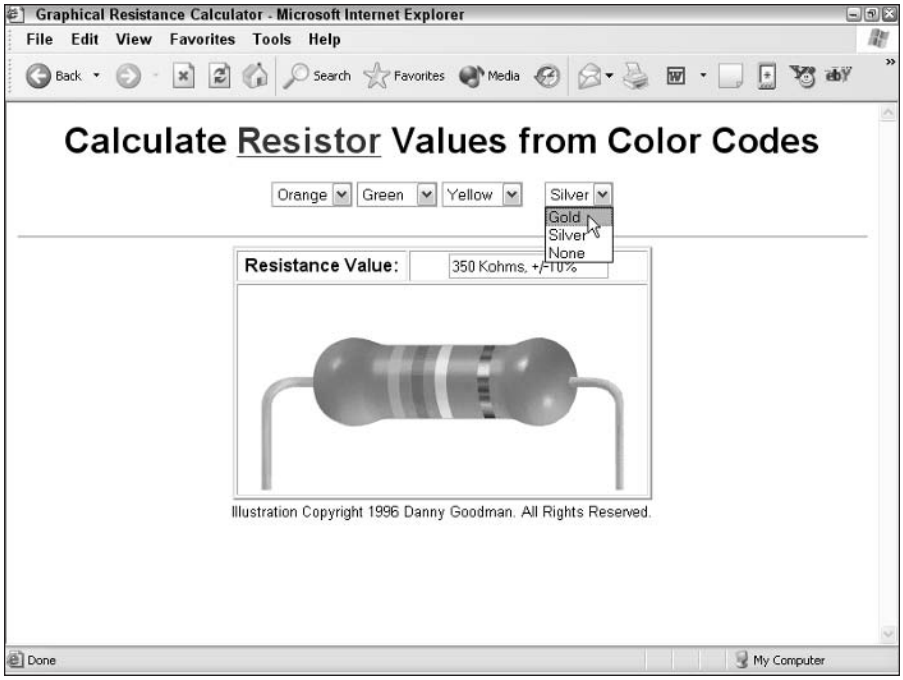


Figure 53-1: The Resistor Calculator with images inside a table border.

The Code

All the action takes place in the file named `resistor.htm`. A second document is an introductory HTML text document that explains what a resistor is and why you need a calculator to determine a component's value. The article, called *The Path of Least Resistance*, can be viewed in a secondary window from a link in the main window. Here you will be looking only at `resistor.htm`, which has been updated to include stylesheets.

The document begins in the traditional way without any surprises:

```
<html>
   <head>
      <title>Graphical Resistance Calculator</title>
      <style type="text/css">
      body {font-family:Arial, Helvetica, serif}
      </style>
      <script type="text/javascript">
```

Basic arrays

In calculating the resistance, the script needs to know the multiplier value for each color. If not for the last two multiplier values actually being negative multipliers (for example, 10-1 and 10-2), I could have used the index values without having to create this array. But the two out-of-sequence values at the end make it easier to work with an array rather than to try special-casing these instances in later calculations:

```
// create array listing all the multiplier values
var multiplier = new Array();
multiplier[0] = 0;
multiplier[1] = 1;
multiplier[2] = 2;
multiplier[3] = 3;
multiplier[4] = 4;
multiplier[5] = 5;
multiplier[6] = 6;
multiplier[7] = 7;
multiplier[8] = 8;
multiplier[9] = 9;
multiplier[10] = -1;
multiplier[11] = -2;

// create array listing all tolerance values
var tolerance = new Array();
tolerance[0] = "+/-5%";
tolerance[1] = "+/-10%";
tolerance[2] = "+/-20%";
```

Although the script doesn't do any calculations with the tolerance percentages, it needs to have the strings corresponding to each color for display in the pop-up menu. The `tolerance` array is there for that purpose.

Calculations and formatting

Before the script displays the resistance value, it needs to format the numbers in values that are meaningful to those who know about these values. Just as measures of computer storage bytes, high quantities of ohms are preceded with "kilo" and "meg" prefixes, commonly abbreviated with the "K" and "M" letters. The `format()` function determines the order of magnitude of the final calculation (from another function shown in the following section) and formats the results with the proper unit of measure:

```
// format large values into kilo and meg
function format(ohmage) {
   if (ohmage >= 1e6) {
      ohmage /= 1e6;
      return "" + ohmage + " Mohms";
   } else {
      if (ohmage >= 1e3) {
         ohmage /= 1e3;
         return "" + ohmage + " Kohms";
      } else {
         return "" + ohmage + " ohms";
      }
   }
}
```

The selections from the pop-up menus meet the calculation formulas of resistors in the `calcOhms()` function. Because this function is triggered indirectly by each of the `select` objects, sending any of their parameters to the function is a waste of effort. Moreover, the `calcOhms()` function is invoked by the `onload` event handler, which is not tied to the form or its controls. Therefore, the function obtains the reference to the form and then extracts the necessary values of the four `select` objects by using explicit (named) references. Each value is stored in a local variable for convenience in completing the ensuing calculation.

Recalling the rules used to calculate values of the resistor bands, the first statement of the calculation multiplies the "tens" pop-up value times 10 to determine the tens digit and then adds the ones digit. From there, the combined value is multiplied by the exponent value of the selected multiplier value. Notice that the expression first assembles the value as a string to concatenate the exponent factor and then evaluates it to a number. Although I try to avoid the `eval()` function because of its slow performance, the one call per calculation is not a performance issue at all. The evaluated number is passed to the `format()` function for proper formatting (and setting of order of magnitude). In the meantime, the tolerance value is extracted from its array, and the combined string is plugged into the result text field (which is in a separate form, as described later):

```
// calculate resistance and tolerance values
function calcOhms() {
   var form = document.forms["input"];
   var d1 = form.tensSelect.selectedIndex;
   var d2 = form.onesSelect.selectedIndex;
   var m = form.multiplierSelect.selectedIndex;
   var t = form.toleranceSelect.selectedIndex;
   var ohmage = (d1 * 10) + d2;
   ohmage = eval("" + ohmage + "e" + multiplier[m]);
   ohmage = format(ohmage);
   var tol = tolerance[t];
   document.forms["output"].result.value = ohmage + ", " + tol;
}
```

Preloading images

As part of the script that runs when the document loads, the next group of statements precaches all possible images that can be displayed for the resistor art. For added scripting convenience, the color names are loaded into an array. With the help of that just-created array of color names, you then create another array (`imageDB`), which both generates `Image` objects for each image file and assigns a URL to each image. Notice an important subtlety about the index values being used to create each entry of the `imageDB` array: Each index is a `colorArray` entry, which is the name of the color. As you found out in Chapter 30, if you create an array element with a named index, you must use that style of index to retrieve the data thereafter; you cannot switch arbitrarily between numeric indexes and names. As you see in a moment, this named index provides a critical link between the choices a user makes in the pop-up lists and the image objects being updated with the proper image file.

```
// pre-load all color images into image cache
var colorArray = new Array("Black","Blue","Brown","Gold","Gray",
   "Green","None","Orange","Red","Silver","Violet","White","Yellow");
var imageDB = new Array();
for (i = 0; i < colorArray.length; i++) {
   imageDB[colorArray[i]] = new Image(21,182);
   imageDB[colorArray[i]].src = colorArray[i] + ".gif";
}
```

The act of assigning a URL to the `src` property of an `Image` object instructs the browser to pre-load the image file into memory. This pre-loading happens as the document is loading, so another few seconds of delay won't be noticed by the user.

Changing images on the fly

The next four functions are invoked by their respective `select` object's `onchange` event handler. For example, after a user makes a new choice in the first `select` object (the "tens" value color selector), that `select` object reference is passed to the `setTens()` function. Its job is to extract the text of the choice and use that text as the index into the `imageDB` array. Alternatively, the color name can also be assigned to the `value` attribute of each `option`, and the `value` property read here. You need this connection to assign the `src` property of that array entry to the `src` property of the image that you see on the page (defined in the following section). This assignment is what enables the images of the resistor to be updated instantaneously — just the one image "slice" affected by the user's choice in a `select` object:

```
      function setTens(choice) {
         var tensColor = choice.options[choice.selectedIndex].value;
         document.tens.src = imageDB[tensColor].src;
         calcOhms();
      }
      function setOnes(choice) {
         var onesColor = choice.options[choice.selectedIndex].value;
         document.ones.src = imageDB[onesColor].src;
         calcOhms();
      }
      function setMult(choice) {
         var multColor = choice.options[choice.selectedIndex].value;
         document.mult.src = imageDB[multColor].src;
         calcOhms();
      }
      function setTol(choice) {
         var tolColor = choice.options[choice.selectedIndex].value;
         document.tol.src = imageDB[tolColor].src;
         calcOhms();
      }
```

The rest of the script for the Head portion of the document merely provides the statements that open the secondary window to display the introductory document:

```
      function showIntro() {
         window.open("resintro.htm","","width=400,height=320,
            left=100,top=100");
      }
      </script>
   </head>
```

Creating the select objects

A comparatively lengthy part of the document is consumed with the HTML for the four `select` objects. Notice, however, that the document contains an `onload` event handler in the `<body>` tag. This handler calculates the results of the currently selected choices whenever the document loads or reloads. If it weren't for this event handler, you would not see the resistor art

when the document first appears. Also, because many browsers maintain their form controls' setting while the page is in history, a return to the page later must display the images previously selected in the form:

```
<body onload="calcOhms()">
   <center>
      <h1>Calculate <a href="javascript:showIntro()"
         onmouseover="status='An introduction to the resistor electronic
            component...';return true">
         Resistor</a> Values from Color Codes</h1>
      <form name="input">
         <select name="tensSelect" onchange="setTens(this)">
            <option value="Black" selected="selected">Black</option>
            <option value="Brown">Brown</option>
            <option value="Red">Red</option>
            <option value="Orange">Orange</option>
            <option value="Yellow">Yellow</option>
            <option value="Green">Green</option>
            <option value="Blue">Blue</option>
            <option value="Violet">Violet</option>
            <option value="Gray">Gray</option>
            <option value="White">White</option>
         </select> <select name="onesSelect" onchange="setOnes(this)">
            <option value="Black" selected="selected">Black</option>
            <option value="Brown">Brown</option>
            <option value="Red">Red</option>
            <option value="Orange">Orange</option>
            <option value="Yellow">Yellow</option>
            <option value="Green">Green</option>
            <option value="Blue">Blue</option>
            <option value="Violet">Violet</option>
            <option value="Gray">Gray</option>
            <option value="White">White</option>
         </select> <select name="multiplierSelect" onchange="setMult(this)">
            <option value="Black" selected="selected">Black</option>
            <option value="Brown">Brown</option>
            <option value="Red">Red</option>
            <option value="Orange">Orange</option>
            <option value="Yellow">Yellow</option>
            <option value="Green">Green</option>
            <option value="Blue">Blue</option>
            <option value="Violet">Violet</option>
            <option value="Gray">Gray</option>
            <option value="White">White</option>
            <option value="">Gold</option>
            <option value="">Silver</option>
         </select>     <select name="toleranceSelect"
            onchange="setTol(this)">
            <option value="Gold" selected="selected">Gold</option>
            <option value="Silver">Silver</option>
            <option value="None">None</option>
         </select>
      </form>
      <hr />
```

Drawing the initial images

The balance of the document, mostly in JavaScript, is devoted to creating the table and image objects whose `src` properties will be modified with each choice of a `select` object. The act of assembling the HTML for the image table occurs right after the `select` objects have rendered. References to those `select` elements are required in order to extract the currently selected values. If the `form` element that holds the `select` elements is not closed, you can't build a valid (and backward-compatible) reference to the `select` elements. Therefore, the page contains two forms: One for the `select` elements; one for the output text box inside the table:

```
        <script type="text/javascript">
        var form = document.forms["input"];
        var tensDigit = form.tensSelect.selectedIndex;
        var tensColor = form.tensSelect.options[tensDigit].value;
        var onesDigit = form.onesSelect.selectedIndex;
        var onesColor = form.onesSelect.options[onesDigit].value;
        var multDigit = form.multiplierSelect.selectedIndex;
        var multColor = form.multiplierSelect.options[multDigit].value;
        var tolDigit = form.toleranceSelect.selectedIndex;
        var tolColor = form.toleranceSelect.options[tolDigit].value;

        var table ="<table border='2'><form name='output'>";
        table += "<tr><th align='middle'>Resistance Value:<\/th>";
        table +=" <th align='middle'><input type='text' name='result' size=20
           onfocus='this.blur()'>";
        table +="<\/th><\/tr><tr><td colspan=2> ";
        table +="<img SRC='resleft.gif' width=127 height=182><img" +
           " src='" + tensColor +
           ".gif' name='tens' width=21 height=182><img" +
           " src='" + onesColor +
           ".gif' name='ones' width=21 height=182><img" +
           " src='" + multColor +
           ".gif' name='mult' width=21 height=182><img" +
           " src='spacer.gif' width=17 height=182><img" +
           " src='" + tolColor + ".gif' name='tol' width=21 height=182><img" +
           " src='resright.gif' width=127 height=182>";
        table += " <\/td><\/tr><\/form><\/table>";
        document.write(table);
        </script> <font size="2">Illustration Copyright 1996 Danny Goodman.
        All Rights Reserved.</font>
      </center>
   </body>
</html>
```

As you can see, the resistor images appear in one table cell (in the second row) that contains all seven image objects smashed against each other. To keep the images flush against each other, there can be no spaces or carriage returns between `<img>` tags.

Further Thoughts

I am very pleased with the improvements to performance and perceived quality that swappable images and image precaching bring to the current version of this calculator. Images change crisply. Network latency is no longer an issue.

In the layout department, however, annoying differences still exist among different platforms. At one point in the design process, I considered trying to align the pop-up menus with images of the resistor (or callout line images), but the differences in platform rendering of pop-up menus made that idea impractical. At best, I now separate the three left `select` objects from the right one by way of hard-coded spaces (` `).

You should notice from this exercise that I look for ways to blend JavaScript object data structures with my own data's structure. For example, the `select` objects serve multiple duties in these scripts. Not only does the text of each option point to an image file of the same name, but the index values of the same options are applied to the calculations. Things don't always work out that nicely, but whenever your scripts bring together user interface elements and data elements, look for algorithmic connections involving names or index integers that you can leverage to create elegant, concise code.

✦ ✦ ✦

CHAPTER 54

Application: Intelligent "Updated" Flags

In This Chapter

Temporary and persistent cookies

World time calculations

CGI-like intelligence

It happens to every active Web user all the time: You visit a site periodically and never know for sure what material is new since your last visit. Often, Web page authors may flag items with "New" or "Updated" `.gif` images after they update those items themselves. But if you fail to visit the site over a few modification sessions, the only items you find flagged are those that are new as of the most recent update by the page's author. Several new items from a few weeks back may be of vital interest to you, but you won't have the time to look through the whole site in search of material that is more recent than *your* last visit. Even if the items display their modification dates, do you remember for sure the date and time of your last visit to the page?

As much as I might expect a CGI program and database on a Web site to keep track of my last visit, that really is asking a great deal of the Web site. Besides, not every Web site has the wherewithal to build such a database system — if it can even put up its own CGIs. Plus, some users won't visit sites if they need to identify themselves or register.

After surveying the way scriptable browsers store cookie information and how time calculations are performed under popular Web browsers, I found that a feasible alternative is to build this functionality into HTML documents and let the scripting manage the feature for users. The goal is to save in the visitor's cookie file the date and time of the last visit to a page and then use that point as a measure against items that have an authorship time stamp in the HTML document.

The Cookie Conundrum

Managing the cookie situation in this application is a bit more complicated than you may think. The reason is that you have to take into account the possible ways visitors may come and go from your site while surfing the Web. You cannot use just one cookie to store the last time a user visits the site, because you cannot predict when you should update that information with today's date and time. For example, if you have a cookie with the previous visit in it, you eventually

need to store today's visit. But you cannot afford to overwrite the previous visit immediately (say, in `onload`) because your scripts need that information to compare against items on the page not only right now, but even after the visitor vectors off from a link and comes back later. That also means you cannot update that last visit cookie solely via an `onunload` event handler, because that, too, would overwrite information that you need when the visitor comes back a minute later.

To solve the problem, I devised a system of two cookies. One is written to the cookie that is given an expiration date of some time out in the future — the *hard cookie,* I call it. The other is a temporary cookie — the *soft cookie* — which stays in cookie memory but is never written to the file. Such temporary cookies are automatically erased as the browser quits.

The hard cookie stores the time stamp when a visitor first loads the page since the last launch of the browser. In other words, the hard cookie contains a time stamp of the current visit. Before the previous entry is overwritten, however, it is copied into the soft cookie. That soft cookie maintains the time stamp of the previous visit and becomes the measure against which author time stamps in the HTML document are compared. To guard against inadvertent overwriting of both cookies, a function triggered by the document's `onload` event handler looks to see if the soft cookie has any data in it. If so, the function knows that the visitor has been to this page in the current session and leaves the current settings alone. Thus, the visitor can come to the site and see what's new, vector off to some other location, and come back to see the same new items flagged and pick up from there.

One potential downside to this system is that if a user never quits the browser (or if the browser quits only by crashing), the cookies will never be updated. If you discover that a great deal of your users keep their computers and browsers running all the time, you can build in a kind of timeout that invalidates the soft cookie if the hard cookie is more than, say, 12 hours old.

Time's Not on Your Side

Thanks to more than 15 years' experience programming applications that involve tracking time, I am overly sensitive to the way computers and programming languages treat time on a global basis. This issue is a thorny one, what with the vagaries of Daylight Savings Time and time zones in some parts of the world that differ from their neighbors by increments other than whole hours.

To accomplish a time tracking scheme for this application, I had to be aware of two times: the local time of the visitor and the local time of the page author. Making times match up in what can be widely disparate time zones, I use one time zone — GMT — as the reference point.

When a visitor arrives at the page, the browser needs to save that moment in time so that it can be the comparison measure for the *next* visit. Fortunately, whenever you create a new date object in JavaScript, it does so internally as the GMT date and time. Even though the way you attempt to read the date and time created by JavaScript shows you the results in your computer's local time, the display is actually filtered through the time zone offset as directed by your computer's time control panel. In other words, the millisecond value of every date object you create is maintained in memory in its GMT form. That's fine for the visitor's cookie value.

For the page author, however, I was presented with a different problem. Rather than force the author to convert the time stamps throughout the document to GMT, I wanted to let the author enter dates and times in local time. Aside from the fact that many people have trouble doing time zone conversions, looking at an existing item in the HTML with a local time stamp and instantly recognizing when that was last updated is much easier.

The problem, then, is how to let the visitor's browser know what time the author's time stamp is in GMT terms. To solve the issue, the author's time stamp needs to include a reference to the author's time zone relative to GMT. An Internet convention provides a couple of ways to do this: specifying the number of hours and minutes difference from GMT or, where supported by the browser, the abbreviation of the time zone. In JavaScript, you can create a new date object out of one of the specially formatted strings containing the date, time, and time zone. Three examples follow for the Christmas Eve dinner that starts at 6 p.m. in the Eastern Standard Time zone of North America:

```
var myDate = new Date("24 Dec 1997 23:00:00 GMT");
var myDate = new Date("24 Dec 1997 18:00:00 GMT-0500");
var myDate = new Date("24 Dec 1997 18:00:00 EST");
```

The first assumes you know the Greenwich Mean Time for the date and time that you want to specify. But if you don't, you can use the GMT designation and offset value. The syntax indicates the date and time is in a time zone exactly five hours west of GMT (values to the east would be positive numbers) in `hhmm` format. Browsers also know all of the time zone abbreviations for North America (EST, EDT, CST, CDT, MST, MDT, PST, and PDT, where "S" is for standard time and "D" is for daylight time).

When a user visits a page with this application embedded in it, the visitor's browser converts the author's time stamp to GMT (with the help of the author's zone offset parameter), so that both the author time stamp and last visit time stamp (in the soft cookie) are comparing apples to apples.

The Application

All of this discussion may make the application sound complicated. That may be true, internally. But the goal, as in most of the samples in this book, is to make the application easy to use in your site, even if you're not sure how everything works inside.

The sample page described in this chapter and on the CD-ROM (`whatsnew.htm`) is pretty boring to look at, because the power all lies in the scripting that users don't see (see Figure 54-1). Though this figure may be the most uninspired graphic presentation of the book, the functionality may be the most valuable addition that you make to your Web site.

When you first open the document (do so from a copy on your hard disk so that you can modify the author time stamp in a moment), all you see are the two items on the page without any flags. Although both entries have author time stamps that pre-date the time you're viewing the page, a soft cookie does not yet exist against which to compare those times. But the act of making the first visit to the page has created a hard cookie of the date and time that you first opened the page.

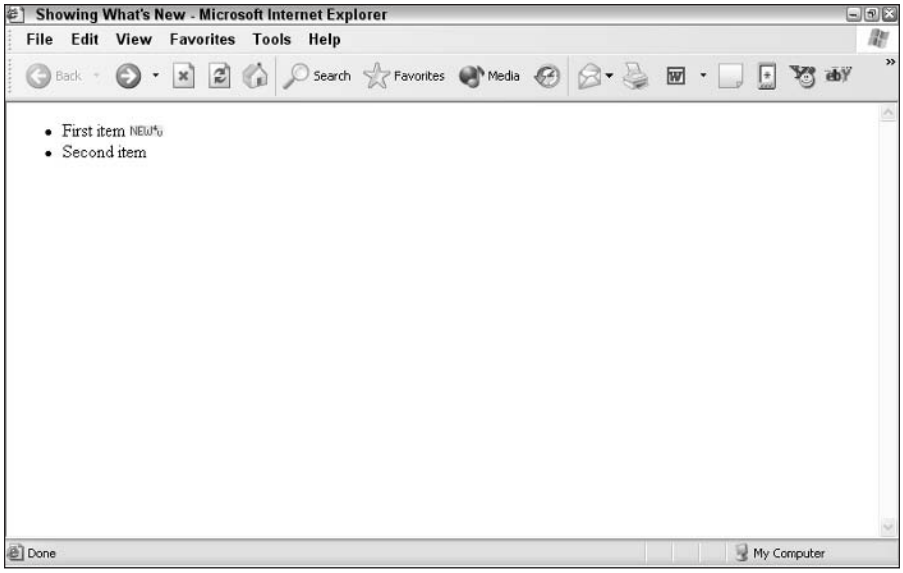


Figure 54-1: An item flagged as being new since my last visit to the page.

Quit the browser to get that hard cookie officially written to the cookie file. Then open the `whatsnew.htm` file in your script editor. Scroll to the bottom of the document, where you see the `<body>` tag and the interlaced scripts that time stamp anything that you want on the page. This application is designed to display a special `.gif` image that says "NEW 4U" whenever an item has been updated since your last visit.

Each interlaced script looks like this:

```
<script type="text/javascript">
document.write(newAsOf("12 Oct 2003 13:36:00 PDT"))
</script>
```

The `document.write()` method writes to the page whatever HTML comes back from the `newAsOf()` function. The parameter to the `newAsOf()` function is what holds the author time stamp and zone offset information. The time stamp value must be in the string format, as shown in the preceding example, with the date and time following the exact order ("`dd mmm yyyy hh:mm:ss`"). Month abbreviations are in English (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec).

As you see in the code that follows, the `newAsOf()` function returns an `<img>` tag with the "NEW 4U" image if the author time stamp (after appropriate conversion) is later than the soft cookie value. This image can be placed anywhere in a document. For example, at my Web site, I sometimes place the image before a contents listing rather than at the end. This means, too, that if part of your page is written by `document.write()` methods, you can just insert the `newAsOf()` function call as a parameter to your own `document.write()` calls.

If you want to see the author time stamping work, edit one of the time stamps in the `whatsnew.htm` file to reflect the current time. Save the document and relaunch the browser to view the page. The item whose author time stamp you modified should now show the bright "NEW 4U" image.

The Code

The sample page starts by initializing three global variables that are used in the statements that follow. One variable is a Boolean value indicating whether the visitor has been to the page before. Another variable, `lastVisit`, holds the value of the soft cookie whenever the visitor is at this page. One other variable, `dateAdjustment`, is (unfortunately) necessary to take into account a date bug that persists in Macintosh versions of Navigator (times of new date objects can be off by one hour). I use this variable to automatically handle any discrepancies:

```
<html>
  <head>
     <title>Showing What's New</title>
     <script type="text/javascript">
     // globals
     var repeatCustomer = false;
     var lastVisit = 0; // to hold date & time of previous access in GMT ms
     var dateAdjustment = 0; // to accommodate date bugs on some platforms
```

For reading and writing cookie data, I use virtually the same cookie functions from the outline table of contents (see Chapter 52). The only difference is that the cookie writing function includes an expiration date, because I want this cookie to hang around in the cookie file for a while — at least until the next visit, whenever that may be:

```
     // shared cookie functions
     var mycookie = document.cookie;
     // read cookie data
     function getCookieData(name) {
        var label = name + "=";
        var labelLen = label.length;
        var cLen = mycookie.length;
        var i = 0;
        while (i < cLen) {
           var j = i + labelLen;
           if (mycookie.substring(i,j) == label) {
              var cEnd = mycookie.indexOf(";",j);
              if (cEnd ==   -1) {
                 cEnd = mycookie.length;
              }
              return unescape(mycookie.substring(j,cEnd));
           }
           i++;
        }
        return "";
     }

     // write cookie data
     function setCookieData(name,dateData,expires) {
        mycookie = document.cookie = name + "=" + dateData + "; expires=" +
           expires;
     }
```

Notice that the `setCookieData()` function still maintains a level of reusability by requiring a name for the cookie to be passed as a parameter along with the data and expiration date. I could have hard-wired the name into this function, but that goes against my philosophy of designing for reusability.

Next comes a function that figures out if any problems with JavaScript date accuracy exist on any platform. Essentially, the function creates two date objects, one to serve as a baseline. Even the baseline date can be bad, so to test against it, you want to use the second object to create another date using the first date object's own values as a parameter. If any major discrepancies occur, they will show up loud and clear:

```
// set dateAdjustment to accommodate Mac bug in Navigator 3
function adjustDate() {
   var base = new Date();
   var testDate = base;
   testDate = testDate.toLocaleString();
   testDate = new Date(testDate);
   dateAdjustment = testDate.getTime() - base.getTime();
}
```

In truth, this function always shows some adjustment error, because both the baseline date and test date cannot be created simultaneously. Even in an accurate system, the two will vary by some small number of milliseconds. For the purposes here, this amount of variance is insignificant.

Setting the stage

Functions in the next part of the script get your cookies all in a row. The first function (`saveCurrentVisit()`) deals with the visitor's local time, converting it to a form that will be useful on the next visit. Although one of the local variables is called `nowGMT`, all the variable does is take the new date object and convert it to the GMT milliseconds value (minus any `dateAdjustment` value) by invoking the `getTime()` method of the date object. I use this name in the variable to help me remember what the value represents:

```
// write date of current visit (in GMT time) to cookie
function saveCurrentVisit() {
   var visitDate = new Date();
   var nowGMT = visitDate.getTime() - dateAdjustment;
   var expires = (nowGMT + (180 * 24 * 60 * 60 *1000));
   expires = new Date(expires);
   expires = expires.toGMTString();
   setCookieData("lastVisit", nowGMT, expires);
}
```

From the current time, I create an expiration date for the cookie. The example shows a date roughly six months (180 days, to be exact) from the current time. I leave the precise expiration date up to your conscience and how long you want the value to linger in a user's cookie file.

The final act of the `saveCurrentVisit()` function is to pass the relevant values to the function that actually writes the cookie data. I assign the name `lastVisit` to the cookie. If you want to manage this information for several different pages, assign a different cookie name for each page. This setup can be important in case a user gets to only part of your site during a visit. On the next visit, the code can point to page-specific newness of items.

The "temporary" cookie also needs to set an expiration date for itself. Normally, temporary cookies disappear when the user quits the browser. But these days, it's not uncommon for users to keep their computers and applications running for days (with fewer crashes than not too long ago). To force the browser to visit the site after awhile as if coming fresh (to look for new flags), the `nextPrevVisit` cookie is set to expire one hour after it is set:

```
// set cookie with next previous visit with one-hour expiration
function saveNextPrevVisit(lastStoredVisit) {
   var visitDate = new Date();
   var nowGMT = visitDate.getTime() - dateAdjustment;
   var expires = (nowGMT + (60*1000));
   expires = new Date(expires);
   expires = expires.toGMTString();
   setCookieData("nextPrevVisit", lastStoredVisit, expires);
}
```

The bulk of what happens in this application takes place in an initialization function. All the cookie swapping occurs there, as well as the setting of the `repeatCustomer` global variable value:

```
// set up global variables and establish whether user is repeat customer
function initialize() {
   var lastStoredVisit = getCookieData("lastVisit");
   var nextPrevStoredVisit = getCookieData("nextPrevVisit");

   adjustDate();

   if (!lastStoredVisit) { // never been here before
      saveCurrentVisit();
      repeatCustomer = false;
   } else { // been here before...
      if (!nextPrevStoredVisit) {  // but first time this session
         saveNextPrevVisit(lastStoredVisit); // expires in one hour
         lastVisit = parseFloat(lastStoredVisit);
         saveCurrentVisit();
         repeatCustomer = true;
      } else { // back again during this session (perhaps reload or Back)
         lastVisit = parseFloat(nextPrevStoredVisit);
         repeatCustomer = true;
      }
   }
}

initialize();
```

The first two statements retrieve both hard (`lastVisit`) and soft (`nextPrevVisit`) cookie values. After calling the function that sets any necessary date adjustment, the script starts examining the values of the cookies to find out where the visitor stands upon coming to the page.

The first test is whether the person has ever been to the page before. You do this by checking whether a hard cookie value (which would have been set in a previous visit) exists. If no such cookie value exists, the current visit time is written to the hard cookie and `repeatCustomer` is set to `false`. These actions prepare the visitor's cookie value for the *next* visit.

Should a user already be a repeat customer, you have to evaluate whether this visit is the user's first visit since launching the browser. You do that by checking for a value in the soft cookie. If that value doesn't exist, it means the user is here for the first time "today." You then grab the hard cookie and drop it into the soft cookie before writing today's visit to the hard cookie.

For repeat customers who have been here earlier in this session, you update the `lastVisit` global variable from the cookie value. The variable value will have been destroyed when the user left the page just a little while ago, whereas the soft cookie remains intact, enabling you to update the variable value now.

Outside of the function definition, the script automatically executes the `initialize()` function by that single statement. This function runs every time the page loads.

The date comparison

Every interlaced script in the body of the document calls the `newAsOf()` function to find out if the author's time stamp is after the last visit of the user to the page. This function is where the time zone differences between visitor and author must be neutralized so that a valid comparison can be made:

```
      function newAsOf(authorDate) {
         authorDate = new Date(authorDate);
         var itemUpdated = authorDate.getTime();
         alert(itemUpdated + ":" + lastVisit);
         return ((itemUpdated > lastVisit) && repeatCustomer) ?
            "<img src='updated.gif' height=10 width=30 \/>" : "";
      }
      </script>
   </head>
```

As you saw earlier, calls to this function require one parameter: a specially formatted date string that includes time zone information. The first task in the function is to re-cast the author's date string to a date object. You reuse the variable name (`authorDate`) because its meaning is quite clear. The date object created here is stored internally in the browser in GMT time, relative to the time zone data supplied in the parameter. To assist in the comparison against the `lastVisit` time (stored in milliseconds), the `getTime()` method converts `authorDate` to GMT milliseconds.

The last statement of the function is a conditional expression that returns the `<img>` tag definition for the "NEW 4U" image only if the author's time stamp is later than the soft cookie value and the visitor has been here before. Otherwise, the function returns an empty string. Any `document.write()` method that calls this function and executes via this branch writes an empty string — nothing — to the page.

A live <body>

For the sample document, I have you create a simple bulleted list of two entries, imaginatively called "First item" and "Second item." Interlaced into the HTML are scripts that are ready to insert the "NEW 4U" image if the author time stamp is new enough:

```
   <body>
      <ul>
         <li>First item <script type="text/javascript">
            document.write(newAsOf("19 Nov 2003 19:55:00 PST"))</script></li>
         <li>Second item <script type="text/javascript">
            document.write(newAsOf("18 Nov 2003 13:36:00 PST"))</script></li>
      </ul>
   </body>
</html>
```

All these script tags make the HTML a bit hard to read, but I believe the functionality is worth the effort.

Further Thoughts

You can, perhaps, go overboard with the way that you use this technique at a Web site. Like most features in JavaScript, I recommend using it in moderation and confining the flags to high-traffic areas that repeat visitors frequent. One hazard is that you can run out of the 20 cookies if you have too many page-specific listings.

You can share the same cookie among documents in related frames. Locate all the functions from the script in this chapter's Head section into a Head section of a framesetting document. Then, modify the call to the `newAsOf()` function by pointing it to the parent:

```
document.write(parent.newAsOf("18 Nov 2003 17:40:00 PDT"));
```

That way, one cookie can take care of all documents that you display in that frameset.

✦ ✦ ✦

CHAPTER 55

Application: Decision Helper

In This Chapter

Multiple frames

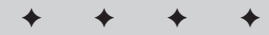
Multiple-document applications

Multiple windows

Persistent storage (cookies)

Scripted image maps

Scripted charts



The list of key concepts for this chapter's application looks like the grand finale to a fireworks show. As JavaScript implementations go, the application is, in some respects, over the top, yet not out of the question for presenting a practical interactive application on a Web site without any server programming.

The Application

I wanted to implement a classic application often called a *decision support system*. My experience with the math involved here goes back to the first days of Microsoft Excel. Rather than design a program that had limited appeal (covering only one possible decision tree), I set out to make a completely user-customizable decision helper. All the user has to do is enter values into fields on a series of screens; the program performs the calculations to let the user know how the various choices rank against each other.

Although I won't be delving too deeply into the math inside this application, you will find it helpful to understand how a user approaches this program and what the results look like. The basic scenario is a user who is trying to evaluate how well a selection of choices measure up to his or her expectations of performance. User input includes:

- The name of the decision
- The names of up to five alternatives (people, products, ideas, and so on)
- The factors or features of concern to the user
- The importance of each of the factors to the user
- A user ranking of the performance of every alternative in each factor

What makes this kind of application useful is that it forces the user to rate and weigh a number of often-conflicting factors. By assigning hard numbers to these elements, the user leaves the difficult process of figuring out the weights of various factors to the computer.

Results come in the form of floating-point numbers between 0 and 100. As an extra touch, I've added a graphical charting component to the results display.

The Design

With so much user input necessary for this application, conveying the illusion of simplicity was important. Rather than lump all text objects on a single scrolling page, I decided to break them into five pages, each consisting of its own HTML document. As an added benefit, I could embed information from early screens into the HTML of later screens, rather than having to create all changeable items out of text objects so that the application would work with older browsers. This "good idea" presented one opportunity and one rather large challenge.

The opportunity was to turn the interface for this application into something resembling a multimedia application using multiple frames. The largest frame would contain the forms the user fills out as well as the results page. Another frame would contain a navigation panel with arrows for moving forward and backward through the sequence of screens, plus buttons for going back to a home page and getting information about the program. I also thought a good idea would be to add a frame that provides instructions or suggestions for the users at each step. And so, the three-frame window was born, as shown in the first entry screen in Figure 55-1.

Using a navigation bar also enables me to demonstrate how to script a client-side image map — not an obvious task with JavaScript.

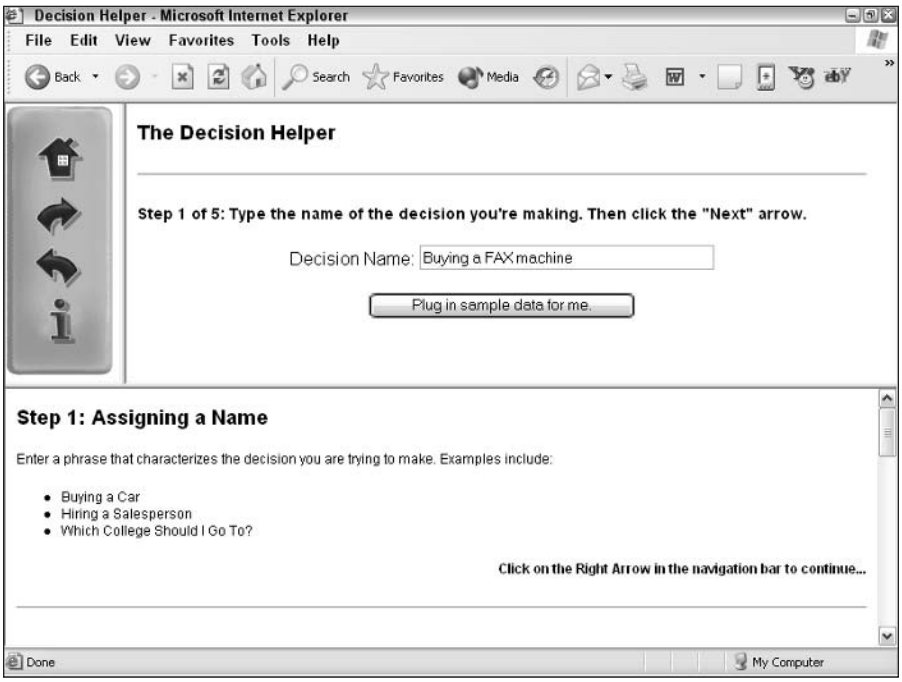


Figure 55-1: The Decision Helper window consists of three frames.

On the challenge side of this design, finding a way to maintain data globally as the user navigates from screen to screen was necessary. Every time one of the entry pages unloads, none of its text fields is available to a script. My first attack at this problem was to store the data as global variable data (mostly arrays) in the parent document that creates the frames. Because JavaScript enables you to reference any parent document's object, function, or variable (by preceding the reference with `parent`), I thought this task would be a snap. A nasty bug in Navigator 2 (the prominent browser when this application was first developed) got in the way at the time: If a document in any child window unloaded, the variables in the parent window got jumbled. The other hazard here is that a reload of the frameset could erase the current state of those variables.

My next hope was to use the `document.cookie` as the storage bin for the data. A major problem I faced was that this program needs to store a total of 41 individual data points, yet no more than 20 uniquely named cookies can be allotted to a given domain. But the cookie proved to be the primary solution for this application (although see the "Further Thoughts" section at the end of the chapter about a non-cookie version on the CD-ROM). For some of the data points (which are related in an array-like manner), I fashioned my own data structures so that one named cookie could contain up to five related data points. That reduced my cookie demands to 17. Note that the application with cookies works only from a Web server (not locally) with Safari 1.0.

The Files

Before I get into the code, let me explain the file structure of this application. Table 55-1 gives a rundown of the files used in the Decision Helper.

Table 55-1: Files Comprising the Decision Helper Application

File	*Description*
`index.htm`	Framesetting parent document
`dhNav.htm`	Navigation bar document which contains some scripting
`dhNav.gif`	Image displayed in `dhNav.htm`
`dh1.htm`	First Decision Helper entry page
`dh2.htm`	Second Decision Helper entry page
`dh3.htm`	Third Decision Helper entry page
`dh4.htm`	Fourth Decision Helper entry page
`dh5.htm`	Results page
`chart.gif`	Tiny image file used to create bar charts in `dh5.htm`
`dhHelp.htm`	Sample data and instructions document for lower-right frame
`dhAbout.htm`	Document that loads into a second window

A great deal of interdependence exists among these files. As you see later, assigning the names to some of these files was strategic for the implementation of the image map.

The Code

With so many JavaScript-enhanced HTML documents in this application, you can expect a great deal of code. To best grasp what's going on here, first try to understand the structure and interplay of the documents, especially the way the entry pages rely on functions defined in the parent document. My goal in describing this structure is not to teach you how to implement this application, but rather how to apply the lessons I learned while building this application to the more complex ideas that may be aching to get out of your head and into JavaScript.

index.htm

Taking a top-down journey through the JavaScript and HTML of the Decision Helper, start at the document that loads the frames. Unlike a typical framesetting document, however, this one contains JavaScript code in its Head section — code that many other documents rely on:

```
<html>
   <head>
      <title>Decision Helper</title>
```

An important consideration to remember is that in a multiple-frame environment, the title of the parent window's document is the name that appears in the window's title bar, no matter how many other documents are open inside its subframes.

The first items of the script control a global variable, `currTitle`, which is set by a number of the subsidiary files as the user navigates through the application. This variable ultimately helps the navigation bar buttons do their jobs correctly. Because this application relies on the `document.cookie` so heavily, the cookie management functions (slightly modified versions of Bill Dortch's Cookie Functions — see Chapter 18) are located in the parent document so they load only once. I simplified the cookie writing function because this application uses default settings for pathname and expiration. With no expiration date, the cookies don't survive the current browser session, which is perfect for this application:

```
      <script type="text/javascript">
      // global variable settings of current dh document number
      var currTitle = "";
      function setTitleVar(titleVal) {
         currTitle = "" + titleVal;
      }
      function getCookieVal(offset) {
         var endstr = document.cookie.indexOf (";", offset);
         if (("" + endstr) == "" || endstr == -1)
            endstr = document.cookie.length;
         return unescape(document.cookie.substring(offset, endstr));
      }

      function getCookie(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 getCookieVal (j);
            i = document.cookie.indexOf(" ", i) + 1;
            if (i == 0)
               break;
         }
```

```
   return null;
}

function setCookie(name, value) {
   document.cookie = name + "=" + escape (value) + ";";
}
```

When this application loads (or a user elects to start a new decision), it's important to grab the cookies you need and initialize them to basic values that the entry screens will use to fill entry fields when the user first visits them. All statements inside the `initializeCookies()` function call the `setCookie()` function, defined in the preceding listing. The parameters are the name of each cookie and the initial value — mostly empty strings. Before going on, study the cookie labeling structure carefully. I refer to it often in discussions of other documents in this application:

```
function initializeCookies() {
   setCookie("decName","");
   setCookie("alt0","");
   setCookie("alt1","");
   setCookie("alt2","");
   setCookie("alt3","");
   setCookie("alt4","");
   setCookie("factor0","");
   setCookie("factor1","");
   setCookie("factor2","");
   setCookie("factor3","");
   setCookie("factor4","");
   setCookie("import","0");
   setCookie("perf0","");
   setCookie("perf1","");
   setCookie("perf2","");
   setCookie("perf3","");
   setCookie("perf4","");
}
```

The following functions should look familiar to you. They were borrowed either wholesale or with minor modification from the data-entry validation section of the Social Security number database lookup in Chapter 50. I'm glad I wrote these as generic functions, making them easy to incorporate into this script. Because many of the entry fields on two screens must be integers between 1 and 100, I brought the data validation functions to the parent document rather than duplicating them in each of the subdocuments:

```
// JavaScript sees numbers with leading zeros as octal values, so strip
// zeros
function stripZeros(inputStr) {
   var result = inputStr;
   while (result.substring(0,1) == "0") {
      result = result.substring(1,result.length);
   }
   return result;
}

// general purpose function to see if a suspected numeric input
// is a positive integer
function isNumber(inputStr) {
   for (var i = 0; i < inputStr.length; i++) {
      var oneChar = inputStr.substring(i, i + 1);
      if (oneChar < "0" || oneChar > "9") {
         return false;
      }
```

```
      }
      return true;
   }

   // function to determine if value is in acceptable range for this
   // application
   function inRange(inputStr) {
      num = parseInt(inputStr);
      if (num < 1 || num > 100) {
         return false;
      }
      return true;
   }
```

To control the individual data-entry validation functions in the master controller, I again was able to borrow heavily from the application in Chapter 50:

```
   // Master value validator routine
   function isValid(inputStr) {
      if (inputStr != "" ) {
         inputStr = stripZeros(inputStr);
         if (!isNumber(inputStr)) {
            alert("Please make sure entries are numbers only.");
            return false;
         } else {
            if (!inRange(inputStr)) {
               alert("Entries must be numbers between 1 and 100.  Try
                  another value.");
               return false;
            }
         }
      }
      return true;
   }
```

Each of the documents containing entry forms retrieves and stores information in the cookie. Because all cookie functions are located in the parent document, it simplifies coding in the subordinate documents to have functions in the parent document acting as interfaces to the primary cookie functions. For each category of data stored as cookies, the parent document has a pair of functions for getting and setting data. The calling statements pass only the data to be stored when saving information; the interface functions handle the rest, such as storing or retrieving the cookie with the correct name.

In the following pair of functions, the decision name (from the first entry document) is passed back and forth between the cookie and the calling statements. Not only must the script store the data, but if the user returns to that screen later for any reason, the entry field must retrieve the previously entered data:

```
   function setDecisionName(str) {
      setCookie("decName",str);
   }
   function getDecisionName() {
      return getCookie("decName");
   }
```

The balance of the storage and retrieval pairs does the same thing for their specific cookies. Some cookies are named according to index values (`factor1`, `factor2`, and so on), so their cookie-accessing functions require parameters for determining which of the cookies to access, based on the request from the calling statement. Many of the cookie retrieval functions are called to fill in data in tables of later screens during the user's trip down the decision path:

```
      function setAlternative(i,str) {
         setCookie("alt" + i,str);
      }
      function getAlternative(i) {
         return getCookie("alt" + i);
      }
      function setFactor(i,str) {
         setCookie("factor" + i,str);
      }
      function getFactor(i) {
         return getCookie("factor" + i);
      }
      function setImportance(str) {
         setCookie("import",str);
      }
      function getImportance(i) {
         return getCookie("import");
      }
      function setPerformance(i,str) {
         setCookie("perf" + i,str);
      }
      function getPerformance(i) {
         return getCookie("perf" + i);
      }
```

One sequence of code that runs when the parent document loads is the one that looks to see if a cookie structure is set up. If no such structure is set up (the retrieval of a designated cookie returns a `null` value), the script initializes all cookies via the function described earlier:

```
      if (getDecisionName() == null) {
         initializeCookies();
      }
      </script>
   </head>
```

The balance of the parent document source code defines the frameset for the browser window. It establishes some hard-wired pixel sizes for the navigation panel. This ensures that the entire `.gif` file is visible whenever the frameset loads, without a ton of unwanted white space if the browser window is large:

```
   <frameset rows="250,*">
      <frameset cols="104,*">
         <frame name="navBar" src="dhNav.htm" scrolling="no" marginheight="2"
         marginwidth="1" />
         <frame name="entryForms" src="dh1.htm" />
      </frameset>
      <frameset rows="100%">
         <frame name="instructions" src="dhHelp.htm" />
      </frameset>
      <noframes>
         <body>
            <h1>It's really cool...</h1>
            <h2>...but only if your browser handles frames</h2>
            <hr />
            <a href="../index.html">Back</a>
         </body>
      </noframes>
   </frameset>
</html>
```

I learned an important lesson about scripting framesets along the way. Older browsers, especially NN through version 4, do not respond to changes in framesetting size attributes through a simple reload of the page. I found it necessary to reopen the frameset file from time to time. I also found it necessary to sometimes quit early Navigators altogether and relaunch to make some changes visible. Therefore, if you seem to be making changes, but reloading the frameset doesn't make the changes appear, try reopening or — as a last resort — quitting the browser.

dhNav.htm

Because of its crucial role in controlling the activity around this program, look into the navigation bar's document next. To accomplish the look and feel of a multimedia program, this document was designed as a client-side image map that has four regions scripted corresponding to the locations of the four buttons (see Figure 55-1). One function is connected to each button.

The first function is linked to the graphical Home button. For the listing here, I just present an alert dialog box replicating the action of navigating back to a real Web site's home page:

```
<html>
   <head>
      <title>Navigation Bar</title>
      <script type="text/javascript">
      function goHome() {
         alert("Navigate back to home page on a real site.");
      }
```

Each of the arrow navigation buttons brings the user to the next or previous entry screen in the sequence. To facilitate this without building tables of document titles and names, you call upon the `currTitle` global variable in the parent document. This value contains an integer in the range between 1 and 5, corresponding to the main content documents, `dh1.htm`, `dh2.htm`, and so on. As long as the offset number is no higher than the next-to-last document in the sequence, the `goNext()` function increments the index value by one and concatenates a new location for the frame. At the same time, the script advances the help document (in the bottom frame) to the anchor corresponding to the chosen entry screen by setting the `location.hash` property of that frame. Similar action navigates to the previous screen of the sequence through the `goPrev()` function. This time, the index value is decremented by one, and an alert warns the user if the current page is already the first in the sequence:

```
      function goNext() {
         var currOffset = parseInt(parent.currTitle);
         if (currOffset <= 4) {
            ++currOffset;
            parent.entryForms.location.href = "dh" + currOffset + ".htm";
            parent.instructions.location.hash = "help" + currOffset;
         } else {
            alert("This is the last form.");
         }
      }
      function goPrev() {
         var currOffset = parseInt(parent.currTitle);
         if (currOffset > 1) {
            --currOffset;
            parent.entryForms.location.href = "dh" + currOffset + ".htm";
            parent.instructions.location.hash = "help" + currOffset;
         } else {
            alert("This is the first form.");
         }
      }
```

Clicking the Info button displays a smaller window containing typical About-box data for the program (see Figure 55-2):

```
      function goInfo() {
         var newWindow = window.open("dhAbout.htm","","HEIGHT=250,WIDTH=380");
      }
      </script>
   </head>
```

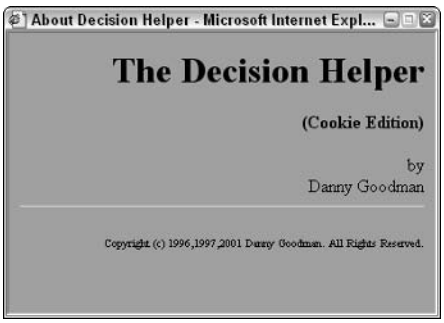


Figure 55-2: The About Decision Helper screen appears in a separate window.

The Body of the navigation document contains the part that enables you to script a client-side image map. Mouse `click` events weren't available to `area` elements until version 4 browsers, so to let these image maps work with older versions, mouse action is converted to script action by assigning a `javascript:` pseudo-URL to the `HREF` attribute for each `area` element. Instead of pointing to an entirely new URL (as `area` elements usually work), the attributes point to the JavaScript functions defined in the Head portion of this document. After a user clicks the rectangle specified by an `<area>` tag, the browser invokes the function instead.

```
<body bgcolor="white">
      <map id="navigation" name="navigation">
         <area shape="rect" coords="23,22,70,67" href="javascript:goHome()" />
         <area shape="rect" coords="25,80,66,116" href="javascript:goNext()" />
         <area shape="rect" coords="24,125,67,161"
         href="javascript:goPrev()" />
         <area shape="rect" coords="35,171,61,211"
         href="javascript:goInfo()" />
      </map> <img alt="image" src="dhNav.gif" height="240" width="96"
      border="0" usemap="#navigation" />
   </body>
</html>
```

Although not shown here, you can assign `onmouseover` event handlers to each `area` element for NN3+ and IE4+ to display a friendly message about the action of each button.

dh1.htm

Of the five documents that display in the main frame, `dh1.htm` is the simplest (refer to Figure 55-1). It contains a single entry field in which the user is invited to enter the name for the decision.

Only one function adorns the Head. This function summons one of the cookie interface functions in the parent window. A test is located here in case there is a problem with initializing the cookies. Rather than show `null` in the field, the conditional expression substitutes an empty string:

```
<html>
  <head>
     <title>DH1</title>
     <style type="text/css">
     body {font-family:Arial, Helvetica, serif}
     h2 {font-size:18px}
     h4 {font-size:14px}
     </style>
     <script type="text/javascript">
     function loadDecisionName() {
        var result = parent.getDecisionName();
        result = (result == null) ? "" : result;
        document.forms[0].decName.value = result;
     }
     </script>
  </head>
```

After the document loads, it performs three tasks (in the `onload` event handler). The first task is to set the global variable in the parent to let it know which number of the five main documents is currently loaded. Next, the script must fill the field with the decision name stored in the cookie. This task is important because users will want to come back to this screen to review what they entered previously. A third statement in the `onload` event handler sets the focus of the entire browser window to the one text object. This task is especially important in a multi-frame environment, such as this design. After a user clicks on the navigation panel, that frame has the focus. To begin typing into the field, the user has to tab (repeatedly) or click the text box to give the text box focus for typing. By setting the focus in the script when the document loads, you save the user time and aggravation:

```
  <body onload="parent.setTitleVar(1);
  loadDecisionName();document.forms[0].decName.focus()">
     <h2>The Decision Helper</h2>
     <hr />
     <h4>Step 1 of 5: Type the name of the decision you're making. Then click
        the "Next" arrow.</h4>
```

In the text field itself, an `onchange` event handler saves the value of the field in the parent's cookie for the decision name. No special Save button or other instruction is necessary here because any navigation that the user does via the navigation bar automatically causes the text field to lose focus and triggers the `onchange` event handler:

```
     <center>
        <form>
           Decision Name: <input type="text" name="decName" size="40"
           onchange="parent.setDecisionName(this.value)" />
        </form>
     </center>
  </body>
</html>
```

The copy of this file on the CD-ROM also has code that allows for plugging in sample data (as seen on my Web site) and a (commented out) textarea object that you can use for debugging cookie data.

dh2.htm

For the second data-entry screen (shown in Figure 55-3), five fields invite the user to enter descriptions of the alternatives under consideration. As with the decision name screen, the scripting for this page must both retrieve and save data displayed or entered in the fields.

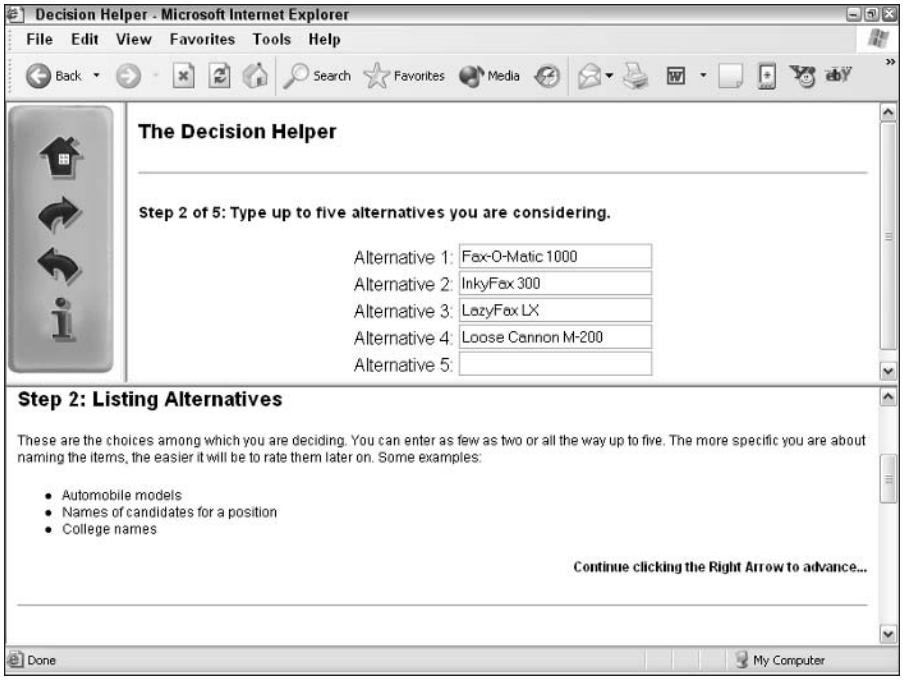


Figure 55-3: The second data-entry screen.

In one function, the script retrieves the alternative cookies (five total) and stuffs them into their respective text fields (as long as their values are not `null`). This function script uses a `for` loop to cycle through all five items — a common process throughout this application. Whenever a cookie is one of a set of five, the parent function has been written (in the following example) to store or extract a single cookie, based on the index value. Text objects holding like data (defined in the following listing) are all assigned the same name, so that JavaScript lets you treat them as array objects — greatly simplifying the placement of values into those fields inside a `for` loop:

```
<html>
  <head>
     <title>DH2</title>
     <style type="text/css">
     body {font-family:Arial, Helvetica, serif}
     h2 {font-size:18px}
     h4 {font-size:14px}
     </style>
     <script type="text/javascript">
     function loadAlternatives() {
        for (var i = 0; i < 5; i++) {
           var result = parent.getAlternative(i);
           result = (result == null) ? "" : result;
           document.forms[0].alternative[i].value = result;
        }
     }
     </script>
  </head>
```

After the document loads, the document number is sent to the parent's global variable, its fields are filled by the function defined in the Head, and the first field is handed the focus to assist the user in entering data the first time:

```
   <body onload="parent.setTitleVar(2);
   loadAlternatives();document.forms[0].alternative[0].focus()">
      <h2>The Decision Helper</h2>
      <hr />
      <h4>Step 2 of 5: Type up to five alternatives you are considering.</h4>
```

Any change that a user makes to a field is stored in the corresponding cookie. Each `onchange` event handler passes its indexed value (relative to all like-named fields) plus the value entered by the user as parameters to the parent's cookie-saving function:

```
      <center>
         <form>
            Alternative 1: <input type="text" name="alternative" size="25"
            onchange="parent.setAlternative(0,this.value)" /><br />
            Alternative 2: <input type="text" name="alternative" size="25"
            onchange="parent.setAlternative(1,this.value)" /><br />
            Alternative 3: <input type="text" name="alternative" size="25"
            onchange="parent.setAlternative(2,this.value)" /><br />
            Alternative 4: <input type="text" name="alternative" size="25"
            onchange="parent.setAlternative(3,this.value)" /><br />
            Alternative 5: <input type="text" name="alternative" size="25"
            onchange="parent.setAlternative(4,this.value)" /><br />
         </form>
      </center>
   </body>
</html>
```

dh3.htm

With the third screen, the complexity increases a bit. Two factors contribute to this increase in difficulty. One is that the limitation on the number of cookies available for a single domain forces you to join into one cookie the data that might normally be distributed among five cookies. Second, with the number of text objects on the page (see Figure 55-4), it becomes more efficient (from the standpoint of tedious HTML writing) to let JavaScript deploy the fields. The fact that two sets of five related fields exist facilitates using `for` loops to lay out and populate them.

One initial function here is reminiscent of Head functions in previous entry screens. This function retrieves a single factor cookie from the set of five cookies:

```
<html>
  <head>
     <title>DH3</title>
     <style type="text/css">
        body {font-family:Arial, Helvetica, serif}
        h2 {font-size:18px}
        h4 {font-size:14px}
     </style>
     <script type="text/javascript">
     function getdh3Factor(i) {
        var result = parent.getFactor(i);
        if (result == null) {
        return "";
        }
        return result;
     }
```

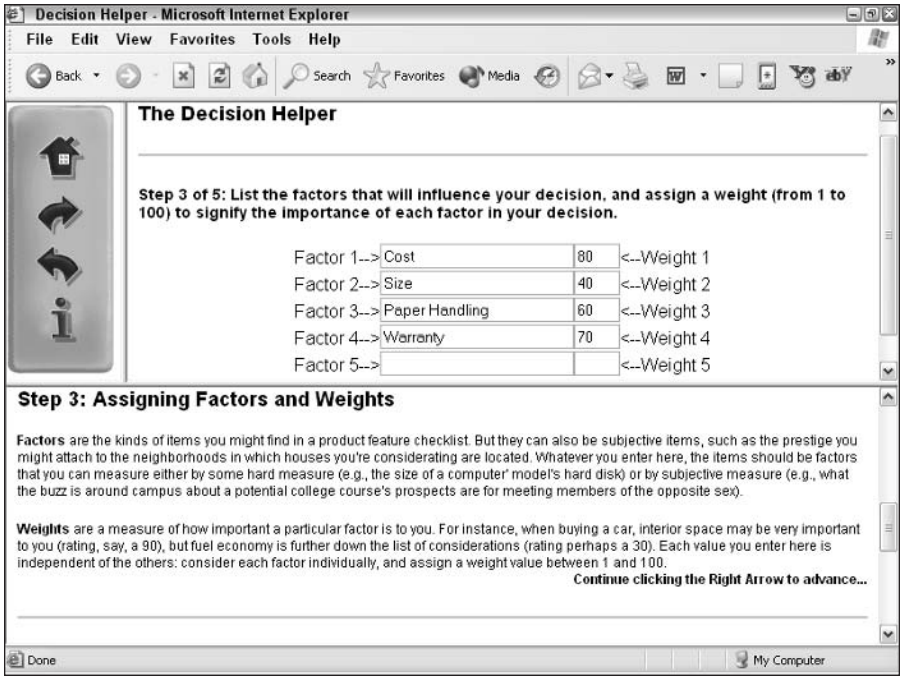


Figure 55-4: Screen for entering decision factors and their weights.

Values for the five possible weight entries are stored together in a single cookie. To make this work, I had to determine a data structure for the five "fields" of a single cookie "record." Because all entries are integers, I can choose any nonnumeric character. I arbitrarily selected the period:

```
function setdh3Importance() {
   var oneRecord = "";
   for (var i = 0; i < 5; i++) {
      var dataPoint = document.forms[0].importance[i].value;
      if (!parent.isValid(dataPoint)) {
         document.forms[0].importance[i].focus();
         document.forms[0].importance[i].select();
         return;
      }
      oneRecord += dataPoint + ".";
   }
   parent.setImportance(oneRecord);
   return;
}
```

The purpose of the `setdh3Importance()` function is to assemble all five values from the five Weight entry fields (named "importance") into a period-delimited record that is ultimately sent to the cookie for safekeeping. Another of the many `for` loops in this application cycles through each of the fields, checking for validity and then appending the value with its trailing period to the variable (`oneRecord`) that holds the accumulated data. As soon as the loop finishes, the entire record is sent to the parent function for storage.

Although the function shows two `return` statements, the calling statement does not truly expect any values to be returned. Instead, I use the `return` statement inside the `for` loop as a way to break out of the `for` loop without any further execution whenever an invalid entry is found. Just prior to that, the script sets the focus and select to the field containing the invalid entry. JavaScript, however, is sensitive to the fact that a function with a `return` statement in one possible outcome doesn't have a `return` statement for other outcomes (an error message to this effect appears in some browsers if you try the function without balanced returns). By putting a `return` statement at the end of the function, all other possibilities are covered to the browser's satisfaction.

The inverse of storing the weight entries is retrieving them. Because the `parent.getImportance()` function returns the entire period-delimited record, this function must break apart the pieces and distribute them into their corresponding Weight fields. A combination of string methods determines the offset of the period and how far the data extraction should go into the complete record. Before the `for` loop repeats each time, it is shortened by one "field's" data. In other words, as the `for` loop executes, the copy of the cookie data returned to this function is pared down one entry at a time as each entry is stuffed into its text object for display:

```
      function getdh3Importance() {
         var oneRecord = parent.getImportance();
         if (oneRecord != null) {
            for (var i = 0; i < 5; i++) {
               var recLen = oneRecord.length;
               var offset = oneRecord.indexOf(".");
               var dataPoint = (offset >= 0 ) ?
                  oneRecord.substring(0,offset) : "";
               document.forms[0].importance[i].value = dataPoint;
               oneRecord = oneRecord.substring(offset+1,recLen);
            }
         }
      }
      </script>
   </head>
```

Upon loading the document, the only tasks that the `onload` event handler needs to do are to update the parent global variable about the document number and to set the focus to the first entry field of the form:

```
   <body onload="parent.setTitleVar(3);document.forms[0].factor[0].focus()">
      <h2>The Decision Helper</h2>
      <hr />
      <h4>Step 3 of 5: List the factors that will influence your decision, and
         assign a weight (from 1 to 100) to signify the importance of each
         factor in your decision.</h4>
```

Assembling the HTML for the form and its 10 data-entry fields needs only a few lines of JavaScript code. Performed inside a `for` loop, the script assembles each line of the form, which consists of a label for the Factor (and its number), the factor input field, the importance input field, and the label for the Weight (and its number). A `document.write()` method writes each line to the document:

```
      <p>
         <script type="text/javascript">
         var output = "<center><form>";
         for (i = 0; i < 5; i++) {
            output += "Factor " + (i+1) + "--><input type='text' name='factor'
               size='25' ";
```

```
            var eHandler = " onchange=\'parent.setFactor(" + i +
               ",this.value)\'";
            output += eHandler + "value='" + getdh3Factor(i) + "'>";

            output += "<input type='text' name='importance' size='3' ";
            var eHandler = " onchange=\'setdh3Importance()\'";
            output += eHandler + "value=''>";

            output += "<--Weight " + (i+1) + "<br \/>";
            document.write(output);
            output = "";
         }
         document.write("<\/form><\/center>");
         getdh3Importance();
         </script>
      </p>
   </body>
</html>
```

Each of the scripted text objects has an event handler. Notice that each event handler is first defined as a variable on a statement line just above its insertion into the string being assembled for the `input` object definition. One reason for this fact is that the nested quote situation gets quite complex when you are doing these tasks all in one massive assignment statement. Rather than mess with matching several pairs of deeply nested quotes, I found it easier to break out one portion (the event handler definition) as a variable value and then insert that preformatted expression into the concatenated string for the `input` definition.

Notice, too, how the different ways of storing the data in the cookies influence the ways the existing cookie data is filled into the fields as the page draws itself. For the factors, which have one cookie per factor, the `value` attribute of the field is set with a specific indexed call to the parent factor cookie retriever, one at a time. But for the importance values, which are stored together in the period-delimited chunk, a separate function call (`getdh3Importance()`) executes after the fields are already drawn (with initial values of empty strings) and fills all the fields in a batch operation.

dh4.htm

Step 4 of the decision process (shown in Figure 55-5) is the most complex step because of the sheer number of entry fields: 25 in all. Notice that this screen retrieves data from two of the previous screens (or rather from the cookies preserving the entries) and embeds the values into the fixed parts of the table. All these tasks are possible when you create those tables with JavaScript.

Functions for getting and setting performance data are complex because of the way I was forced to combine data into five "field" records. In other words, one parent cookie exists for each row of data cells in the table. To extract cell data for storage in the cookie, I use nested `for` loop constructions. The outer loop counts the rows of the table, whereas the inner loop (with the `j` counter variable) works its way across the columns for each row.

Because all cells are named identically, they are indexed with values from 0 to 24. Calculating the row (`i * 5`) plus the column number establishes the cell index value. After you check for validity, each cell's value is added to the row's accumulated data. Each row is then saved to its corresponding cookie. As in the code for `dh3.htm`, the `return` statement is used as a way to break out of the function if an entry is deemed invalid.

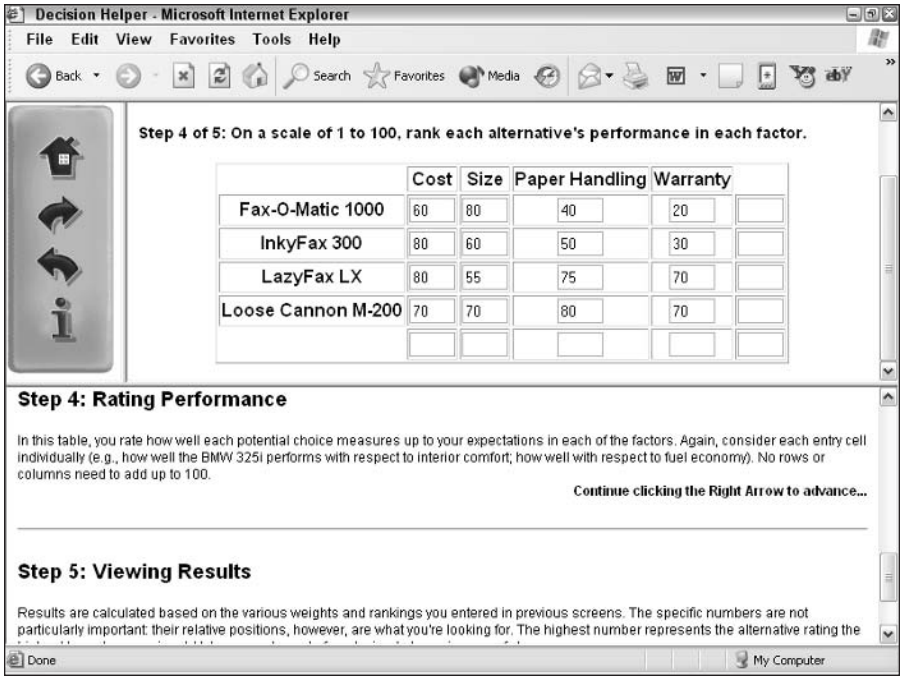


Figure 55-5: A massive table includes label data from earlier screen entries.

Retrieving the data and populating the cells for the entire table requires an examination of each of the five performance cookies, and for each labeled cookie's data, a parsing for each period-delimited entry. After a given data point is in hand (one entry for a cell), it must go into the cell with the proper index:

```
<html>
  <head>
     <title>DH4</title>
     <style type="text/css">
     body {font-family:Arial, Helvetica, serif}
     h2 {font-size:18px}
     h4 {font-size:14px}
     </style>
     <script type="text/javascript">
     function getdh4Performance() {
        var oneRecord = "";
        var recLen = 0;
        var offset = 0;
        var dataPoint = "";
        var cellNum = 0;
        for (var i = 0; i < 5; i++) {
           oneRecord = parent.getPerformance(i);
           if (oneRecord == null) {
              continue;
           }
```

```
            for (var j = 0; j < 5; j++) {
               recLen = oneRecord.length;
               offset = oneRecord.indexOf(".");
               dataPoint = oneRecord.substring(0,offset);
               cellNum = j + (i * 5);
               document.forms[0].ranking[cellNum].value = dataPoint;
               oneRecord = oneRecord.substring(offset+1,recLen);
            }
         }
      }
      function setdh4Performance() {
         var oneRecord = "";
         var cellNum = 0;
         var dataPoint = "";
         for (var i = 0; i < 5; i++) {
            oneRecord = "";
            for (var j = 0; j < 5; j++) {
               cellNum = j + (i * 5);
               dataPoint = document.forms[0].ranking[cellNum].value;
               if (!parent.isValid(dataPoint)) {
                  document.forms[0].ranking[cellNum].focus();
                  document.forms[0].ranking[cellNum].select();
                  return;
               }
               oneRecord += dataPoint + ".";
            }
            parent.setPerformance(i,oneRecord);
         }
         return;
      }
      </script>
   </head>
```

After the document is loaded, the `onload` event handler sends the document number to the parent global variable and brings focus to the first field of the table:

```
   <body onload="parent.setTitleVar(4);document.forms[0].ranking[0].focus()">
      <h2>The Decision Helper</h2>
      <hr />
      <h4>Step 4 of 5: On a scale of 1 to 100, rank each alternative's
         performance in each factor.</h4>
```

To lessen the repetitive HTML for all tables, JavaScript again assembles and writes the HTML that defines the tables. In the first batch, the script uses yet another `for` loop to retrieve the factor entries from the parent cookie so that the words can be embedded into `<th>` tags of the first row of the table. If every factor field is not filled in, the table cell is set to empty:

```
      <p>
         <script type="text/javascript">
         var output = "<div align='center'><form name='perfRankings'><table
            border='1'>";
         output += "<tr><td><\/td>";
         for (var i = 0; i < 5; i++) {
            var oneFactor = parent.getFactor(i);
            oneFactor = (oneFactor == null) ? "" : oneFactor;
            output += "<th>" + oneFactor + "<\/th>";
         }
```

Next comes the assembly of subsequent rows of the table. The first column displays the name of each alternative (within `<th>` tags). The remaining columns are text boxes, all with the same name and event handler. As each row of table definition is completed, it is written to the document. After the table and form closing tags are also written, the `getdh4Performance()` function retrieves all cookie data for the fields and distributes it accordingly:

```
         for (var i = 0; i < 5; i++) {
            var oneAlt = parent.getAlternative(i);
            oneAlt = (oneAlt == null) ? "" : oneAlt;
            output += "<tr><th>" + oneAlt + "<\/th>";
            for (var j = 0; j < 5; j++) {
               output += "<td align='center'><input type='text' size='3'
                  name='ranking' value=''
                  onchange='setdh4Performance()'><\/td>";
            }
            output += "<\/tr>";
            document.write(output);
            output = "";
         }
         document.write("<\/table><\/form><\/div>");
         getdh4Performance();
         </script>
      </p>
   </body>
</html>
```

dh5.htm

From a math standpoint, dh5.htm's JavaScript gets pretty complicated. But because the complexity is attributed to the decision support calculations that turn the user's entries into results, I treat the calculation script shown here as a black box. You're free to examine the details, if you're so inclined.

Results appear in the form of a table (see Figure 55-6) with columns showing the numeric results and an optional graphical chart.

For the purposes of this example, you only need to know a couple of things about the `calculate()` function. First, this function calls all the numeric data stored in parent cookies to fulfill values in its formulas. Second, results are tabulated and placed into a five-entry indexed array called `itemTotal[i]`. This array is defined as a global variable, so that its contents are available to scripts coming up in the Body portion of the document:

```
<html>
   <head>
      <title>DH5</title>

      <style type="text/css">
      body {font-family:Arial, Helvetica, serif}
      h2 {font-size:18px}
      h4 {font-size:14px}
      </style>
      <script type="text/javascript">
      var itemTotal = new Array();

      function calculate() {
         var scratchpad = "";
         var importanceSum = 0;
         var oneRecord = parent.getImportance();
         var weight = new Array();
```

```
      for (i = 0; i < 5; i++) {
         var recLen = oneRecord.length;
         var offset = oneRecord.indexOf(".");
         scratchpad = oneRecord.substring(0,offset);
         importanceSum += (scratchpad == "" || scratchpad == "NaN") ?
            0 : parseInt(scratchpad);
         oneRecord = oneRecord.substring(offset+1,recLen);
      }
      oneRecord = parent.getImportance();
      for (i = 0; i < 5; i++) {
         recLen = oneRecord.length;
         offset = oneRecord.indexOf(".");
         scratchpad = oneRecord.substring(0,offset);
         weight[i] = (scratchpad == "" || scratchpad == "NaN") ?
            0 : parseInt(scratchpad)/importanceSum * 100;
         oneRecord = oneRecord.substring(offset+1,recLen);
      }
      for (i = 0; i < 5; i++) {
         oneRecord = parent.getPerformance(i);
         if (oneRecord == null) {
            continue;
         }
         scratchpad = 0;
         for (var j = 0; j < 5; j++) {
            var recLen = oneRecord.length;
            var offset = oneRecord.indexOf(".");
            var dataPoint = oneRecord.substring(0,offset);
            scratchpad += (dataPoint != "" || dataPoint == "NaN") ?
               parseInt(dataPoint) * weight[j] / 100 : 0;
            oneRecord = oneRecord.substring(offset+1,recLen);
         }
         itemTotal[i] = scratchpad;
      }
   }
   calculate();
   </script>
</head>
```

Constructing this function served up many reminders about keeping data types straight. Because the data stored in cookies was in the form of strings, when it comes time to do some real math with those values, careful placement of the `parseInt()` function is essential for getting the math operators to work.

An `onload` event handler sends the document number to the global variable, as usual. The results display in this document relies heavily on stored and calculated values, so the table is constructed entirely out of JavaScript. That also means it can redisplay the decision name as part of the page:

```
<body onload="parent.setTitleVar(5)">
   <h2>The Decision Helper</h2>
   <hr />
   <script type="text/javascript">
   document.write("<h4>" + parent.getDecisionName() + "<\/h4><br \/>
      <br \/>");
   var output = "<div align='center'><form name='Results'><table
      border='1'>";
   output += "<tr><td><\/td><th>Results<\/th><th>Ranking<\/th>";
   document.write(output);
   output = "";
```

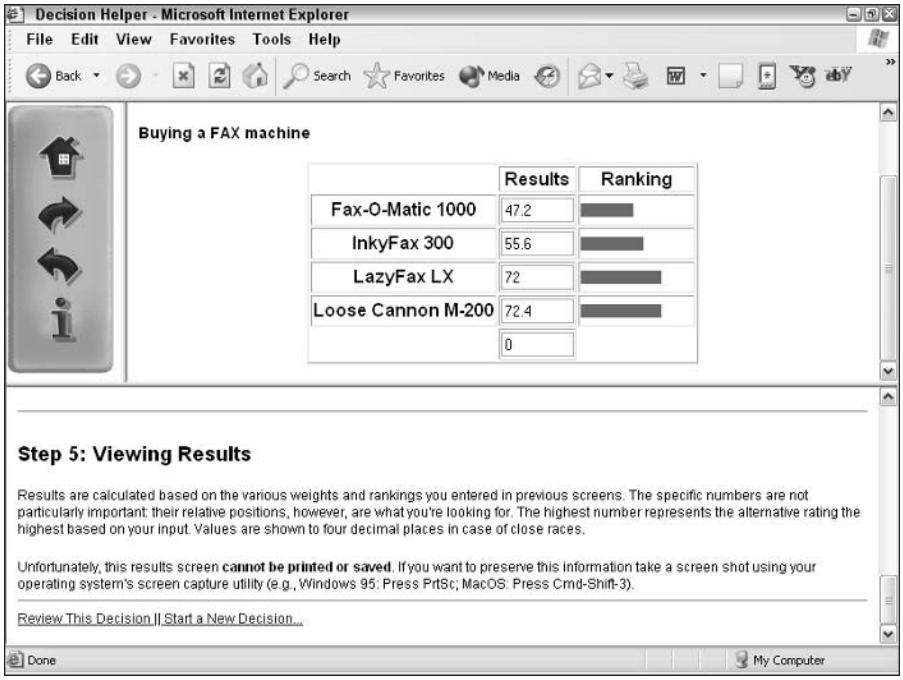


Figure 55-6: The results screen for a decision.

I need to break up the discussion of the `for` loop that produces the results because there are two distinct parts of this HTML assembly. The first, shown in the following script segment, assembles the first two cells of each row of the table. The first cell contains an embedded listing of the alternative name (in `<th>` tags). To highlight the calculated values — and enable the `size` attribute to do the artificial job of truncating the floating-point number — the results are shown in text boxes. For each row, the corresponding result in `itemTotal[i]` is inserted as the `value` attribute of the text box. The `size` attribute is set to 7, which allows the typical double-digit results, a decimal point, and four digits to the right of the decimal (an extra pixel shows on the Macintosh version, however):

```
for (var i = 0; i < 5; i++) {
   var oneAlt = parent.getAlternative(i);
   oneAlt = (oneAlt == null || oneAlt == "") ? "" : oneAlt;
   itemTotal[i] = (oneAlt == "") ? 0 : itemTotal[i];
   output += "<tr><th>" + oneAlt + "<\/th>";
   output += "<td align='center'><input type='text' size='7'
      name='ranking' value=" + itemTotal[i] + "><\/td>";
```

For extra pizzazz, a third column "draws" a bar chart within a 100-pixel-wide cell. The bars are actually scalings of a one-pixel-wide `.gif` file (an orange line, 12 pixels tall). A single-color `.gif` image scales to fill whatever width is assigned in the `width` attribute. This method is faster and far better than a more tedious method (tedious from the Web page author's point

of view) of creating 100 different .gif files, one for each possible width of the bar. I also could have used a one-pixel square .gif file with equal ease:

```
         output += "<td width='100'>";
         chartWidth = Math.round(itemTotal[i]);
         if (chartWidth > 0) {
            output += "<img src='chart.gif' height='12' width='" + chartWidth +
               "' \/>";
         }
         output += "<\/td><\/tr>";
         document.write(output);
         output = "";
      }
      document.write("<\/table><\/form><\/div>");
      </script>
   </body>
</html>
```

dhHelp.htm

The only other code worth noting in this application is in the dhHelp.htm document, which appears in the lower-right frame of the window. At the end of this document are two links that call separate JavaScript functions in this document's Head section. The Head functions are as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
     <title>Decision Helper Help</title>
     <style type="text/css">
     body {font-family:Arial, Helvetica, serif; font-size:12px}
     h4 {font-size:18px}
     div {align:right}
     </style>
     <script type="text/javascript">
     function goFirst() {
        parent.entryForms.location = "dh1.htm";
        self.location.hash = "help1";
     }
     function restart() {
        if (confirm("Erase current decision and start a new one?")) {
           parent.initializeCookies();
           parent.entryForms.location = "dh1.htm";
           self.location.hash = "help1";
        }
     }
     </script>
  </head>
```

One function merely returns the user to the beginning of the sequences for both the entry screens and the help screen. The second function is a rare instance in which a confirm dialog box makes sense: It is about to erase all entered data. If the user says it's okay to go ahead, the parent window's function for initializing all cookies is called, and the navigation for both the entry and help screens goes back to the beginning.

The links at the bottom of the document (see Figure 55-6) are coded to trigger JavaScript functions (rather than navigate to URLs) and include `onmouseover` event handlers to provide more information about the link in the status bar:

```
<a href="javascript:goFirst()"
onmouseover="window.status='Go back to beginning to review
data...';return true">
Review This Decision</a> || <a href="javascript:restart()"
onmouseover="window.status='Erase current data and start over...';
return true">Start a New Decision...</a>
```

Further Thoughts

If you've managed to follow through with this application's discussions, you will agree that it's quite a JavaScript workout. But this application proves that, without a ton of code, JavaScript provides enough functionality to add a great deal of interactivity and pseudo-intelligence to an otherwise flat HTML document.

As an alternative to using cookies for data storage, I have also implemented a version of the application that uses text boxes defined in a frame defined with a row height of 0. This technique further challenges the synchronization of frames during reloading when a user resizes the browser window or navigates with the Back or Forward browser buttons. This alternate version is located on the CD-ROM for your own investigation and comparison.

Dynamic HTML also offers some possibilities for this application. The entire program can be presented in a no-frame window, with the navigation, interactive content, and instructions frames incorporated into individual positionable objects. The interactive content area can be treated almost like a slide show, with successive pages flying in from one edge.

Not only is this application instructive for many JavaScript techniques, but it is also fun to play with as a user. Some financial Web sites have adapted it to assist visitors with investment decisions. You can use it to dream about where to go on a dream vacation, or help you decide the most ethical of a few paths confronting you in a personal dilemma. There's something about putting in data, turning a crank, and watching results (with a bar chart to boot!) magically appear on the screen.

✦ ✦ ✦

CHAPTER 56

Application: Cross-Browser DHTML Map Puzzle

In This Chapter

Applying a DHTML API

Scripting, dragging, and layering of multiple elements

Event handling for three DOMs at once

Dynamic HTML allows scripts to position, overlap, and hide or show elements under the control of stylesheets and scripting. To demonstrate modern cross-browser DHTML development techniques, this chapter describes the details of a jigsaw puzzle game using pieces of a map of the "lower 48" United States (I think everyone would guess where Alaska and Hawaii go on a larger map of North America). I chose this application because it allows me to demonstrate several typical tasks you might want to script in DHTML: hiding and showing elements; handling events for multiple elements; tracking the position of an element with the mouse cursor; absolute positioning of elements; changing the z-order of elements; changing element colors; and animating movement of elements.

As with virtually any programming task, the example code here is not laid out as the quintessential way to accomplish a particular task. Each author brings his or her own scripting style, experience, and implementation ideas to a design. Very often, you have available several ways to accomplish the same end. If you find other strategies or tactics for the operations performed in these examples, it means you are gaining a good grasp of both JavaScript and Dynamic HTML.

The Puzzle Design

Figure 56-1 shows the finished map puzzle with the game in progress. To keep the code to a reasonable length, the example provides positionable state maps for only seven western states. Also, the overall design is intentionally Spartan so as to place more emphasis on the positionable elements and their scripting, rather than on fancy design.

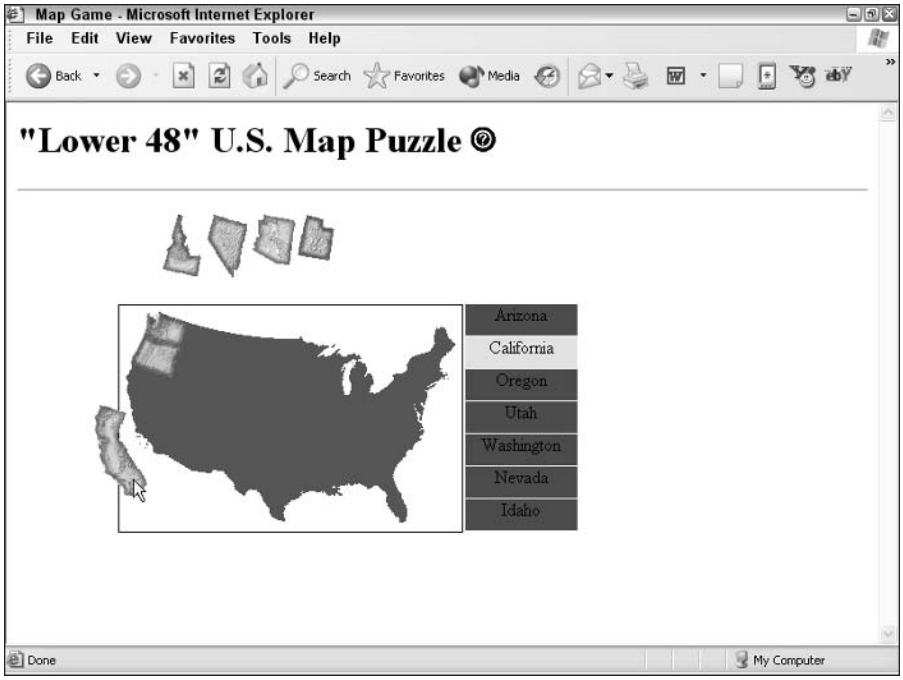


Figure 56-1: The puzzle map game DHTML example.
(Images courtesy Map Resources — www.mapresources.com)

When the page initially loads, all the state maps are presented across the top of the puzzle area. The state labels all have a red background, and the silhouette of the continental United States has no features in it. To the right of the title is a question mark icon. A click of this icon causes a panel of instructions to glide to the center of the screen from the right edge of the browser window. That panel has a button that hides the panel.

To play the game (no scoring or time keeping is in this simplified version), a user clicks and drags a state with the goal of moving it into its rightful position on the silhouette. While the user drags the state, its label background to the right of the main map turns yellow to highlight the name of the state being worked on. To release the state in its trial position, the user releases the mouse button. If the state is within a 4-pixel square region around its true location, the state snaps into its correct position and the corresponding label background color turns green. If the state is not dropped close enough to its destination, the label background reverts to red, meaning that the state still needs to be placed.

After the last state map is dropped into its proper place, all the label backgrounds will be green, and a congratulatory message is displayed where the state map pieces originally lay. Should a user then pick up a state and drop it out of position, the congratulatory message disappears.

I had hoped that all versions of the application would look the same on all platforms. They do, with one small exception. Because the labels are generated as positioned `div` elements for all browsers, NN4 (especially on the Windows version) doesn't do as good a rendering job as other browsers. If I were to use genuine `layer` elements for the labels just for NN4, they'd

look better. And, while the code could use scripts to generate `layer`s for NN4 and `div`s for others, the choice here was to stay with `div` elements alone. If you try this game on NN4 and other DHTML browsers, you will see minor differences in the way the labels are colored (red, yellow, and green) during game play. All other rendering and behavior is identical (although a rendering bug in NN6 is discussed later).

Implementation Details

Due to the number of different scripted properties being changed in this application, I decided to implement a lot of the cross-platform scripting as a custom API loaded from an external `.js` file library. The library, whose code is dissected and explained in Chapter 47, contains functions for most of the scriptable items you can access in DHTML. Having these functions available simplified what would have been more complex functions in the main part of the application.

Although I frown on using global variables except where absolutely necessary, I needed to assign a few globals for this application. All of them store information about the state map currently picked up by the user and the associated label. This information needs to survive the invocations of many functions between the time the state is picked up until it is dropped and checked against the "database" of state data.

That database is another global object — a global that I don't mind using at all. Constructed as a multidimensional array, each "record" in the database stores several fields about the state, including its destination coordinates inside the outline map and a Boolean field to store whether the state has been correctly placed in position.

Out of necessity for NN4, the state map images are encased in individual `div` elements. This makes their positionable characteristics more stable, as well as making it possible to capture mouse events that NN4's image objects do not recognize. If the application were being done only for IE4+ and W3C DOMs, the images themselves could be positionable, and the DHTML API could be used without modification. You could certainly argue that it's time to let go of version 4 browsers and move forward, but to maximize compatibility across the broadest possible user base, supporting version 4 browsers whenever possible still isn't a bad idea.

The custom API

To begin the analysis of the code, you should be familiar with the API that is linked in from an external `.js` library file. Listing 47-2 contains that code and its description.

The main program

Code for the main program is shown in Listing 56-1. The listing is a long document, so I interlace commentary throughout the listing. Before diving into the code, however, allow me to present a preview of the structure of the document. With two exceptions (the map silhouette and the help panel), all positionable elements have their styles set via stylesheets in the `head` of the document. Notice the way class and id selectors are used to minimize the repetitive nature of the styles across so many similar items. After the stylesheets come the scripts for the page. All of this material is inside the `<head>` tag section. I leave the `<body>` section to contain the visible content of the page. This approach is an organization style that works well for me, but you can adopt any style you like, provided various elements that support others on the page are loaded before the dependent items (for example, define a style before assigning its name to the corresponding content tag's ID attributes).

Listing 56-1: The Main Program (mapgame.htm)

```
<html>
   <head>
      <title>Map Game</title>
```

Most of the positionable elements have their CSS properties established in the `<style>` tag at the top of the document. Positionable elements whose styles are defined here include a text label for each state, a map for each state, and a congratulatory message. Notice that the names of the label and state map objects begin with a two-letter abbreviation of the state. This labeling comes in handy in the scripts when synchronizing the selected map and its label.

The label objects are nested inside the background map object. Therefore, the coordinates for the labels are relative to the coordinate system of the background map, not the page. That's why the first label has a `top` property of zero.

While both the background map and help panel are also positionable elements, scripts need to read the positions of these elements without first setting the values. Recall that in the IE4+ and W3C DOMs, the `style` property of an object does not reveal property values that are set in remote stylesheet rules. Although there are ways to read the effective style properties applied to an element from stylesheets (the `currentStyle` property in IE5+ and the `window.getComputedStyle()` method in NN6+/Moz), neither IE4 nor Safari 1.0 afford that luxury. Therefore, the stylesheet rules for the background map and help panel are specified as `style` attributes in those two elements' tags later in the listing.

```
      <style type="text/css">
      .labels {position:absolute;
         background-color:red; layer-background-color:red;
         width:100; height:28; border:none; text-align:center}
      #azlabel {left:310; top:0}
      #calabel {left:310; top:29}
      #orlabel {left:310; top:58}
      #utlabel {left:310; top:87}
      #walabel {left:310; top:116}
      #nvlabel {left:310; top:145}
      #idlabel {left:310; top:174}

      #camap {position:absolute; left:20; top:100; width:1;}
      #ormap {position:absolute; left:60; top:100; width:1;}
      #wamap {position:absolute; left:100; top:100; width:1;}
      #idmap {position:absolute; left:140; top:100; width:1;}
      #nvmap {position:absolute; left:180; top:100; width:1;}
      #azmap {position:absolute; left:220; top:100; width:1;}
      #utmap {position:absolute; left:260; top:100; width:1;}

      #congrats {position:absolute; visibility:hidden; left:20; top:100;
         width:1; color:red}
      </style>
```

The next statement loads the external `.js` library file that contains the API described in Chapter 47. I tend to load external library files before listing any other JavaScript code in the page, just in case the main page code relies on global variables or functions in its initializations.

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

Now comes the main script, which contains all the document-specific functions and global variables. Global variables here are ready to hold information about the selected state object (and associated details), as well as the offset between the position of a click inside a map object and the top-left corner of that map object. You will see that this offset is important to allow the map to track the cursor at the same offset position within the map. And because the tracking is done by repeated calls to a function (triggered by numerous mouse events), these offset values must have global scope.

```
// global declarations
var offsetX = 0;
var offsetY = 0;
var selectedObj;
var states = new Array();
var statesIndexList = new Array();
var selectedStateLabel;
```

As you will see later in the code, an `onload` event handler for the document invokes an initialization function, whose main job is to build the array of objects containing information about each state. The fields for each `state` object record are for the two-letter state abbreviation, the full name (not used in this application, but included for use in a future version), the x and y coordinates (within the coordinate system of the background map) for the exact position of the state, and a Boolean flag to be set to `true` whenever a user correctly places a state. I come back to the last two statements of the constructor function in a moment.

Calculating the data for the x and y coordinates required some legwork during development. As soon as I had the pieces of art for each state and the code for dragging them around the screen, I disengaged the part of the script that tested for accuracy. Instead, I added a statement to the code that revealed the x and y position of the dragged item in the status bar (rather than being bothered by alerts). When I carefully positioned a state in its destination, I copied the coordinates from the status bar into the statement that created that state record. Sure, it was tedious, but after I had that info in the database, I could adjust the location of the background map and not have to worry about the destination coordinates, because they were based on the coordinate system inside the background map.

```
// object constructor for each state; preserves destination
// position; invokes assignEvents()
function state(abbrev, fullName, x, y) {
   this.abbrev = abbrev;
   this.fullName = fullName;
   this.x = x;
   this.y = y;
   this.done = false;
   assignEvents(this);
   statesIndexList[statesIndexList.length] = abbrev;
}
// initialize array of state objects
function initArray() {
   states["ca"] = new state("ca", "California", 7, 54);
   states["or"] = new state("or", "Oregon", 7, 24);
   states["wa"] = new state("wa", "Washington", 23, 8);
   states["id"] = new state("id", "Idaho", 48, 17);
   states["az"] = new state("az", "Arizona", 45, 105);
   states["nv"] = new state("nv", "Nevada", 27, 61);
   states["ut"] = new state("ut", "Utah", 55, 69);
}
```

The act of creating each `state` object causes all statements in the constructor function to execute. Moreover, they were executing within the context of the object being created. That opened up channels for two important processes in this application. One was to maintain a

list of abbreviations as its own array. This becomes necessary later on when the script needs to loop through all objects in the `states` array to check their `done` properties. Because the array is set up like a hash table (with string index values), a `for` loop using numeric index values is out of the question. So, this extra `statesIndexList` array provides a numerically indexed array that can be used in a `for` loop; values of that array can then be used as index values of the `states` array. Yes, it's a bit of indirection, but other parts of the application benefit greatly by having the state information stored in a hash-table-like array.

We now come to the functions that operate while the user interacts with the map puzzle pieces. The first function, acting as a vital behind-the-scenes utility function, is called `setSelectedMap()`. It receives as its sole parameter an event object that is of the proper type for the browser currently running (that's done in the `engage()` function, described next). This function has three jobs to do, two of which set global variables. The first global variable, `selectedObj`, maintains a reference to the layer (puzzle piece) being dragged by the user. At the same time, the `selectedStateLabel` variable holds onto a reference to the layer that holds the label (recall that its color changes during dragging and release). All of this requires DOM-specific references that are generated through the aid of object detecting branches of the function. The last job of this function is to set the stacking order of the selected map to a value higher than the others so that while the user drags the map, it is in front of everything else on the page.

To assist in establishing references to the map and label layers, naming conventions of the HTML objects (shown later in the code) play an important role. Despite the event handlers being assigned to the `div`s that hold the images, the mouse events are actually targeted at the image objects. The code must associate some piece of information about the event target with the `div` that holds it ("parent" types of references don't work across all browsers, so we have to make the association the hard way). To prevent conflicts with so many objects on this page named with the lowercase abbreviations of the states, the image objects are assigned uppercase abbreviations of the state names. As `setSelectedMap()` begins to execute, it uses object detection to extract a reference to the element object regarded as the target of the event (`target` in W3C DOM browsers, `srcElement` in IE). To make sure that the event being processed comes from an image, the next statement makes sure that the target has both `name` and `src` properties, in which case a lowercase version of the name is assigned to the `abbrev` local variable (if only IE4+ and W3C DOMs were in play here, a better verification is checking that the `tagName` property of the event target is `img`). That `abbrev` variable then becomes the basis for element names used in references to objects assigned to `selectedObj` and `selectedStateLabel`. Notice how the NN4 version requires a double-layer nesting to the reference for the label because labels are nested inside the `bgmap` layer.

The presence of a value assigned to `selectedObj` becomes an important case for all three drag-related functions later. That's why the `setSelectedMap()` function nulls out the value if the event comes from some other source.

```
/**********************************************
BEGIN INTERACTION FUNCTIONS
**********************************************/

// set global reference to map being engaged and dragged
function setSelectedMap(evt) {
   var target = (evt.target) ? evt.target : evt.srcElement;
   var abbrev = (target.name && target.src) ?
      target.name.toLowerCase() : "";
   if (abbrev) {
      if (document.layers) {
         selectedObj = document.layers[abbrev + "map"];
         selectedStateLabel =
            document.layers["bgmap"].document.layers[abbrev + "label"];
```

```
            } else if (document.all) {
               selectedObj = document.all(abbrev + "map");
               selectedStateLabel = document.all(abbrev + "label");
            } else if (document.getElementById) {
               selectedObj = document.getElementById(abbrev + "map");
               selectedStateLabel = document.getElementById(abbrev + "label");
            }
            setZIndex(selectedObj, 100);
            return;
         }
         selectedObj = null;
         selectedStateLabel = null;
         return;
      }
```

Next comes the `engage()` function definition. This function is invoked by `mousedown` events inside any of the state map layers (event handler assignment code comes later). NN4+/W3C DOM browsers pass an event object as the sole parameter to the function (picked up by the `evt` parameter variable). If that parameter contains a value, it stands as the event object for the rest of the processing; but for IE, the `window.event` object is assigned to the `evt` variable. After setting the necessary object globals through `setSelectedMap()`, the next major task for `engage()` is to calculate and preserve in global variables the number of pixels within the state map layer at which the `mousedown` event occurred. By preserving these values, the `dragIt()` function makes sure that the motion of the state map layer keeps in sync with the mouse cursor at the very same point within the state map. If it weren't for taking the offset into account, the layer would jump unexpectedly to bring the top-left corner of the layer underneath the cursor. That's not how users expect to drag items on the screen.

The calculations for the offsets require a variety of DOM-specific properties. For example, both NN4 and NN6+ offer `pageX` and `pageY` properties of the `event` object, but the coordinates of the layer itself require `left`/`top` properties for NN4 and `offsetLeft`/`offsetTop` properties for NN6+. A nested object detection takes place in each assignment statement. The IE branch has some additional branching within each of the assignment statements. These extra branches cover a disparity in the way WinIE and MacIE report the offset properties of an event. WinIE ignores window scrolling, while MacIE takes scrolling into account. Later calculations for positioning must take window scrolling into account, so that scrolling is factored into the preserved offset global values if there are indications that the window has scrolled and the values are being affected by the scroll (in which case the offset values go very negative). The logic is confusing, and it won't make much sense until you see later how the positioning is invoked. Conceptually, all of these offset value calculations may seem like a can of worms, but they are essential, and are performed amazingly compactly.

After the offsets are established, the state's label layer's background color is set to yellow. The function ends with `return false` to make sure that the `mousedown` event doesn't propagate through the page (causing a contextual menu to appear on the Macintosh, for instance).

```
// set relevant globals onmousedown; set selected map
// object global; preserve offset of click within
// the map coordinates; set label color to yellow
function engage(evt) {
   evt = (evt) ? evt : event;
   setSelectedMap(evt);
   if (selectedObj) {
      if (evt.pageX) {
         offsetX = evt.pageX - ((selectedObj.offsetLeft) ?
            selectedObj.offsetLeft : selectedObj.left);
         offsetY = evt.pageY - ((selectedObj.offsetTop) ?
            selectedObj.offsetTop : selectedObj.top);
```

```
            } else if (evt.offsetX || evt.offsetY) {
               offsetX = evt.offsetX - ((evt.offsetX < -2) ?
                  0 : document.body.scrollLeft);
               offsetY = evt.offsetY - ((evt.offsetY < -2) ?
                  0 : document.body.scrollTop);
            }
            setBGColor(selectedStateLabel,"yellow");
            return false;
         }
      }
```

The `dragIt()` function, compact as it is, provides the main action in the application by keeping a selected state object under the cursor as the user moves the mouse. This function is called repeatedly by the `mousemove` events, although the actual event handling methodology varies with platform (precisely the same way as with `engage()`, as shown previously). Regardless of the event property detected, event coordinates (minus the previously preserved offsets) are passed the `shiftTo()` function in the API.

Before the dragging action branch of the function ends, the event object's `cancelBubble` property is set to `true`. In truth, only the IE4+ and W3C DOM event objects have such a property, but assigning a value to a nonexistent object property for NN4 does no harm. It's important that this function operate as quickly as possible, because it must execute with each `mousemove` event. Canceling event bubbling helps in a way, but more important, the cancellation allows the `mousemove` event to be used for other purposes, as described in a moment.

```
      // move div on mousemove
      function dragIt(evt) {
         evt = (evt) ? evt : event;
         if (selectedObj) {
            if (evt.pageX) {
               shiftTo(selectedObj, (evt.pageX - offsetX), (evt.pageY -
                  offsetY));
            } else if (evt.clientX || evt.clientY) {
               shiftTo(selectedObj, (evt.clientX - offsetX), (evt.clientY -
                  offsetY));
            }
            evt.cancelBubble = true;
            return false;
         }
      }
```

When a user drops the currently selected map object, the `release()` function invokes the `onTarget()` function to find out if the current location of the map is within range of the desired destination. If it is in range, the background color of the state label object is set to green, and the done property of the selected state's database entry is set to `true`. One additional test (the `isDone()` function call) looks to see if all the done properties are `true` in the database. If so, the `congrats` object is shown. But if the object is not in the right place, the label reverts to its original red color. In case the user moves a state that was previously okay, its database entry is also adjusted. No matter what the outcome, however, the user has dropped the map, so key global variables are set to `null` and the layer order for the item is set to zero (bottom of the heap) so that it doesn't interfere with the next selected map.

One more condition is possible in the `release()` function. As shown later in the initialization function, the `document` object's `onmousemove` event handler is assigned to the `release()` function (to compare the `onmousemove` events for the state maps go to `dragIt()`). The reasoning behind this document-level event assignment is that no matter how streamlined the dragging function may be, it is possible for the user to move the mouse so fast that the map can't keep up. At that point, `mousemove` events are firing at the `document` (or other object,

eventually bubbling up to the `document`), and not the state map. If that happens while a state map is registered as the selected object, but the image is no longer the target of the event, the code performs the same act as if the user had released the map. The label reverts to red, and all relevant globals are set to `null`, preventing any further interaction with the map until the user mouses down again on the map.

```
// onmouseup, see if dragged map is near its destination
// coordinates; if so, mark it as 'done' and color label green
function release(evt) {
   evt = (evt) ? evt : event;
   var target = (evt.target) ? evt.target : evt.srcElement;
   var abbrev = (target.name && target.src) ?
      target.name.toLowerCase() : "";
   if (abbrev && selectedObj) {
      if (onTarget(evt)) {
         setBGColor(selectedStateLabel, "green");
         states[abbrev].done = true;
         if (isDone()) {
            show("congrats");
         }
      } else {
         setBGColor(selectedStateLabel, "red");
         states[abbrev].done = false;
         hide("congrats");
      }

      setZIndex(selectedObj, 0);
   } else if (selectedStateLabel) {
      setBGColor(selectedStateLabel, "red");
   }
   selectedObj = null;
   selectedStateLabel = null;
}
```

To find out if a dropped map is in (or near) its correct position, the `onTarget()` function first calculates the target spot on the page by adding the location of the `bgmap` object to the coordinate positions stored in the `states` database. Because the `bgmap` object doesn't come into play in other parts of this script, it is convenient to pass merely the object name to the two API functions that get the object's left and top coordinate points.

Next, the script uses platform-specific properties to get the recently dropped state map object's current location. A large `if` condition checks whether the state map object's coordinate point is within a 4-pixel square region around the target point. If you want to make the game easier, you can increase the cushion values from 2 to 3 or 4.

If the map is within the range, the script calls the `shiftTo()` API function to snap the map into the exact destination position and reports back to the `release()` function the appropriate Boolean value.

```
// compare position of dragged element against the destination
// coordinates stored in corresponding state object; after shifting
// element to actual destination, return true if item is within
// 2 pixels.
function onTarget(evt) {
   evt = (evt) ? evt : event;
   var target = (evt.target) ? evt.target : evt.srcElement;
   var abbrev = (target.name && target.src) ?
      target.name.toLowerCase() : "";
```

```
      if (abbrev && selectedObj) {
         var x = states[abbrev].x + getObjectLeft("bgmap");
         var y = states[abbrev].y + getObjectTop("bgmap");
         var objX, objY;
         if (selectedObj.pageX) {
            objX = selectedObj.pageX;
            objY = selectedObj.pageY;
         } else if (selectedObj.style) {
            objX = parseInt(selectedObj.style.left);
            objY = parseInt(selectedObj.style.top);
         }
         if ((objX >= x-2 && objX <= x+2) && (objY >= y-2 && objY <= y+2)) {
            shiftTo(selectedObj, x, y);
            return true;
         }
         return false;
      }
      return false;
   }
```

A `for` loop cycles through the `states` database (with the help of the hash table values stored indirectly in the `statesIndexList` array) to see if all of the done properties are set to `true`. When they are, the `release()` function (which calls the `isDone()` function) displays the congratulatory object.

```
   // test whether all state objects are marked 'done'
   function isDone() {
      for (var i = 0; i < statesIndexList.length; i++) {
         if (!states[statesIndexList[i]].done) {
            return false;
         }
      }
      return true;
   }
```

The help panel is created differently than the map and label objects (details coming up in a moment). When the user clicks the Help button at the top of the page, the instructions panel flies in from the right edge of the window (see Figure 56-2). The `showHelp()` function begins the process by setting its location to the current right window edge, bringing its layer to the very front of the heap, showing the object. To assist `moveHelp()` in calculating the center position on the screen, the `showHelp()` function retrieves (just once per showing) the DOM-specific property for the width of the help panel. That value is passed as a parameter to `moveHelp()` as it is repeatedly invoked through the `setInterval()` mechanism.

```
   /**********************************************
   BEGIN HELP ELEMENT FUNCTIONS
   **********************************************/
   // initiate show action
   function showHelp() {
      var objName = "help";
      var helpWidth = 0;
      shiftTo(objName, insideWindowWidth, 80);
      setZIndex(objName,1000);
      show(objName);
      if (document.layers) {
         helpWidth = document.layers[objName].document.width;
      } else if (document.all) {
         helpWidth = document.all(objName).offsetWidth;
```

```
   } else if (document.getElementById) {
      if (document.getElementById(objName).offsetWidth >= 0) {
         helpWidth = document.getElementById(objName).offsetWidth;
      }
   }
   intervalID = setInterval("moveHelp(" + helpWidth + ")", 1);
}
```

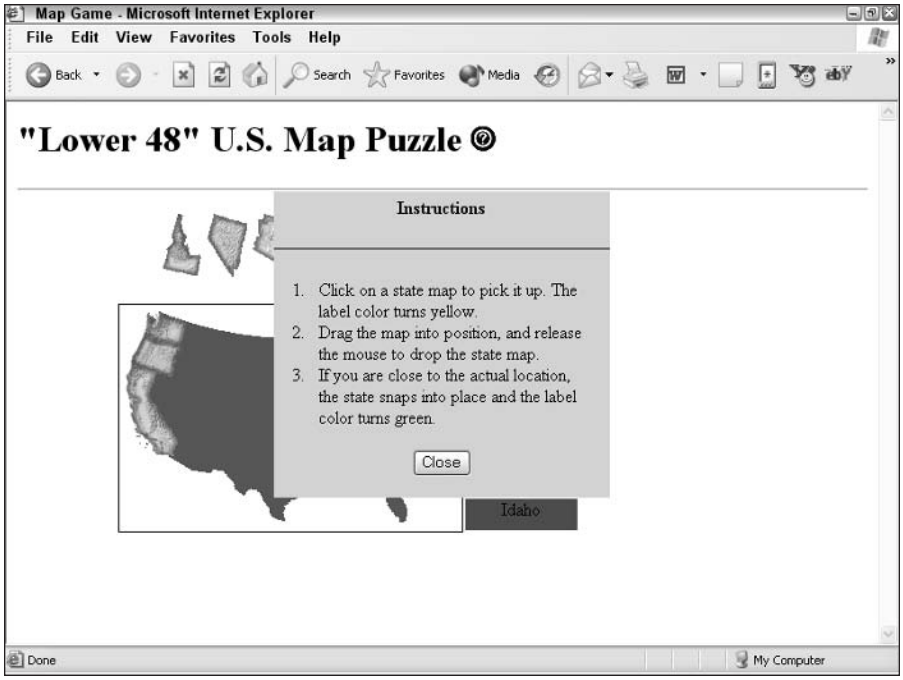


Figure 56-2: Instruction panel "flies" in from left to center screen.

In the `moveHelp()` function, the help object is shifted in 5-pixel increments to the left. The ultimate destination is the spot where the object is in the middle of the browser window. That midpoint must be calculated each time the page loads, because the window may have been resized. The width of the `help` object, received as a parameter to the function, gets a workout in the mid-point calculation.

This function is called repeatedly under the control of a `setInterval()` method in `showHelp()`. But when the object reaches the middle of the browser window, the interval ID is canceled, which stops the animation.

The help object processes a mouse event to hide the object. An extra `clearInterval()` method is called here in case the user clicks the object's Close button before the object has reached mid-window (where `moveHelp()` cancels the interval). The script also shifts the position to the right edge of the window, but it isn't absolutely necessary, because the `showHelp()` method positions the window there.

```
// iterative move help div to center of window
function moveHelp(w) {
   shiftBy("help",-5,0);
   var objectLeft = getObjectLeft("help");
```

```
         if (objectLeft <= (insideWindowWidth/2) - w/2) {
            clearInterval(intervalID);
         }
      }
      // hide the help div
      function hideMe() {
         clearInterval(intervalID);
         hide("help");
         shiftTo("help", insideWindowWidth, 80);
      }
```

The document's `onload` event handler invokes the `init()` function, which, in turn, calls two functions and assigns the `document` object's mouse event handlers. The first function is `initArray()`, which builds the `states[]` database and assigns event handlers to the state map layers. Because the layers are defined so late in the document, initializing their events after the page has loaded is safest.

For convenience in moving the help window to the center of the browser window, the `setWinWidth()` function sets a global variable (`insideWindowWidth`) to hold the width of the browser window. This function is also invoked by the `onresize` event handler for the window to keep the value up to date.

The most important parts of the `init()` function are the event handler assignments. The trio — engaging the map on `mousedown`, dragging it on `mousemove`, releasing it on `mouseup` — are assigned to the `document` object so that the events track correctly even if the cursor speeds past a puzzle piece's edge on a fast drag action.

```
      // calculate center of window for help div
      function setWinWidth() {
         if (window.innerWidth) {
            insideWindowWidth = window.innerWidth;
         } else if (document.body.scrollWidth) {
            insideWindowWidth = document.body.scrollWidth;
         } else if (document.width) {
            insideWindowWidth = document.width;
         }
      }

      /***********************************************
      INITIALIZE THE APPLICATION
      ***********************************************/
      // initialize application
      function init() {
         initArray();
         setWinWidth();
         if (document.layers) {
            document.captureEvents(Event.MOUSEDOWN | Event.MOUSEMOVE |
Event.MOUSEUP);
         }
         document.onmousedown = engage;
         document.onmousemove = dragIt;
         document.onmouseup = release;
      }
      </script>
   </head>
```

Now comes the part of the document that generates the visible content. The `<body>` tag contains the two event handlers just discussed. An image rollover for the help icon simply displays a message in the status bar.

```
<body onload="init()" onresize="setWinWidth()">
   <h1>"Lower 48" U.S. Map Puzzle <a href="javascript:void showHelp()"
      onmouseover="status='Show help panel...';return true"
      onmouseout="status='';return true"><img alt="image" src="info.gif"
      height="22" width="22" border="0" /></a></h1>
   <hr />
```

Next come tags for all of the `div` elements. The `style` attribute for the `bgmap div` lets scripts read the positioned values to assist in calculating positions in the `onTarget()` function, as shown previously. The `bgmap` layer also contains all labels so that if the design calls for moving the map to another part of the page, the labels follow automatically. Notice how the lowercase state abbreviations are part of the names of both the label and map layers. As you saw in a few functions shown previously, a systematic approach to object naming can offer powerful shortcuts in determining references to elements.

```
   <div id="bgmap" style="position:absolute; left:100; top:180; width:406">
      <img alt="image" src="us11.gif" width="306" height="202"
      border="1" /> 
      <div class="labels" id="azlabel">Arizona</div>
      <div class="labels" id="calabel">California</div>
      <div class="labels" id="orlabel">Oregon</div>
      <div class="labels" id="utlabel">Utah</div>
      <div class="labels" id="walabel">Washington</div>
      <div class="labels" id="nvlabel">Nevada</div>
      <div class="labels" id="idlabel">Idaho</div>
   </div>
   <div id="camap"><img alt="image" name="CA" src="ca.gif" width="47"
      height="82" border="0" /></div>
   <div id="ormap"><img alt="image" name="OR" src="or.gif" width="57"
      height="45" border="0" /></div>
   <div id="wamap"><img alt="image" name="WA" src="wa.gif" width="38"
      height="29" border="0" /></div>
   <div id="idmap"><img alt="image" name="ID" src="id.gif" width="34"
      height="55" border="0" /></div>
   <div id="azmap"><img alt="image" name="AZ" src="az.gif" width="38"
      height="45" border="0" /></div>
   <div id="nvmap"><img alt="image" name="NV" src="nv.gif" width="35"
      height="56" border="0" /></div>
   <div id="utmap"><img alt="image" name="UT" src="ut.gif" width="33"
      height="41" border="0" /></div>
   <div id="congrats"><h1>Congratulations!</h1></div>
```

In developing this application, I encountered an unfriendly NN4 bug. When defining the help panel as a positioned `div` element in NN4, the browser exhibited unwanted behavior after the instruction panel was shown and flown into place under script control. Even after hiding the help layer, the page no longer received mouse events, making it impossible to pick up a state map after the instructions appeared. The problem did not surface, however, if the help object was defined in the document with a `<layer>` tag.

Therefore, I did what I don't like to do unless absolutely necessary: I created branches in the content that used `document.write()` to create the same content with different HTML syntax, depending on the browser. For non-`layer` browsers, the page creates the same kind of block (with the `<div>` tag pair) used elsewhere in the document. Positioning properties are assigned to this block via a `style` attribute in the `<div>` tag. You cannot assign a style in the `<style>` tag that is visible to the entire document, because that specification and a like-named `<layer>` tag get confused.

For NN4, the page uses the `<layer>` tag and loads the content of the object from a separate HTML file (`instrux.htm`). One advantage I had with the `<layer>` tag was that I could assign an initial horizontal position of the help object with a JavaScript entity. The entity reaches into the `window.innerWidth` property to set the `left` attribute of the layer.

```
      <script type="text/javascript">
      var output = "";
      if (document.layers) {
         output = "<layer id='help' top='80' left=&{window.innerWidth};
            width='300' visibility='hidden' src='instrux.htm'></layer>";
      } else {
         output = "<div id='help' onclick='hideMe()' style='position:absolute;
            visibility:hidden; top:80; width:300; border:none;
            background-color:#98FB98;'>\n";
         output += "<p style='margin-top:5'>
            <center><b>Instructions<\/b><\/center><\/p>\n";
         output += "<hr color='seagreen' \/>\n<ol style='margin-right:20'>";
         output += "<li>Click on a state map to pick it up. The label color
            turns yellow.<\/li>";
         output += "<li>Drag the map into position, and release the mouse to
            drop the state map.<\/li>";
         output += "<li>If you are close to the actual location, the state
            snaps into place and the label color turns green.<\/li>";
         output += "<\/ol>\n<form>\n<center><input type='button'
            value='Close'>\n<\/form><\/div>";
      }
      document.write(output);
      </script>
   </body>
</html>
```

This page has a lot of code to digest in one reading. Run the application, study the structure of the source code listing file, and re-read the previous explanations. It may take several readings for a mental picture of the application to form.

Lessons Learned

As soon as the external cross-platform API was in place, it helped frame a lot of the other code in the main program. The APIs provided great comfort in that they encouraged me to reference a complex object fully in the main code as a platform-shared value (for example, the `selectedObj` and `selectedStateLabel` global variables). At the same time, I could reference top-level elements (that is, non-nested objects) simply by their names when passing them to API functions.

In many respects, the harder task was defining the stylesheet attributes and syntax that both browsers would treat similarly. In the case of the label objects, I couldn't reach complete parity in a cross-platform environment (the labels look different in NN4), and in the case of the `help` object, I had to code the HTML separately for each platform. Therefore, when approaching this kind of project, work first with the HTML and CSS syntax to build the look that works best for all platforms. Then start connecting the scripted wires. You may have to adjust the CSS code if you find odd behavior in one platform or the other with your scripting, but starting with a good layout is still easier.

But without a doubt the biggest lesson you learn from working on a project like this is how important it is to test an application on as many browsers and operating systems as possible. Designing a cross-platform application on one browser and having it run flawlessly on the other the first time is nearly impossible. Be prepared to go back and forth among multiple browsers, breaking and repairing existing working code along the way until you eventually reach a version that works on every browser that you can test.

✦ ✦ ✦

CHAPTER 57

Application: Transforming XML Data

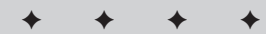
In This Chapter

Mapping XML data to JavaScript objects

Complex JavaScript data structures

Advanced array sorting

Dynamic tables



Chapter 52 ends with an example of an interactive outliner whose data arrives from an external XML file, a feature supported by the Windows versions of IE5 or later and Mozilla-based browsers. The application described in this chapter picks up from there.

As you recall from the Chapter 52 outline, the node structure of the XML data was used as a guide to the structure for a one-time rendering of HTML elements. There was a one-to-one correlation between XML element nesting and the HTML element nesting. Adjusting stylesheet properties for displaying or hiding elements controlled all interactivity. What you're about to see here is a case for converting XML into JavaScript objects that can be used multiple times as a convenient data source for HTML that is displayed in any number of formats. In particular, you see how JavaScript's array sorting prowess supplies XML-supplied data with extraordinary flexibility in presentation.

You will see a lot of code in this chapter. The code is presented here as a way to demonstrate the potential for rich data handling. At the same time, the code may provide ideas for server-side processing of XML data being output to the client. If a server program can convert the XML data into the shortcut object and array notation of version 4 browsers or later, suddenly a broader range of browsers is capable of dealing with data stored as XML on the server.

Application Overview

Understanding the data is a good place to start in describing this application. The scenario is a small American company (despite its grandiose name: GiantCo) that has divided the country into three sales regions. Two of the regions have two sales representatives, while the third region has three reps. The time is at the end of a fiscal year, at which point the management wants to review and present the performance of each salesperson. An XML report (`salesrpt.xml`) delivers the sales forecast and actual sales per quarter for each sales rep. An HTML and JavaScript page is charged with not only loading the XML data displaying the raw tabular data, but also allowing for a variety of views and sorting possibilities so that management can analyze performance by sales rep and region, as well as by quarter.

A server-based searching and reporting program collects the requested data and outputs each sales rep's record in an XML structure, such as the following one excerpted from the `salesrpt.xml` file:

```
<salesrep>
   <employeeid>12345</employeeid>
   <contactinfo>
      <firstname>Brenda</firstname>
      <lastname>Smith</lastname>
      <email>brendas@giantco.com</email>
      <phone>312-555-9923</phone>
      <fax>312-555-9901</fax>
   </contactinfo>
   <manager>
      <employeeid>02934</employeeid>
      <firstname>Alistair</firstname>
      <lastname>Renfield</lastname>
   </manager>
   <region>Central</region>
   <salesrecord>
      <period>
         <id>Q1_2000</id>
         <forecast>300000</forecast>
         <actual>316050</actual>
      </period>
      <period>
         <id>Q2_2000</id>
         <forecast>280000</forecast>
         <actual>285922</actual>
      </period>
      <period>
         <id>Q3_2000</id>
         <forecast>423000</forecast>
         <actual>432930</actual>
      </period>
      <period>
         <id>Q4_2000</id>
         <forecast>390000</forecast>
         <actual>399200</actual>
      </period>
   </salesrecord>
</salesrep>
```

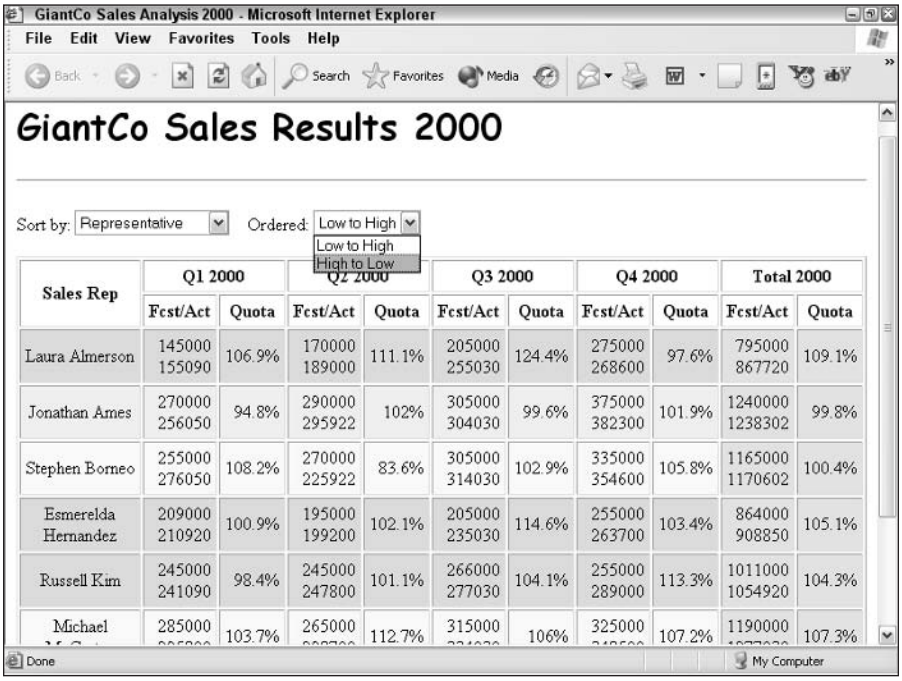
As you can see, the data consists of several larger blocks, such as contact information, a pointer to the rep's manager, and then the details of each quarterly period's forecast and

actual sales. The goal is to present the data in table form with a structure similarly shown in Figure 57-1. Not only is the raw data presented, but numerous calculations are also made on the results, such as the percentage of quota attained for each reporting period, plus totals along each axis of the spreadsheet-like table.

Just above the table are two `select` elements. These controls' labels indicate that the table's data can be sorted by a number of criteria and the results of each sort can be ordered in different ways. Sorting in the example offers the following possibilities:

```
Representative's Name
Sales Region
Q1 Forecast
Q1 Actual
Q1 Performance
[the last three also for Q2, Q3, Q4]
Total Forecast
Total Actual
Total Performance
```

Ordering of the sorted results is a choice between "Low to High" or "High to Low." While ordering of most sort categories is obviously based on numeric value, the sorting of the representatives' names is based on the alphabetical order of the last names. One other point about the user interface is that the design needs to signify via table cell background color the sales region of each representative. The colors aren't easily distinguishable in Figure 57-1, but if you open the actual example listing in WinIE5+ or NN7+/Moz on your computer, you will see the coloration.



Sales Rep	Q1 2000		Q2 2000		Q3 2000		Q4 2000		Total 2000	
	Fcst/Act	Quota	Fcst/Act	Quota	Fcst/Act	Quota	Fcst/Act	Quota	Fcst/Act	Quota
Laura Almerson	145000 155090	106.9%	170000 189000	111.1%	205000 255030	124.4%	275000 268600	97.6%	795000 867720	109.1%
Jonathan Ames	270000 256050	94.8%	290000 295922	102%	305000 304030	99.6%	375000 382300	101.9%	1240000 1238302	99.8%
Stephen Borneo	255000 276050	108.2%	270000 225922	83.6%	305000 314030	102.9%	335000 354600	105.8%	1165000 1170602	100.4%
Esmerelda Hernandez	209000 210920	100.9%	195000 199200	102.1%	205000 235030	114.6%	255000 263700	103.4%	864000 908850	105.1%
Russell Kim	245000 241090	98.4%	245000 247800	101.1%	266000 277030	104.1%	255000 289000	113.3%	1011000 1054920	104.3%
Michael	285000	103.7%	265000	112.7%	315000	106%	325000	107.2%	1190000	107.3%

Figure 57-1: One view of the XML data output.

Implementation Plan

Clearly all the data needed for numerous sorted and ordered views arrives in one batch from the XML file. Despite the element and node referencing properties and methods of the W3C DOM, trying to use the XML elements as the sole data store for scripts to sort the data each time would be impractical. For one thing, none of the elements have ID attributes — there's no need for it in the XML stored on the server database. And even if they did have IDs, how would scripts that you desire to write for generalizability make use of them unless the IDs were generated in a well-known sequence? Moreover, after a sales rep's record is rendered in the table, how easy would it be to dive back into that record and drill down for further information, such as the name of a representative's manager?

A solution that can empower the page author in this case is to use the node-walking properties and methods of the W3C DOM to assemble a JavaScript-structured database while the page loads. In other words, the conversion is performed just once during page loading, and the JavaScript version is preserved in an array (of XML "records" in this case) as a global variable. Any transformations on the data can be done from the JavaScript database with the help of additional powers of the language.

Given that route, the basic operation of the scripting of the page is schematically simple:

1. Convert the XML into an array of objects at load time.
2. Predefine all necessary sorting functions based on properties of those objects.
3. Provide a function that rebuilds the HTML table each time data is sorted.

With this sequence in mind, now look into the code that does the job.

The Code

Rather than work through the long document in source code order, the following descriptions follow a more functional order. You can open the actual source code file (`salesrpt.htm`) to see where the various functions are positioned. To best understand this application, seeing the "how" rather than the "where" is more important. Also, many of the code lines (even some single expressions) are too wide for the printed page and therefore break unnaturally in the listings that follow. Trust the formatting of the source file on the CD-ROM.

Style sheets

For the example provided on the CD-ROM, one set of stylesheet rules is embedded in the HTML document. As you can see from the rule selectors, many are tied to very specific classes of table-related elements used to render the content. In a production version of this application, I would expect that there would be more and quite different views of the data available to the users, such as bar charts for each salesperson or region. Each view would likely require its own unique set of stylesheet rules. In such a scenario, the proper implementation would be to use the LINK element to bring in a different external stylesheet file for each view type. All could be linked in at the outset, but only the current `styleSheet` object would be enabled.

```
<style type="text/css">
td {text-align:right}
td.rep, td.grandTotalLabel {text-align:center}
```

```
tr.East {background-color:#FFFFCC}
tr.Central {background-color:#CCFFFF}
tr.West {background-color:#FFCCCC}
tr.QTotal {background-color:#FFFF00}
td.repTotal {background-color:#FFFF00}
td.grandTotal{background-color:#00FF00}
h1 {font-family:"Comic Sans MS",Helvetica,sans-serif}
</style>
```

One stylesheet rule is essential: The one that suppresses the rendering of any XML element. That data is hidden from the user's view.

Initialization sequence

An `onload` event handler invokes the `init()` function, which sets a lot of machinery in motion to get the document ready for user interaction. After loading the XML file into the `xDoc` global variable (via the `loadXMLDoc()` function), its most important job is running a `for` loop that builds the JavaScript database from the XML elements. Next, it sorts the database based on the current choice in the sorting `select` element. The sorting function ends by triggering the rendering of the table. These three actions correspond to the fundamental operation of the entire application.

```
// initialize global variable that stores JavaScript data
var db = new Array();

// Initialization called by onLoad
function init() {
   loadXMLDoc("salesrpt.xml");
   if (xDoc) {
      for (var i = 0; i < xDoc.getElementsByTagName("salesrep").length; i++) {
         db[db.length] = getOneSalesRep(i);
      }
      selectSort(document.getElementById("sortChooser"));
   }
}
```

Converting the data

The controlling factor for creating the JavaScript database is the structure of the XML data. With the complete XML document stored in the `xDoc` variable, scripts can use DOM methods to look for elements bearing relevant tag names. Data for each sales rep is contained by a `salesrep` element. The number of `salesrep` elements determines how many records (JavaScript objects) are to be added to the `db` array. A call to the `getOneSalesRep()` function creates an object for each sales representative's data.

Despite the length of the `getOneSalesRep()` function, its operation is very straightforward. Most of the statements do nothing more than retrieve the data inside the various XML elements within a `salesrep` container and assign that data to a like-named property of the custom object. Following the structure of the XML example shown earlier in this chapter, you can see where some properties of a JavaScript object representing the data are, themselves, objects or arrays. For example, one of the properties is called `manager`, corresponding to the `manager` element. But that element has nested items inside. Then, making those nested elements properties of a `manager` object is only natural. Similarly, the repetitive nature of the data within each of the four quarterly periods calls for even greater nesting: The object property named `sales` is an array, with each item of the array

corresponding to one of the periods. Each period also has three properties (a period ID, forecast sales, and actual sales). Thus, the `sales` property is an array of objects.

```
function getOneSalesRep(i) {
   var oneRecord = new Object();
   var oneElem = xDoc.getElementsByTagName("salesrep")[i];
   oneRecord.id =
      oneElem.getElementsByTagName("employeeid")[0].firstChild.data;
   var contactInfoElem = oneElem.getElementsByTagName("contactinfo")[0];
   oneRecord.firstName =
      contactInfoElem.getElementsByTagName("firstname")[0].firstChild.data;
   oneRecord.lastName =
      contactInfoElem.getElementsByTagName("lastname")[0].firstChild.data;
   oneRecord.eMail =
      contactInfoElem.getElementsByTagName("email")[0].firstChild.data;
   oneRecord.phone =
      contactInfoElem.getElementsByTagName("phone")[0].firstChild.data;
   oneRecord.fax =
      contactInfoElem.getElementsByTagName("fax")[0].firstChild.data;
   oneRecord.manager = new Object();
   var oneMgrElem = oneElem.getElementsByTagName("manager")[0];
   oneRecord.manager.id =
      oneMgrElem.getElementsByTagName("employeeid")[0].firstChild.data;
   oneRecord.manager.firstName =
      oneMgrElem.getElementsByTagName("firstname")[0].firstChild.data;
   oneRecord.manager.lastName =
      oneMgrElem.getElementsByTagName("lastname")[0].firstChild.data;
   oneRecord.region =
      oneElem.getElementsByTagName("region")[0].firstChild.data;
   oneRecord.sales = new Array();
   var allPeriods = oneElem.getElementsByTagName("salesrecord")[0].childNodes;
   var temp;
   var accumForecast = 0, accumActual = 0;
   for (var i = 0; i < allPeriods.length; i++) {
      if (allPeriods[i].nodeType == 1) {
         temp = new Object();
         temp.period =
            allPeriods[i].getElementsByTagName("id")[0].firstChild.data;
         temp.forecast = parseInt(allPeriods[i].getElementsByTagName(
            "forecast")[0].firstChild.data);
         temp.actual = parseInt(allPeriods[i].getElementsByTagName(
            "actual")[0].firstChild.data);
         temp.quotaPct = getPercentage(temp.actual, temp.forecast);
         oneRecord.sales[temp.period] = temp;
         accumForecast += temp.forecast;
         accumActual += temp.actual;
      }
   }
   oneRecord.totalForecast = accumForecast;
   oneRecord.totalActual = accumActual;
   oneRecord.totalQuotaPct = getPercentage(accumActual, accumForecast);
   return oneRecord;
}
function getPercentage(actual, forecast) {
   var pct = (actual/forecast * 100) + "";
   pct = pct.match(/\d*\.\d/);
   return parseFloat(pct);
}
```

Assuming that the raw XML database stores only the sales forecast and actual dollar figures, it is up to analysis programs to perform their own calculations, such as how the actual sales compare against the forecasts. As you saw in the illustration of the rendered table, this application not only displays the percentage differences between the pairs of values, but it also provides sorting facilities on those percentages. To speed the sorting, the percentages are calculated as the JavaScript database is being accumulated, and the percentages are stored as properties of each object. Percentage calculation is called upon in two different statements of the `getOneSalesRep()` function, so that the calculation is broken out to its own function, `getPercentage()`. In that function, the two passed values are massaged to calculate the percentage value, and then the string result is formatted to no more than one digit to the right of the decimal (by way of a regular expression). The value returned for the property assignment is converted to a number data type, because sorting on these values needs to be done according to numeric sorting, rather than string sorting.

You can already get a glimpse at the contribution JavaScript is making to the scripted representation of the data transmitted in XML form. By virtue of planning for subsequent calculations, the JavaScript object contains considerably more information than was originally delivered, yet all the properties are derived from "hard" data supplied by the server database.

Sorting the JavaScript database

With so many sorting keys for the user to choose from, it's no surprise that sorting code occupies a good number of script lines in this application. All sorting code consists of two major blocks: *dispatching* and *sorting*.

The dispatching portion is nothing more than one gigantic `switch` construction that sends execution to one of the 17 (!) sorting functions that match whichever sort key is chosen in the `select` element on the page. This dispatcher function, `selectSort()`, is also invoked from the `init()` function at load time. Thus, if the user makes a choice in the page, navigates to another page, and then returns with the page still showing the previous selection, the `onLoad` event handler will reconstruct the table precisely as it was. When sorting is completed, the table is drawn, as you see shortly.

```
function selectSort(chooser) {
   switch (chooser.value) {
   case "byRep" :
      db.sort(sortDBByRep);
      break;
   case "byRegion" :
      db.sort(sortDBByRegion);
      break;
   case "byQ1Fcst" :
      db.sort(sortDBByQ1Fcst);
      break;
   case "byQ1Actual" :
      db.sort(sortDBByQ1Actual);
      break;
   case "byQ1Quota" :
      db.sort(sortDBByQ1Quota);
      break;
   case "byQ2Fcst" :
      db.sort(sortDBByQ2Fcst);
      break;
   case "byQ2Actual" :
      db.sort(sortDBByQ2Actual);
      break;
```

```
      case "byQ2Quota" :
         db.sort(sortDBByQ2Quota);
         break;
      case "byQ3Fcst" :
         db.sort(sortDBByQ3Fcst);
         break;
      case "byQ3Actual" :
         db.sort(sortDBByQ3Actual);
         break;
      case "byQ3Quota" :
         db.sort(sortDBByQ3Quota);
         break;
      case "byQ4Fcst" :
         db.sort(sortDBByQ4Fcst);
         break;
      case "byQ4Actual" :
         db.sort(sortDBByQ4Actual);
         break;
      case "byQ4Quota" :
         db.sort(sortDBByQ4Quota);
         break;
      case "byTotalFcst" :
         db.sort(sortDBByTotalFcst);
         break;
      case "byTotalActual" :
         db.sort(sortDBByTotalActual);
         break;
      case "byTotalQuota" :
         db.sort(sortDBByTotalQuota);
         break;
      }
      drawTextTable();
}
```

Each specific sorting routine is a function that automatically works repeatedly on pairs of entries of an array (see Chapter 30). Array entries here (from the `db` array) are objects — and rather complex objects at that. The benefit of using JavaScript array sorting is that the sorting can be performed on any property of objects stored in the array. For example, sorting on the `lastName` property of each `db` array object is based on a comparison of the `lastName` property for each of the pairs of array entries passed to the `sortDBByRep()` sort function. But looking down a little further, you can see that the mechanism allows sorting on even more deeply nested properties, such as the `sales.Q1_2000.forecast` property of each array entry. If a property in an object can be referenced, it can be used as a sorting property inside one of these functions.

```
function sortDBByRep(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.lastName < b.lastName) ? -1 : 1;
   } else {
      return (a.lastName > b.lastName) ? -1 : 1;
   }
}
function sortDBByRegion(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.region < b.region) ? -1 : 1;
   } else {
      return (a.region > b.region) ? -1 : 1;
   }
}
```

```
function sortDBByQ1Fcst(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.sales.Q1_2000.forecast - b.sales.Q1_2000.forecast);
   } else {
      return (b.sales.Q1_2000.forecast - a.sales.Q1_2000.forecast);
   }
}
function sortDBByQ1Actual(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.sales.Q1_2000.actual - b.sales.Q1_2000.actual);
   } else {
      return (b.sales.Q1_2000.actual - a.sales.Q1_2000.actual);
   }
}
function sortDBByQ1Quota(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.sales.Q1_2000.quotaPct - b.sales.Q1_2000.quotaPct);
   } else {
      return (b.sales.Q1_2000.quotaPct - a.sales.Q1_2000.quotaPct);
   }
}
function sortDBByQ2Fcst(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.sales.Q2_2000.forecast - b.sales.Q2_2000.forecast);
   } else {
      return (b.sales.Q2_2000.forecast - a.sales.Q2_2000.forecast);
   }
}
function sortDBByQ2Actual(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.sales.Q2_2000.actual - b.sales.Q2_2000.actual);
   } else {
      return (b.sales.Q2_2000.actual - a.sales.Q2_2000.actual);
   }
}
function sortDBByQ2Quota(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.sales.Q2_2000.quotaPct - b.sales.Q2_2000.quotaPct);
   } else {
      return (b.sales.Q2_2000.quotaPct - a.sales.Q2_2000.quotaPct);
   }
}
function sortDBByQ3Fcst(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.sales.Q3_2000.forecast - b.sales.Q3_2000.forecast);
   } else {
      return (b.sales.Q3_2000.forecast - a.sales.Q3_2000.forecast);
   }
}
function sortDBByQ3Actual(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.sales.Q3_2000.actual - b.sales.Q3_2000.actual);
   } else {
      return (b.sales.Q3_2000.actual - a.sales.Q3_2000.actual);
   }
}
function sortDBByQ3Quota(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.sales.Q3_2000.quotaPct - b.sales.Q3_2000.quotaPct);
   } else {
```

```
      return (b.sales.Q3_2000.quotaPct - a.sales.Q3_2000.quotaPct);
   }
}
function sortDBByQ4Fcst(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.sales.Q4_2000.forecast - b.sales.Q4_2000.forecast);
   } else {
      return (b.sales.Q4_2000.forecast - a.sales.Q4_2000.forecast);
   }
}
function sortDBByQ4Actual(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.sales.Q4_2000.actual - b.sales.Q4_2000.actual);
   } else {
      return (b.sales.Q4_2000.actual - a.sales.Q4_2000.actual);
   }
}
function sortDBByQ4Quota(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.sales.Q4_2000.quotaPct - b.sales.Q4_2000.quotaPct);
   } else {
      return (b.sales.Q4_2000.quotaPct - a.sales.Q4_2000.quotaPct);
   }
}
function sortDBByTotalFcst(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.totalForecast - b.totalForecast);
   } else {
      return (b.totalForecast - a.totalForecast);
   }
}
function sortDBByTotalActual(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.totalActual - b.totalActual);
   } else {
      return (b.totalActual - a.totalActual);
   }
}
function sortDBByTotalQuota(a, b) {
   if (document.getElementById("orderChooser").value == "inc") {
      return (a.totalQuotaPct - b.totalQuotaPct);
   } else {
      return (b.totalQuotaPct - a.totalQuotaPct);
   }
}
```

For this application, all sorting functions branch in their execution based on the choice made in the "Ordered" `select` element on the page. The relative position of the two array elements under test in these simple subtraction comparison statements reverses when the sort order is from low to high (increasing) and when it is from high to low (decreasing). This kind of array sorting is extremely powerful in JavaScript and probably escapes the attention of most scripters.

Constructing the table

As recommended back in Chapter 37's discussion of `table` and related elements, it is often convenient to manipulate the structure of a `table` element by way of the specialized methods for tables, rather than mess with nodes and elements. The `drawTextTable()` function is devoted to employing those methods to create the rendered contents of the table below the

headers (which are hard-wired in the document's HTML). Composing an 11-column table requires a bit of code, and the `drawTextTable()`'s length attests to that fact. You can tell by just glancing at the code, however, that for big chunks of it, there is a comfortable regularity that is aided by the JavaScript object that holds the data.

Additional calculations take place while the table's elements are being added to the `table` element. Column totals are accumulated during the table assembly (row totals are calculated as the object is generated and preserved as properties of the object). A large `for` loop cycles through each (sorted) row of the `db` array; each row of the `db` array corresponds to a row of the table. Class names are assigned to various rows or cells so that they will pick up the stylesheet rules defined earlier in the document. Another subtlety of this version is that the `region` property of a sales rep is assigned to the `title` property of a row. If the user pauses the mouse pointer anywhere in that row, the name of the region pops up briefly.

```
function drawTextTable() {
   var newRow;
   var accumQ1F = 0, accumQ1A = 0, accumQ2F = 0, accumQ2A = 0;
   var accumQ3F = 0, accumQ3A = 0, accumQ4F = 0, accumQ4A = 0;
   deleteRows(document.getElementById("mainTableBody"));
   for (var i = 0; i < db.length; i++) {
      newRow = document.getElementById("mainTableBody").insertRow(i);
      newRow.className = db[i].region;
      newRow.title = db[i].region + " Region";
      appendCell(newRow, "rep", db[i].firstName + " " + db[i].lastName);
      appendCell(newRow, "Q1", db[i].sales.Q1_2000.forecast + "<br />" +
         db[i].sales.Q1_2000.actual);
      appendCell(newRow, "Q1", db[i].sales.Q1_2000.quotaPct + "%");
      appendCell(newRow, "Q2", db[i].sales.Q2_2000.forecast + "<br />" +
         db[i].sales.Q2_2000.actual);
      appendCell(newRow, "Q2", db[i].sales.Q2_2000.quotaPct + "%");
      appendCell(newRow, "Q3", db[i].sales.Q3_2000.forecast + "<br />" +
         db[i].sales.Q3_2000.actual);
      appendCell(newRow, "Q3", db[i].sales.Q3_2000.quotaPct + "%");
      appendCell(newRow, "Q4", db[i].sales.Q4_2000.forecast + "<br />" +
         db[i].sales.Q4_2000.actual);
      appendCell(newRow, "Q4", db[i].sales.Q4_2000.quotaPct + "%");
      accumQ1F += db[i].sales.Q1_2000.forecast;
      accumQ1A += db[i].sales.Q1_2000.actual;
      accumQ2F += db[i].sales.Q2_2000.forecast;
      accumQ2A += db[i].sales.Q2_2000.actual;
      accumQ3F += db[i].sales.Q3_2000.forecast;
      accumQ3A += db[i].sales.Q3_2000.actual;
      accumQ4F += db[i].sales.Q4_2000.forecast;
      accumQ4A += db[i].sales.Q4_2000.actual;
      appendCell(newRow, "repTotal", db[i].totalForecast + "<br />" +
         db[i].totalActual);
      appendCell(newRow, "repTotal", db[i].totalQuotaPct + "%");
   }
   newRow = document.getElementById("mainTableBody").insertRow(i);
   newRow.className = "QTotal";
   newRow.title = "Totals";
   appendCell(newRow, "grandTotalLabel", "Grand Total");
   appendCell(newRow, "Q1", accumQ1F + "<br />" + accumQ1A);
   appendCell(newRow, "Q1", getPercentage(accumQ1A, accumQ1F) + "%");
   appendCell(newRow, "Q2", accumQ2F + "<br />" + accumQ2A);
   appendCell(newRow, "Q2", getPercentage(accumQ2A, accumQ2F) + "%");
   appendCell(newRow, "Q3", accumQ3F + "<br />" + accumQ3A);
   appendCell(newRow, "Q3", getPercentage(accumQ3A, accumQ3F) + "%");
   appendCell(newRow, "Q4", accumQ4F + "<br />" + accumQ4A);
```

```
   appendCell(newRow, "Q4", getPercentage(accumQ4A, accumQ4F) + "%");
   var grandTotalFcst = accumQ1F + accumQ2F + accumQ3F + accumQ4F;
   var grandTotalActual = accumQ1A + accumQ2A + accumQ3A + accumQ4A;
   appendCell(newRow, "grandTotal", grandTotalFcst + "<br />" +
      grandTotalActual);
   appendCell(newRow, "grandTotal", getPercentage(grandTotalActual,
      grandTotalFcst) + "%");
}

function appendCell(Trow, Cclass, txt) {
   var newCell = Trow.insertCell(Trow.cells.length);
   newCell.className = Cclass;
   newCell.innerHTML = txt;
}
function deleteRows(tbl) {
   while (tbl.rows.length > 0) {
      tbl.deleteRow(0);
   }
}
```

Many standalone statements at the end of the `drawTextTable()` function are devoted exclusively to generating the Grand Total row, in which the accumulated column totals are entered. At the same time, the `getPercentage()` function, described earlier, is invoked several times again to derive the quota percentage for the accumulated grand total values in each quarter as well as the complete year.

select controls

To round out the code listing for this application, the values assigned to the two `select` elements obviously have a lot to do with the execution of numerous functions in this application. Nothing magic takes place here, but you can see the extent of the detail required in assigning script-meaningful hidden values, and human-meaningful text for both `select` elements. For example, dividing lines help organize the long sort key list into three logical blocks.

```
<p>Sort by: <select id="sortChooser" onchange="selectSort(this)">
   <option value="byRep">Representative</option>
   <option value="byRegion">Sales Region</option>
   <option value="">---------------------</option>
   <option value="byQ1Fcst">Q1 Forecast</option>
   <option value="byQ1Actual">Q1 Actual</option>
   <option value="byQ1Quota">Q1 Performance</option>
   <option value="byQ2Fcst">Q2 Forecast</option>
   <option value="byQ2Actual">Q2 Actual</option>
   <option value="byQ2Quota">Q2 Performance</option>
   <option value="byQ3Fcst">Q3 Forecast</option>
   <option value="byQ3Actual">Q3 Actual</option>
   <option value="byQ3Quota">Q3 Performance</option>
   <option value="byQ4Fcst">Q4 Forecast</option>
   <option value="byQ4Actual">Q4 Actual</option>
   <option value="byQ4Quota">Q4 Performance</option>
   <option value="">---------------------</option>
   <option value="byTotalFcst">Total Forecast</option>
   <option value="byTotalActual">Total Actual</option>
   <option value="byTotalQuota">Total Performance</option>
</select>    Ordered: <select id="orderChooser"
   onchange="selectOrder()">
   <option value="inc">Low to High</option>
   <option value="dec">High to Low</option>
</select></p>
```

Dreams of Other Views

Confining the example to just one type of view — a table of numbers — should help you grasp the important processes taking place. But with the XML data converted to JavaScript objects, you can build many other views of the same data into the same page. For example, a script could completely hide the numeric table, and generate a different one that draws bar charts for each sales representative or each region (see Chapter 55 for a scripted bar chart example). The horizontal axis would be the four quarters, and the vertical axis would be dollars or quota percentages. Clicking a bar opens a small window or layer to reveal more detail from the sales representative's record, such as the name of the person's manager. More `select` elements can let the user select any combination of subsets of the data in either bar chart or numeric table form to facilitate visual comparisons. You might be even more creative and devise ways of showing the data by way of overlapping positioned elements.

The point is that despite the kinds of rendering opportunities afforded by the XSL Transform mechanism (even if you can get comfortable in the syntax and mental model it presents to authors), JavaScript's detailed access to the DOM offers far more potential. Eventually plenty of content authors will mix the two technologies by embedding JavaScript into XSL stylesheets to supplement XSL features.

✦ ✦ ✦

Index

Note: Page numbers preceded by BC refer to Bonus Chapters 36–57 on the CD-ROM.

SYMBOLS

+= (add-by-value) operator, 112–113
+ (addition) operator, 66, 112
& (ampersand), 495
&& (AND) operator, 194
@ (at symbol), 27–28
{ } (curly braces), 76
/ (division) operator, 66, 494
“ (double quotes), 87
// (double slash), 23
= (equal sign), 66
== (equality) operator, 79
(hash) symbol, 622
% (modulus) operator, 71
* (multiplication) operator, 66
\n (newline) character, 161, 692
  (nonbreaking space character), BC22
!= (not equal to) operator, 67
( ) (parentheses), 24
% (percent character), 1033
| (pipe character), 407
+ (plus sign), 24, 66
\r (return character), 692
‘ (single quotes), 87
- (subtraction) operator, 66
_ (underscore), 991

A

a element object properties, 595–599
<a> tag, 137
abbr property, BC57
abort event, 264
abort() method, BC188
above property, BC113
absolute units, 812
AbsolutePosition property, 250
accelerator property, 836
acceptCharset property, 638
access
  access denied warning, 487
  cookies, 522
  errors, BC285
  properties, 43
  server access, reducing, BC386
accessKey property
  as generic object, 204–206
  label object, 1033–1034
action attribute, 99, 108, 638
activeElement property, 513
ActiveX controls
  ActiveXObject object, 1019–1020
  plug-ins, 6
add() method, 710
addBehavior() method
  discussed, 177, BC332
  generic objects, 261–264
  return values, 261
add-by-value (+=) operator, 112–113
AddDesktopComponent() method, 378
addEventListener() method
  discussed, 191
  generic methods, 264
addImport() method, 804–805
adding numbers, 64
addition (+) operator, 66, 112
add-ons, filters, 226
addRange() method, 1065
addReadRequest() method, BC105–BC106
addRow() function, 259
addRule() method, 805
addStyle() function, 558
addTotals() function, BC369
adjustClip() function, BC150
Adobe Acrobat Reader application, 6, 1144–1145
afterBegin parameter, 285
afterEnd parameter, 285

AIFF file format, BC268
alert dialog boxes, 53, 405
alert() method
  functions and, 74
  strings, 112
  window object, 45, 87–88, 404–405
alertUser() function, 53–54
align property
  applet object, BC169
  discussed, 476–477
  embed object, BC178
  hr object, 1030
  Image object, 604–605
  object object, BC173
  table object, BC35
alignment, images, 605
aLink property, 578
alinkColor property
  discussed, 179, 514
  document object, 179
all property, 206–207
allowTransparency property, 463, 477
alpha() method, 838
alt property
  applet object, BC169
  area element object, 623
  Image object, 606
  <meta> tag, 153
  object object, BC173
altHTML property, BC169
altKey property
  IE4+ event object, 755
  NN6+ event object, 775
altLeft property, 756
ampersand (&), 495
anchors
  anchorNode property, 1063
  anchorOffset property, 1063
  anchors property, 516–517
  documents with, 488–489
  Web page navigation, 517
AND (&&) operator, 194
animation effects, 255
anonymous functions, 189
API (Application Programming Interface), BC263, BC324–BC326
appCodeName property, BC69
appCore property, 369
append() function, 267
appendChild() method
  discussed, 139, 185
  generic objects, 267–269
appendData() method, 1070
applets
  applet object, BC167–BC171
  applet-to-script communication, BC257–BC258
  applets property, 517–518
  conditional comments, BC246–BC247
  data type conversions, BC256
  faceless, BC253
  fields, accessing, BC248
  limitations, BC252
  LiveConnect features, BC243–BC244
  reasons for, BC244
  sample design, BC249–BC251
  security, BC300–BC301
  starting and stopping, BC246–BC247
application level objects, BC68
Application Programming Interface (API), BC263, BC324–BC326
applications
  Adobe Acrobat Reader, 6, 1144–1145
  cross-browser, 304
  Decision Helper, 31–32
  HTAs (HTML applications), 177
apply() method, 987
applyElement() method, 269–270
appMinorVersion property, BC76
appName property, 44, 90, BC69
appVersion property, 44, 90, BC69
archives
  archive property
    applet object, BC170
    security, BC307–BC308
  jaws.jar, BC257
area element object
  overview, 621–622
  properties, list of, 623–624

<area> tag, 137
areas property, 625
arguments. *See also* parameters
  arguments property, 984–985
  defined, 44, 74
  dialogArguments property, 374
arithmetic operators, 66
arity property, 985
arrays
  all property, 206–207
  Array object
    array.concat() method, 918–921
    array.join() method, 921–922
    array.pop() method, 923
    array.push() method, 923
    array.reverse() method, 923–925
    array.shift() method, 923
    array.slice() method, 925
    array.sort() method, 925–926
    array.splice() method, 929–930
    array.toLocaleString() method, 930
    array.toString() method, 930
    array.unshift() method, 923
    properties, list of, 916–918
    references, 1108
  concatenation, 920–921
  creating, 77–78, 910
  data, accessing, 78
  defined, 76
  deleting, 912
  dense, 911
  document object in, 80
  element, 80
  forms and, 637
  getSearchAsArray() function, 496
  hash tables and, 916
  imageLibrary, 133
  indexes, 76–77
  items in, accessing, 77
  length property, 77, 143
  literal notation, 911–912
  lookups, 913–914
  multidimensional, 915–916
  objects in, 80
  overview, 909–910
  parallel, 78–80, 912–913
  populating, 911
  properties as, 166
  return values, 926
  spreadsheets and, 76
  two-dimensional, BC356
assign() method, 499
assignment errors, BC284
assignment operators, 61, 967–968
assignment statements, 112
at symbol (@), 27–28
atEnd() method, 1021
attachEvent() method
  discussed, 264
  generic objects, 270
attachToEnd() function, 952
attributes. *See also* properties
  action, 99, 108, 638
  attributes property
    discussed, 183
    as generic object, 207–209
  clearAttributes() method, 275
  content, 151–153
  createAttribute() method, 307
  event, 145, 176
  for, 145, 176
  form attributes, changing, 636
  getAttribute() method, 208–209, 286–287
  getAttributeNodeNS() method, 288–289
  getAttributeNS() method, 289
  hasAttribute() method, 293
  hasAttributeNS() method, 293
  hasAttributes() method, 293–294
  height, <img> tag, 132
  href, 137
  id
    name attribute versus, 40
    naming objects, 39–40
    security issues, B308, BC309
  language, 48, 145
  mergeAttributes() method, 209, 300–301

Continued

attributes *(continued)*
method, 99
name
form object, 99
<form> tag, 91, 97
id attribute versus, 40
object properties, list of, 208
removeAttribute() method, 209, 306–307
removeAttributeNode() method, 307
removeAttributeNS(), 308
setAttribute() method, 187, 207, 209
setAttributeNode() method, 317
setAttributNS() method, 318
src, 47–48
target
form object, 99
window.open() method, 85
type
<input> tag, 100
<script> tag, 47, 145
width, 132
audio and video plug-ins, 6
aural properties, style object, 836–837
authoring environment, setting up
MacOS X setup, 19
reloading issues, 20
save-switch-reload sequence, 18
tools, 17
Windows setup, 18–19
autocomplete property, 639
automation objects, 1019
autoScroll() function, 439
availHeight property, BC100
availTop property, BC101–BC102
availWidth property, BC100
axis property, BC57
azimuth property, 837

B

back-end programs, 5
back() method
discussed, 406
history object, 504
BackColor command, documents, 563
BackCompat property, 519
backgrounds
background property
discussed, 579, 828
layer object, BC116–BC117
table object, BC35
backgroundAttachment property, 828
backgroundColor property, 828
backgroundImage property, 828
backgroundPosition property, 828
backgroundPositionX property, 829
backgroundPositionY property, 829
backgroundRepeat property, 829
layers, setting, BC144–BC146
backreferencing, regular expressions, BC196
bandwidth, 631
banners, 403
base object, BC5–BC6
basefont object, BC6–BC7
BaseHref property, BC173
baseURI property, BC173
BasicImage() method, 844
batch mode validation, BC215
BBEdit script utility, 18
beforeBegin parameter, 285
beforeEnd parameter, 285
behaviors
adding, 261
behavior property
discussed, 836
marquee object, 1035
behaviorCookie property, 756
behaviorPart property, 756
built-in, 261
defined, BC331
draggable element example, BC335–BC337
enabling/disabling, BC332
linking, BC332
namespaces, BC334
resources, BC340
statements and, BC333
text rollover example, BC337–BC340
variable scope, BC333

below property, BC113
beta versions, browsers, 156
bgColor property
  body element object, 578–579
  discussed, 166, 179, 514, 518
  document object, 179
  layer object, BC117–BC118
  marquee object, 1037
  table object, BC36
bgProperties property, 579
bidirectional event model, 190–192
bidirectional text, 820
binary files, 18
binding data, 220–222
bits, defined, 60
bitwise operators, 972–973
blank frames, 365
blank lines between elements, 211
blank space between cells, BC38
blendTrans() method, 840
Blinds() method, 844
blinking text, 34
block-level containers, 242
block-level elements, 179
blockquote object, 1024
blur effects
  blur event, 264
  blur() method
    filter object, 839
    generic objects, 271
    text object, 683
    window object, 272
  onblur event, 326–327
body
  body element object
    aLink property, 578
    background property, 579
    bgColor property, 578–579
    bgProperties property, 579
    bottomMargin property, 580
    createControlRange() method, 583
    createTextRange() method, 583
    doScroll() method, 583–584
    leftMargin property, 580
    link property, 578, 580
    noWrap property, 580–581
    onafterprint property, 583–584
    onbeforeprint property, 584
    onscroll property, 585
    overview, 577
    rightMargin property, 580–581
    scroll property, 581
    scrollLeft property, 581
    scrollTop property, 581–582
    text property, 578, 583
    topMargin property, 580, 583
    vLink property, 583
  body property, 518
  <body> tag, 168
  scripts in, 49–50
bold
  bold command, 1085
  boldface text, 1045
bookmarks property, 756
Boolean operations
  Boolean object, 890
  operators, 968–972
  values, 56
borders
  border property
    discussed, 470
    Image object, 606
    object object, BC174
    style object, 829
    table object, BC36
  borderBottomColor property, 830
  borderBottomStyle property, 830
  borderBottomWidth property, 830
  borderCollapse property, 834
  borderColor property
    discussed, 463, 471
    style object, 830
    table object, BC36
  borderColorDark property, BC36
  borderColorLight property, BC36–BC37
  borderLeft property, 829
  borderLeftColor property, 830

Continued

borders *(continued)*
borderLeftStyle property, 830
borderLeftWidth property, 830
borderRight property, 829
borderRightColor property, 830
borderRightStyle property, 830
borderRightWidth property, 830
borderSpacing property, 834
borderStyle property, 830–831
borderTop property, 829
borderTopColor property, 830
borderTopStyle property, 830
borderTopWidth property, 830
borderWidth property, 831
edge properties and, style object, 829–833
bottom property
discussed, 826
TextRectangle object, 1102
bottomMargin property, 580
boundElements property, 756
bounding rectangles, 289–291
boundingHeight property, 1077
boundingLeft property, 1077
boundingTop property, 1077
boundingWidth property, 1077–1079
box property, BC39
br object, 1024–1025

 tag, 24
branching index pages, 151–153
branching variables, 194
break statement
discussed, 939–940
if construction, 79
browsers
beta versions, 156
browserLanguage property, BC77
compatibility issues
core language standard, 11–12
CSS (cascading style sheets), 13–14
DHTML (Dynamic HTML), 14
discussed, 9–10, 151
document objects, 11
DOM (Document Object Model), 12–13, 18
version numbering, 11–12
conflicts between, 177
crashes, BC295–BC296
cross-browser applications, 304
Firebird, 10
language systems, 235
markers, 10
Mozilla, 10
Netscape, 10
nonscriptable, 50–51, 149–150
Opera, 7, 10
output stream, 92
preferences, reading and writing, BC85–BC86, BC887
Safari, 7–8, 55
selection considerations, 18
sniffer scripts, 151–153
standards, 10
switching between, 19
upgrades, 14–15
version detection, 148–150
bubbles property, 776
buffers
bufferDepth property, BC102
offscreenBuffering property, 386
built-in behaviors, 261
built-in objects, 158
buttons
back, 406
button object
click() method, 655
form property, 653–654
name property, 654
onclick event, 655–656
onmousedown event, 656
onmouseup event, 656
overview, 651–652
type property, 654
value property, 654
button property
IE4+ event object, 757–758
NN6+ event object, 776
in forms, 636
forward, 406
as objects, 102

C

caching
images, 132
loadCached() function, 133
precaching images, 132–133
calculate() method, BC370
calculations
Date object, 117–118
forms, 193
operators, 66
percentages, 240
calculators, graphical, 31
calendars
DHTML table example, BC348–BC352
discussed, BC341–BC342
dynamic tables and, BC345–BC347
static tables and, BC342–BC344
call() method
discussed, BC258
function object, 987–988
caller property, 985–986
calling functions, 73
cancelable property, 776–777
cancelableFlag property, 791
cancelBubble property
IE4+ event object, 758
NN6+ event object, 77
cancelDefault() function, 336
canceling events, 733–734
canHaveChildren property, 209–211
canHaveHTML property, 211
capitalization, text entries, 138
caption property, BC37
captionSide property, 834
captureEvents() method, 406–408, 551–552
carriage returns
inserting, 136
in text areas, 692
Cascading Style Sheets (CSS)
applying and creating, 558–559
browser compatibility issues, 13–14
className property, 138
createStyleSheet() method, 557–559
CSS1 (Cascading Style Sheets Level 1), 13
div element, 172
filters, 176
hyphenated words, 175
overview, 5
phantom page syndrome, 175
settings, changing, 138
span element, 172
case, strings, 113
case-sensitivity
functions, 40
properties, 166
strings, 292
catch construction, 950–952
cautions, dialog boxes, 446–447
cells
blank space between, BC38
cellIndex property, BC58
cellPadding property, BC37
cells property
table object, BC38
tr object, BC54
cellSpacing property, BC37–BC38
content, replacing, BC23–BC25
data in, editing, BC22–BC25
deleting, BC55
inserting, BC55–BC56
populating, BC21–BC23
center value, 446
centered class, 139
CGI. *See* Common Gateway Interface
ch property, BC49
change event, 264
characters
character codes, 344
escaped, 851
extracting, 114
inline, strings and, 851
international, security issues, BC316
keyboard characters, specifying, 204
metacharacters, BC194–BC195
number limitations, 112
positioning, runtime errors, BC280–BC281
special, BC193–BC196
string limitations, 112

characterSet property, 519
charAt() method, 114
charCode property
  discussed, 778–779
  NN6+ event object, 77
CharSet parameter, 220
charset property
  discussed, 518–519
  a element object, 595
  link object, BC10
  meta object, BC13
checkboxes
  input objects
    checked property, 658–659
    click() method, 661
    defaultChecked property, 659
    onclick event, 661–663
    overview, 657
    type property, 659
    value property, 659–660
  as objects, 102–103
checked property
  checkbox input object, 658–659
  discussed, 102
  radio input object, 665–666
Checkerboard() method, 844
checkForEnter() function, 346
checkForm() function, BC241
checkFrameset() function, 497–498
checkNumeric() function, 684
checkTimer() function, 615–616
child nodes
  appendChild() method, 139
  childNodeDetail property, 190
  childNodes property
    discussed, 183–184
    as generic object, 211–213
  child-to-child references, 124, 362
  child-to-parent references, 124, 362
  defined, 42
  discussed, 227
  frames, 122–123
children property, 213–214
chOff property, BC49
chroma() method, 839, 844
chrome
  elements, 368
  windows, controlling, 376–377
circle value, BC63
cite property, 1024
classes
  centered, 139
  defined, BC245
  JSObject, BC256, BC258–BC259
  packages and, BC275
  tables, BC33
classid property, BC174
className property
  cascading style sheets, 138
  as generic object, 215–216
clear() method
  discussed, 552–553
  selection object, 1065
clear property
  br object, 1025
  discussed, 821
clearAttributes() method, 275
clearData() method, 370, 764
clearInterval() method, 408–409
clearRequest() method, BC106
clearTimeout() method, 409–411
click event, 264
click() method
  button object, 655
  checkbox input object, 661
  generic objects, 276
  radio input object, 668
client-side scripting, 8
clientHeight property, 216–217
clientInformation object, BC68
clientInformation property, 370
clientLeft property, 217–218
clientTop property, 217–218
clientWidth property, 216–217
clientX property
  IE4+ event object, 759
  NN6+ event object, 780
clientY property
  IE4+ event object, 759
  NN6+ event object, 780

clip property
discussed, 821
layer object, BC118–BC122
clipboardData property, 370–371
clipBottom property, 821
clipLeft property, 821
clipping layers, BC146–BC147
clipRight property, 821
clipTop property, 821
cloneContents() method, 1046
cloneNode() method
discussed, 185
generic objects, 276–277
cloneRange() method, 1046
close() method
discussed, 553–554
window object, 85–86, 411–412
closed property, 371–372
closeNewWindow() function, 86
closing tags, 47
code examples. *See* listings
code property
applet object, BC170
object object, BC174
codebase principal, security, BC306
codeBase property
applet object, BC170
object object, BC174
codeType property, BC175
col object, BC50–BC52
colgroup object, BC50–BC52
collapse() method
Range object, 1047
selection object, 1066
TextRange object, 1080–1081
collapsed property, 1043
collapseToEnd() method, 1066
collapseToStart() method, 1066
collections
HTML, 300
layers and, 173
color
color property
basefont object, BC7
cascading style sheets, 138
font object, 1026
hr object, 1032
style object, 815
colorDepth property, BC102
layer backgrounds, BC118
of pages, changing, 209–210
of pop-up lists, changing, 712
of text, changing, 248
cols property
discussed, 471
frames, 364
table object, BC38
textarea object, 692
colSpan property, BC58
columns, in tables, editing, BC30–BC32
comma-delimited strings, BC13
command-line systems, events, 719–720
commands. *See also* functions; methods
bold, 1085
copy, 1085
cut, 331–332, BC324
document, 563
italic, 1086
paste, 331–332, BC325
query, 569
underline, 1086
CommandState query command, 569
comments
comment tags, 23
conditional, BC246–BC247
defined, 51
statements, 51
Common Gateway Interface (CGI)
overview, 27
prototyping, 33
serverless, 33
commonAncestorContainer property, 1043–1044
compact property, BC61
compareBoundaryPoints() method, 1047–1048
compareEndPoints() method, 1081–1084
comparePoint() method, 1051
comparison operators, 66–67, 962–963

compatibility issues
browsers
core language standard, 11–12
CSS (cascading style sheets), 13–14
DHTML (Dynamic HTML), 14
discussed, 9–10, 151
document objects, 11
DOM (document object model), 12–13, 18
version numbering, 11–12
DHTML, BC319–BC320
events, 741–742
scripts, 145
Web pages, 155
`compatMode` property, 519–520
competitors, Web sites, 4
`compile()` method, BC207
`complete` property
image `input` object, 670
`Image` object, 606–607
`isindex` object, BC8
`complete` value, 249
`componentFromPoint()` method, 277–280
`Components` property, 369, 372
concatenation
arrays, 920–921
strings, 66, 112
text, 24
condition expressions, 70, 932–933
conditional comments, applets and, BC246–BC247
confirm dialog boxes, 88, 413
`confirm()` method, 412–413
conflicts between browsers, 177
connubial operators, 965–967
console window, JavaScript, 55
`const` keyword, 1017–1018
constants
`fixed`, 579
`MASK`, 752
`constructor` property
`Array` object, 916
`Boolean` object, 890
`Error` object, 958
function object, 986
`Math` object, 887
`RegExp` object, BC206
`string` object, 853–854
containers
block-level, 242
forms as, 98
table-oriented, BC21
containment
hierarchy, 167
inheritance versus, 364
`contains()` method, 279–280
`containsNode()` method, 1066
`content` attribute, 151–153
content-oriented HTML, 5
`content` property
discussed, 821
`meta` object, BC13
`contentDocument` property
discussed, 463–464, 477
frames, 125
`object` object, BC175
`contentEditable` property, 218–219, 233
`contentWindow` property, 366, 464, 477
context-sensitive help, 342–343
control elements, forms, 98, 272–273, 321
control structures
conditions, 70
defined, 70
`if` constructions, 70
`if...else` constructions, 71
`controllers` property, 369, 372
conversions
data types, 64–66, BC256
decimal to hexadecimal, 880–881
numbers to strings, 65–66, 882
strings
case, 113
to numbers, 65, 881–882
cookies
access, 522
`cookie` property, 520–521
`cookieEnabled` property, BC78
data, retrieving, 524

discussed, 496
domain-specific, 521
expiration dates, 521–522
naming, 522
path settings, 523
saving, 522
security issues, 523
soft, BC414
storage, BC379
temporary, BC414
coordinate points, 45
coordinate systems, layers, BC126–BC128
coords property
area element object, 623
discussed, 595
copy command, 1085
copying disabled elements, 225
copyright notifications, 147
Core DOM (Document Object Model), 178
core language objects, 111
countDown() function, 483
countdown timers, 409–410
counterIncrement property, 821
counterReset property, 822
cpuClass property, BC78–BC79
crashes, browsers, BC295–BC296
createAttribute() method, 307, 554
CreateBookmark command, documents, 563
createCaption() method, BC45
createComment() method, 554
createContextualFragment() method, 1051–1052
createControlRange() method, 583
createDocumentFragment() method, 555
createElement() method, 186, 555–556
createEvent() method, 556
createEventObject() method, 557
CreateLink command, documents, 563
createPopup() method, 413
createRange() method
discussed, 234, 557
selection object, 1066
createStyleSheet() method, 557–559
createTextNode() method, 186, 559–560
createTextRange() method
body element object, 583
overview, 234
textarea object, 693
createTFoot() method, BC46
createTHead() method, BC46
createTreeWalker() method, 560
creating
arrays, 77–78, 910
Date object, 893–894
elements, 186
frames, 360
functions, 982–983
image rollovers, 134–137
placeholders, 236
pop-up lists, 413
style sheets, 558–559
variables, 61
windows, 85–86, 425–426
cross-browser applications, 304
cross-confirmation fields, validation, BC239–BC240
crypto property, 373
CSS. *See* Cascading Style Sheets
cssFloat property, 822
cssRule object, 807–810
cssRules property, 799
cssText property
cssRule object, 808
style object, 836
styleSheet object, 799–800
ctrlKey property
IE4+ event object, 755
NN6+ event object, 775
ctrlLeft property, 756
cue property, 837
cueAfter property, 837
cueBefore property, 837
curly braces ({ }), 76
current property, 502
currentNode property, 587
currentStyle property, 219
currentTarget property, 783–785
currTitle variable, frames, 128

cursor property
discussed, 446
style object, 822
custom APIs, BC324–BC326
custom functions
discussed, 982
validation functions, BC221–BC222
custom methods, 999–1000
cut command, 331–332, BC324

D

data
arrays, 78
binding, 220–222
data collections, data lookup, 29
data entry validation, 33
data property
NN4 event object, 748–749
object object, BC175
Text object, 1069
data tainting, BC87
dialog data, retrieving, 445
interactive, 30–31
processing, 107
sorting, 223–224
tab-delimited, 221
text information as, 59
validation, 33, 688
verification, BC296
Data Source Objects (DSOs), 220
data types
conversions, 64–66, BC256
list of, 60
placeholers, 60
undefined, 155
databases
back-end programs, 5
serverless, BC355
dataFld property
discussed, 219
IE4+ event object, 756
dataFormatAs property, 219
dataPageSize property, BC39
dataSrc property, 219–223
dataTransfer property, 763–765
DataURL parameter, 220
Date object
calculations, 117–118
creating, 893–894
form validation, 904–907
getMonth() method, 117
GMT (Greenwich Mean Time), 116, 892
methods, list of, 117, 895–896
nextWeek() function, 118
setMonth() method, 117
snapshots of, 116
strings and, 898
time intervals, adding and subtracting, 118
time zones, 891–892
today variable, 7116
date validation, BC224–BC227
Date.parse() method, 899
dd object, BC64–BC65
Debug menu (Safari), 55
debugging. *See also* errors
browser crashes, BC295–BC296
Evaluator utility, BC291–BC292
expression evaluation, BC289
Microsoft Script Debugger tool, BC290
snapshots, BC292
source code, BC287
statements, BC289
tags, BC286–BC287
timing problems, BC287–BC288
Venkman debugger tool, BC290
decimal to hexadecimal conversion, 880–881
Decision Helper application, 31–32
declaring variables, 61
decodeURI() function, 1010
decodeURIComponent() function, 1010
decoding, strings, 876
defaultCharset property, 529
defaultChecked property
checkbox input object, 659
radio input object, 666
defaultStatus property, 373–374
defaultValue property, 677–678
defaultView property, 529

defer property, BC15
deferred scripts, 52–54, BC345
defineGetter() method, 189
defineSetter() method, 189
delete operator, 973–974
deleteCaption() method, BC45–BC46
deleteCell() method, BC55
deleteContents() method, 1052–1053
deleteData() method, 1070
deleteFromDocument() method, 1066
deleteRow() method, BC46
deleteRule() method, 806
deleteTFoot() method, BC46
deleteTHead() method, BC46
deleting
  arrays, 912
  cells in tables, BC55
  elements, 186
  rows in tables, BC46
demo() function, 75
dense arrays, 911
deprecated properties, BC4
description property
  discussed, 958
  mimeType object, BC88
  plugin object, BC92
designMode property, 530
detach() method, 1053–1054
detachEvent() method, 270–271, 280
detail property, 785
detection, plug-ins, BC96–BC99
dh1.htm file, BC431–BC432
dh2.htm file, BC432–BC434
dh3.htm file, BC434–BC437
dh4.htm file, BC437–BC440
dh5.htm file, BC440–BC443
dhHelp.htm file, BC443–BC444
dhNav.htm file, BC430–BC431
DHTML. *See* Dynamic HTML
dialing operations, 21
dialog boxes
  alert, 53, 405
  cautions, 446–447
  confirm, 88, 413
  displaying, 52
  errors, preventing display of, 54–55
  Internet Explorer, 54
  modeless, 452–454
  prompt, 88
dialog data, retrieving, 445
dialogArguments property, 374
dialogHeight property, 374
dialogLeft property, 375
dialogTop property, 375
dialogWidth property, 374–375
Dictionary object, 1020
digital certificates, security, BC305–BC306
dimensions() method, 1022
dir object, BC65–BC66
dir property, 225
direction property
  discussed, 822
  marquee object, 1037
directive elements, BC3
directories property, 375
disabled elements, selecting and copying, 225
disabled property
  as generic object, 225–226, 234
  link object, BC10
  styleSheet object, 800
disabling behaviors, BC332
disc value, BC63
dispatch lookup tables, BC228–BC229
dispatchEvent() method
  discussed, 271
  generic objects, 280–282
display property, 822–823
div element
  cascading style sheets, 172
  discussed, 456
division (/) operator, 66, 494
dl object, BC64–BC65
doClickEvent() function, 407
doClicks() function, 416
doctype property, 530
DOCTYPE switching, 198–199
document object hierarchy, 164–165

Document Object Model (DOM)
broswer compatibility issues, 12–13, 18
in browser window, 35–36
content, adding/replacing, 188
Core DOM, 178
DOM levels (W3C DOM), 178
form controls, 98
HTML structure and, 34–35
specification levels, 178
window object, 35
Document Type Declaration (DTD), BC4
documents
with anchors, 488–489
commands, list of, 563
document nodes, 42
document object
activeElement property, 513
alinkColor property, 179, 514
anchors property, 516–517
applets property, 517–518
in arrays, 80
bgColor property, 179, 514, 518
body property, 518
captureEvents() method, 551–552
characterSet property, 519
charset property, 518–519
clear() method, 552–553
close() method, 553–554
compatMode property, 519–520
cookie property, 520–521
createAttribute() method, 554
createComment() method, 554
createDocumentFragment() method, 555
createElement() method, 555–556
createEvent() method, 556
createEventObject() method, 557
createRange() method, 557
createStyleSheet() method, 557–559
createTextNode() method, 559–560
createTreeWalker() method, 560
defaultCharset property, 529
defaultView property, 529
designMode property, 530
doctype property, 530
document.createElement() method, 94
document.createTextNode() method, 94
documentElement property, 530–531
document.forms property, 91
document.getElementById() method, 94–95
document.images property, 91
document.write() method, 90, 92–94
domain property, 531
elementFromPoint() method, 560–562
embeds property, 531
execCommand() method, 562–564
expando property, 532
fgColor property, 514, 532
fileCreatedDate property, 532
fileModifiedDate property, 532
fileSize property, 532–534
forms property, 534–536
frames property, 536–537
getElementById() method, 90, 564–565
getElementsByName() method, 565
getSelection() method, 566–567
handleEvent() method, 567
height property, 537
ids property, 537
images property, 537–538
implementation property, 538–539
lastModified property, 539
layers property, 540–542
linkColor property, 179, 514, 542
links property, 542
location object versus, 89
location property, 543
media property, 545
mimeType property, 546
nameProp property, 546
namespaces property, 546
onselectionchange event, 575
onstop event, 575
open() method, 567–568

overview, 36, 512
parentWindow property, 546
plugins property, 546–547
protocol property, 547
queryCommandCommandState() method, 569
queryCommandEnabled() method, 569
queryCommandIndterm() method, 569
queryCommandSupported() method, 569
queryCommandText() method, 569
queryCommandValue() method, 569–570
recalc() method, 570
referrer property, 547–548
releaseEvents() method, 570
roles of, 36
routeEvent() method, 571
scripts property, 548–549
security checks, BC303
security property, 549
selection property, 549–550
styleSheets property, 550
subWrite() method, 93
syntax, 90
tags property, 550
title property, 550
URL property, 543–545, 551
URLUnencoded property, 551
vlinkColor property, 179, 514–516, 551
width property, 537, 551
write() method, 44, 571
writeln() method, 571–575
document property
discussed, 377–378
layer object, BC122
document.createElement() method, 94
document.createTextNode() method, 94
documentElement property, 530–531
document.forms property, 91
document.getElementById() method, 94–95
document.images property, 91
document.write() method, 24, 90, 92–94
.htc, 177
HTML, scripts in, 144–146
doDispatch() function, 280
doDisplay() function, 998
DOM. *See* Document Object Model
DOMActivate event, 264
domain property, 531
domain-specific cookies, 521
DOMAttrModified event, 264
DOMCharacterDataModified event, 264
doMerge() function, 301
DOMFocusIn event, 264
DOMFocusOut event, 264
DOMNodeInserted event, 264
DOMNodeInsertedIntoDocument event, 264
DOMNodeRemoved event, 264
DOMNodeRemovedFromDocument event, 264
DOMSubtreeModified event, 264
done() function, 52–53
doReadRequest() method, BC107–BC108
doScroll() method, 583–584
doSelection() function, BC227
dot notation, properties, 43
dot syntax, 164
double click events, 332–333
double quotes (“), 87
double slash (//), 23
doValidate() method, BC230
do-while loop, 942
drag-and-drop operations, 334–338
dragging layers, BC139–BC140, BC163–BC164
dragIt() function, BC451
drawing images, BC410
drawTextTable() function, BC470
DreamWeaver authoring tool, 17
dropEffect() method, 764
dropShadow() method, 839
DSOs (Data Source Objects), 220
dt object, BC64–BC65
DTD (Document Type Declaration), BC4
duplicate() method, 1084–1085
dynamic font changes, BC7

Dynamic HTML (DHTML)
browser compatibility issues, 14
compatibility issues, BC319–BC320
inline branching, BC320–BC321
innerHTML property, 139
object detection, BC323–BC324
overview, 32, BC317–BC318
platform equivalency, BC321–BC322
stylesheet settings, changing, 138
tables, calendar example, BC348–BC352
dynamic properties, 320–321
dynamic styles, 194
dynamic tables, calendars and, BC345–BC347
dynsrc property
Image object, 608–609
isindex object, BC8

E

ECMA (European Computer Manufacturer's Association), 12
edge and border properties, style object, 829–833
edge value, 446
editors. *See* text editors
effectAllowed() method, 764
element array, 80
element containment hierarchy, 174–175
element referencing, W3C DOM, 180
elements
blank lines between, 211
block-level, 179
createElement() method, 186
deleting, 186
directive, BC3
disabled, selecting and copying, 225
div, 172, 456
elementFromPoint() method, 560–562
elements property, 639–640
floating, 1025
hierarchy, HTML, 35
iframe, 125
inserting into text nodes, 38–39
nested, 209
new, creating, 38
overlapped, 196
reading, 216
span, 138, 172
textarea, forms, 100
elevation property, 837
else keyword, 71
 tag, 38
e-mail, forms, 635–636
embed object, BC177–BC179
embeds property, 531
empty() method, 1067
emptyCells property, 835
Enabled query command, 569
enabledPlugin property, BC88–BC89
enablePrivilege() method, BC310
enabling behaviors, BC332
encodeURI() function, 1010
encodeURIComponent() function, 1010
encodeURLComponent() function, 1010
encoding
encoding property, 641
naming conventions, BC10
strings, 876
enctype property, 99, 641
endContainer property, 1044
endOffset property, 1044
engage() function, BC336
Enter/Return key, text boxes and, 676–677
entries, same word, 43
Enumerator object, 1021
equality
comparisons, operators, 963–964
equal sign (=), 66
equality (==) operator, 79
errors, BC285
errors. *See also* debugging
access, BC285
assignment, BC284
character positioning, BC280–BC281
dialog boxes, preventing display of, 54–55
equality, BC285
error event, 264
Error object
methods, list of, 959
overview, 957
properties, 958–959

exception, BC286
filename, BC279
function, BC282
line number reporting, BC280–BC281
message notification, BC278–BC279
methods, BC282
null, BC284
numbers, BC282
object expected, BC281
property, BC282
runtime
exception handling, 948–949
syntax errors versus, BC277
script errors, controlling, 387–388
simple script example, 24
trapping techniques, 160
undefined, BC281–BC282
viewing, 54–55
warnings and, BC286
escape() function
discussed, 493
global functions, 1011
EscapeChar parameter, 220
escaped characters, 851
European Computer Manufacturer's Association (ECMA), 12
eval() function
discussed, 207, 414
global functions, 1012
Evaluator debugger utility, BC291–BC292
event property
discussed, 378
script object, BC15
<script> tag, 145, 176
eventPhase property, 785
events
abort, 264
addEventListener() method, 191
attachEvent() method, 264, 270
bidirectional event model, 190–192
binding, 176
blur, 264
bubbling, 176
canceling, 733–734
captureEvents() method, 406–408, 551–552
change, 264
click, 408
command-line systems, 719–720
compatibility, 741–742
createEvent() method, 556
createEventObject() method, 557
default actions, canceling, 191
defined, 45
detachEvent() method, 270–271, 280
dispatchEvent() method, 271, 280–282
doClickEvent() function, 407
DOMActivate, 264
DOMAttrModified, 264
DOMCharacterDataModified, 264
DOMFocusIn, 264
DOMFocusOut, 264
DOMNodeInserted, 264
DOMNodeInsertedIntoDocument, 264
DOMNodeRemoved, 264
DOMNodeRemovedFromDocument, 264
DOMSubtreeModified, 264
done() function, 52–53
double click, 332–333
error, 264
event capture, 172, 721–724
event handlers, 45, 168–169
event listeners, 191
examples of, 45
fireEvent() method, 271, 282–285
handleEvent() method, 415–416, 567
IE4+ event object
altKey property, 755
altLeft property, 756
behaviorCookie property, 756
behaviorPart property, 756
bookmarks property, 756
boundElements property, 756
button property, 757–758
cancelBubble property, 758
clientX property, 759
clientY property, 759

Continued

events, IE4+ event object *(continued)*
ctrlKey property, 755
ctrlLeft property, 756
dataFld property, 756
dataTransfer property, 763–765
fromElement property, 765
keyCode property, 767–769
nextPage property, 769
offsetX property, 759
offsetY property, 759
propertyName property, 769
qualifier property, 756
reason property, 756
recordset property, 756
repeat property, 769
returnValue property, 770
saveType property, 770
screenX property, 759
screenY property, 759
shiftKey property, 755–756
shiftLeft property, 756
srcElement property, 770–772
srcFilter property, 772
srcUrn property, 772
toElement property, 765–767, 772
type property, 772–773
wheelData property, 773
x property, 759
y property, 759–763
KeyEvents, 556
load, 264
menu-driven systems, 719–720
as methods, 168
modifier keys, 743–744
mousedown, 146, 264
MouseEvents, 556
mousemove, 264
mouseout, 264
mouseover, 264
mouseup, 264
MutationEvents, 556
NN4 event object
data property, 748–749
layerX property, 749
layerY property, 749
modifiers property, 752
pageX property, 749
pageY property, 749
screenX property, 749
screenY property, 749–751
target property, 752
type property, 753
which property, 753
NN6+ event object
altKey property, 775
bubbles property, 776
button property, 776
cancelable property, 776–777
cancelBubble property, 77
charCode property, 77
clientX property, 780
clientY property, 780
ctrlKey property, 775
currentTarget property, 783–785
detail property, 785
eventPhase property, 785
isChar property, 785
keyCode property, 777–779
layerX property, 780
layerY property, 780
metaKey property, 775
originalTarget property, 785–786
pageX property, 780
pageY property, 780
relatedTarget property, 786–787
screenX property, 780
screenY property, 780
shiftKey property, 775–776
target property, 787–789
timeStamp property, 789–790
objects and, 168–169
onabort, 619
onactivate, 323
onafterprint
body element object, 584
discussed, 456
onafterupdate, 685
onbeforecopy, 323–324

onbeforecut, 324–325
onbeforedeactivate, 323, 325
onbeforeeditfocus, 325
onbeforepaste, 325–326
onbeforeprint
  body element object, 584
  discussed, 456
onbeforeunload, 456–457
onbeforeupdate, 685
onblur
  discussed, 326–327
  layer object, BC143
  text object, 686
onbounce, 1039
onchange
  select object, 711–712
  text object, 688
  this keyword, 106
onclick
  button object, 655–656
  checkbox input object, 661–663
  discussed, 168, 329
  generic objects, 327–329
  radio input object, 668
oncontextmenu, 330
oncontrolselect, 330
oncopy, 330
oncut, 330–331
ondblclick, 329, 332–333
ondeactivate, 323
ondrag, 333–337
ondragdrop, 457–458
ondragend, 333–337
ondragenter, 338
ondragleave, 338
ondragover, 338–339
ondragstart, 333–337, 339
ondrop, 339–340
onfilterchange, 340–341
onfinish, 1039
onfocus
  discussed, 341–342
  layer object, BC143
  text object, 686
onhelp, 342–343, 458–459
onkeydown, 343
onkeypress, 343
onkeyup, 343–344
onload
  <body> tag, 168
  frames, 125
  Image object, 619–620
  layer object, BC143
  link object, BC12
  running scripts from, 53
  window object, 52, 89
onlosecapture, 348–349
onmousedown
  button object, 656
  discussed, 349–350
onmouseenter, 351
onmouseleave, 351
onmousemove, 351
onmouseout
  discussed, 153, 351
  layer object, BC143
onmouseover
  discussed, 153, 351
  layer object, BC143
onmouseup
  button object, 656
  discussed, 349–351
onmove, 460
onpaste, 353–355
onpropertychange, 355–356
onreadystatechange, 356–357
onreset, 645–646
onresize, 357, 460–461
onresizeend, 357
onresizestart, 357
onscroll, 585
onselect, 686–687
onselectionchange, 575
onselectstart, 357–358
onstart, 1039
onstop, 575

Continued

events *(continued)*
onsubmit
form object, 646
<form> tag, 102
onunload, 376, 461
passing, 725
propagation models, 190
as properties, 168–169
redirecting, 728–730
referencing, 720
releaseEvent() method, 429–430
removeEventListener() method, 191, 309
reset, 264
resize, 264
response to, 45
routeEvent() method, 432–433
scroll, 264
select, 264
static, 721
submit, 264
UIEvents, 556
unload, 264
window object and, 176
examples. *See* listings
exception handling
overview, 949–950
runtime errors, 948–949
throwing exceptions, 953–957
exec() method, BC208
execCommand() method
discussed, 562–564
TextRange object, 1085–1087
execScript() method, 414
executing scripts, 51
expand() method, 1087–1088
expandEntityReference property, 587
expando property, 532
expiration dates, cookies, 521–522
exporting signed scripts, BC315
expression evaluation
debugging, BC289
testing, 63
expression() method, 570
expressions
condition, 932–933
as conditions, 70
initial, 72
regular
backreferencing, BC196
discussed, 861–862
grouping, BC196
matching techniques, BC200–BC203
object relationships, BC196–BC200
patterns and, BC191–BC192
search-and-replace operations, BC203–BC205
special characters, BC193–BC196
string replacement, BC203–BC205
testing, 63
update, 72
variables and, 64
extend() method, 1067
extendRow() method, BC367
eXtensible Markup Language (XML), 5
external property, 378–379
extractContents() method, 1054

F

face property
basefont object, BC7
font object, 1027–1028
faceless applets, BC253
factorial() function, 993
Fade() method, 844
false value
checked property, 102
comparison operators, 66
conditions, 70
discussed, 56
else keyword, 71
if constructions, 70
FAQs (Frequently Asked Questions), 1140
fetchFile() function, BC308
fetchText() method, BC254
fgColor property, 514, 532
FieldDelim parameter, 220
fields
applet fields, accessing, BC248
text fields, selecting, 684–685
fieldset element object, 646–647
file object, 717–718

fileAccess() function, BC315
fileCreatedDate property
  discussed, 532
  Image object, 609
fileModifiedDate property
  discussed, 532
  Image object, 609
filename errors, BC279
fileName property, 958
filename property, BC92
files
  binary, 18
  dh1.htm, BC431–BC432
  dh2.htm, BC432–BC434
  dh3.htm, BC434–BC437
  dh4.htm, BC437–BC440
  dh5.htm, BC440–BC443
  dhHelp.htm, BC443–BC444
  dhNav.htm, BC430–BC431
  index.htm, BC426–BC430
  mailto, 495
  Winhelp, 444
fileSize property
  discussed, 532–534
  Image object, 609–610
fileUpdatedDate property, 609
filters
  add-ons, 226
  cascading style sheets, 176
  filter object
    filter names, 843–846
    overview, 837–838
    reveal transitions, 841–842
    static filter types, 838–839
    subproperties, reading and writing, 838
    syntax changes, 842–843
    transition filters, 840
  filter property
    discussed, 587
    style object, 823
  isempty() function, BC216–BC217
  isInteger() function, BC218–BC219
  isNumber() function, BC220–BC221
  isPosInteger() function, BC217–BC218
finally construction, 950–952
find() method, 414–415
findText() method, 1088–1092
Firebird browsers, 10
fireEvent() method
  discussed, 271
  generic objects, 282–285
firstChild() method, 588
firstChild property
  discussed, 183–184
  as generic object, 226–228
firstPage() method, BC47
fixed constant, 579
Flash plug-ins, 6
flipH() method, 839
flipV() method, 839
floating elements, 1025
floating-point numbers
  discussed, 877–879
  integers versus, 65
focus() method
  discussed, 168
  generic objects, 271, 285
  text object, 683–684
focusNode property, 1063
focusOffset property, 1063
fonts
  dynamic font changes, BC7
  font object, 1025–1028
  font property, 815
  FontColor command, documents, 563
  fontFamily property, 815
  FontName command, documents, 563
  FontSize command, documents, 563
  fontSize property, 816
  fontSizeAdjust property, 816
  fontSmoothingEnabled property, BC103
  fontStretch property, 816
  fontStyle property
    discussed, 219
    style object, 816
  fontVariant property, 816
  fontWeight property, 816
  monospace, 1028
  properties, changing, 1026–1027
  sans serif, 1028
  serif, 1028

footers, tables, BC20
for attribute, 145, 176
for loop, 72, 936–937
formatting methods, strings, 874–876
forms
  arrays and, 637
  attributes, changing, 636
  buttons in, 636
  calculations, 193
  as containers, 98
  control elements, 98, 272–273, 321
  data and elements, passing to functions, 106–108
  e-mailing, 635–636
  form object
    acceptCharset property, 638
    action attribute, 99, 108
    action property, 638
    autocomplete property, 639
    elements property, 639–640
    encoding property, 641
    enctype attribute, 99
    enctype property, 641
    form.elements property, 99
    handleEvent() method, 643
    length property, 641
    method attribute, 99
    method property, 642
    name attribute, 99
    name property, 642
    onreset event, 645–646
    onsubmit event, 646
    processData() function, 107
    reset() method, 643–644
    security checks, BC303
    submit() method, 108–109, 644
    target attribute, 99
    target property, 642–643
  form property
    button object, 653–654
    label object, 1034
    object object, BC175
    text object, 678
  <form> tag
    name attribute, 91, 97
    onsubmit event, 102
    form.elements property, 99
    forms property, 534–536
    getFormData() method, 448
    <input> tag, 637–638
    layers and, BC112–BC113
    locking, 225
    passing to functions, 631–634
    simple example of, 535–536
    submission, redirection after, 636–637
    submitting, 108–109
    textarea element, 100
    validation, 30, 108–109
    validation, Date object, 904–907
forward() method
  discussed, 406
  history object, 504–506
  window object, 415
foundArray variable, BC198
foundMatch variable, BC362
frame property, BC39–BC41
frameBorder property, 464–465, 474
frameElement property, 379
frames
  blank, 365
  child, 122–123
  child-to-child references, 124, 362
  child-to-parent references, 124, 362
  cols property, 364
  contentDocument property, 125
  creating, 360
  currTitle variable, 128
  discussed, 359
  document relationship, 360–361
  frame element objects versus, 365–366
  frame object
    allowTransparency property, 463
    borderColor property, 463
    contentDocument property, 463–464
    contentWindow property, 464
    frameBorder property, 464–465
    height property, 465
    longDesc property, 465
    marginHeight property, 466
    marginWidth property, 466
    name property, 466
    noResize property, 466

scrolling property, 467
src property, 468
width property, 465
frameless, switching between, 364
frameset object
border property, 470
borderColor property, 471
cols property, 471
frameBorder property, 474
frameSpacing property, 474
rows property, 471–473
framesets
discussed, 122
forcing to load, 363
goNext() function, 128
iframe element, 125
iframe object
align property, 476–477
allowTransparency property, 477
contentDocument property, 477
contentWindow property, 477
height property, 478
hspace property, 478
longDesc property, 478
marginHeight property, 478
marginWidth property, 478
name property, 479
scrolling property, 479
src property, 479–480
vspace property, 478
width property, 478
inheritance versus containment, 364
loading, 363
loading time, 125
location.hash property, 128
location.href property, 128
multiple
controlling, navigation bars, 126–128
discussed, 31–32
object model, 360–361
onload event, 125
parent window, 122–123
parent-to-child references, 123–124, 362
prevention, 362–363
printFrames() function, 426
referencing, 362
rows property, 364
scripting techniques, 125
single-frame window example, 122
source code, viewing, 365
synchronization, 364–365
top object, 123
two-frame window example, 122
frames property, 379–381, 536–537
frameSpacing property, 474
Frequently Asked Questions (FAQs), 1140
from property, BC8
fromElement property, 765
front end data collection lookup, 29
FrontPage authoring tool, 17
fullName() function, 104
fullScreen() function, 167
function object
apply() method, 987
arguments property, 984–985
arity property, 985
call() method, 987–988
caller property, 985–986
constructor property, 986
length property, 986–987
prototype property, 987
toString() method, 988
valueOf() method, 988
functions. *See also* commands; methods
addRow(), 259
addStyle(), 558
addTotals(), BC369
adjustClip(), BC150
alertUser(), 53–54
anonymous, 189
append(), 267
attachToEnd(), 952
autoScroll(), 439
calling, 73
cancelDefault(), 336
case-sensitivity, 40
checkForEnter(), 346
checkForm(), BC241
checkFrameset(), 497–498

Continued

functions *(continued)*
checkNumeric(), 684
checkTimer(), 615–616
closeNewWindow(), 86
countDown(), 483
creating, 982–983
custom, 982
custom validation functions, BC221–BC222
decodeURI(), 1010
decodeURIComponent(), 1010
defined, 72
demo(), 75
doClickEvent(), 407
doClicks(), 416
document.write(), 24
doDispatch(), 280
doDisplay(), 998
doMerge(), 301
done(), 52–53
doSelection(), BC227
dragIt(), BC451
drawTextTable(), BC470
encodeURI(), 1010
encodeURLComponent(), 1010
engage(), BC336
errors, BC282
escape(), 493, 1011
eval(), 207, 414, 1012
factorial(), 993
fetchFile(), BC308
fileAccess(), BC315
form data and elements, 106–108
fullName(), 104
fullScreen(), 167
function references, 271
generalizable, 993–994
getColor(), 391
getCountDown(), 902
getElementById(), 40, 90
getErrorObject(), 955
getFormData(), 448
getGIF(), BC380
getGrossOffsetLeft(), BC152
getIEVersion(), 194
getNetOffsetTop(), BC152
getObject(), BC325
getParentLayer(), BC152
getSearchAsArray(), 496
getTimeUntil(), 902
goNext(), 128
handleApply(), 452
handleDrop(), 336
handleOK(), 448
highlight(), 304
hilite(), 290
imageOff(), 135
imageOn(), 135
init(), 448
initAudioAPI(), BC266
initExpand(), BC400
initializeCookies(), BC427
inspect(), 709
invoking, 52, 989
isDate(), BC235
isDone(), BC452
isempty(), BC216–BC217
isFinite(), 1012
isInteger() function, BC218–BC219
isNaN(), 1013
isNumber(), BC220–BC221
isPosInteger(), BC217–BC218
isRange(), BC221
isValid(), BC361
isWindows(), BC73
loadCached(), 133, 615
loadIndividual(), 615
loadXMLDoc(), BC397
makeAreas(), 625
makeHTML(), BC398–BC399
makeNewWindow(), 86
makeTitleRow(), BC370
naming, 73
nested, 983–984
newWindow(), 128
nextField(), 221
nextWeek(), 118
Number(), 1013
onClick event, 73

onTarget(), BC453
parameters
  overview, 984
  passing, 438–439
  variables, 992
parseFloat()
  converting strings to numbers, 65
  discussed, 428
  global functions, 1013
parseInt()
  converting strings to numbers, 65
  discussed, 375, 428
  global functions, 1013–1014
passing
  forms to, 631–634
  text to, 682
prevField(), 221
printEvaluator(), BC291–BC292
printFrames(), 426
processData(), 107
recursion in, 992
release(), BC452
removeMember(), BC258
resetField(), 677
resetTab(), 257
restore(), 310, 376
revealClip(), BC147
saveCurrentVisit(), BC418
scripts and, 52
scrollMsg(), 403
searchOnReturn(), BC362
selectChunk(), 247
selectSort(), BC467
setBGColor(), BC328
setClip(), BC147
setCookie(), BC427
setCookieData(), BC417
setCount(), 700
setDocCapture(), 724
setFontAttr(), 1026
setImagePosition(), 242
setInitialColor(), 262
setInnerPage(), BC152
setLang(), 700
setMsg(), 136
setOuterPage(), BC152
setSelected(), 448
setupDrag(), 335–336
setWinWidth(), BC456
setZIndex(), BC328
shiftBy(), BC328
shiftTo(), BC327
showCountDown(), 410
showOffsets(), 393
showValues(), BC119
startTime(), 410
startTimer(), 410
stripZeros(), BC360
syntax, 73
testValues(), 990–991
timeIt(), 336
toggle(), BC380
toggleBar(), 376
toggleColor() BC261
toggleComplete(), 223
toggleDetails(), 374
toString(), 1014–1016
toUpperCase(), 101
trace(), BC293–BC295
trigonometric, 115
turnOn(), 262
unescape(), 493, 1011–1012
unhighlight(), 304
unwatch(), 1016
updateTime(), 443
validation functions, combining, BC222
validDate(), BC224
var keyword and, 74
variable scope, 989–992
verifyData(), BC296
walkChildNodes, 212
watch(), 1016

G

generalizable functions, 993–994
getAdjacentText() method, 285–286
getAllResponseHeaders() method, BC188

getAttribute() method
  generic objects, 286–287
  userProfile object, BC108
getAttributeNode() method, 287–288
getAttributeNodeNS() method, 288–289
getAttributeNS() method, 289
getAttributes() method, 208–209
getBookmark() method, 1092
getBoundingClientRect() method, 289–291
getClientRects() method, 292
getColor() function, 391
getCountDown() function, 902
getData() method, 370
getDate() method, 117
getDay() method, 117
getElementById() function
  document object, 90
  objects, referencing, 40
getElementById() method, 564–565
getElementsByName() method, 565
getElementsByTagName() method
  discussed, 238
  generic objects, 292
getElementsByTagNameNS() method, 292–293
getErrorObject() function, 955
getExpression() method, 293
getFile() method, BC254
getFormData() function, 448
getFullYear() method, 117, 897
getGIF() function, BC380
getGrossOffsetLeft() function, BC152
getHours() method, 117
getIEVersion() function, 194
getItem() method, 1022
getMember() method, BC258
getMinutes() method, 117
getMonth() method, 117
getNetOffsetTop() function, BC152
getObject() function, BC325
getParentLayer() function, BC152
getRangeAt() method, 1067
getResponseHeader() method, BC188
getSearchAsArray() function, 496
getSeconds() method, 117
getSelection() method, 566–567
getSlot() method, BC258
getTime() method, 117
getTimeUntil() function, 902
getWindow() method, BC258
getYear() method, 117
global functions
  decodeURI(), 1010
  decodeURIComponent(), 1010
  encodeURI(), 1010
  encodeURLComponent(), 1010
  escape(), 1011
  eval(), 1012
  isFinite(), 1012
  isNaN(), 1013
  Number(), 1013
  parseFloat(), 1013
  parseInt(), 1013–1014
  toString(), 1014–1016
  unescape(), 1011–1012
  unwatch(), 1016
  watch(), 1016
global property, BC206
global variables, 74–75, 989
Globally Unique Identifier (GUID), BC96
glow() method, 839
GMT (Greenwich Mean Time), 892
go() method, 506–507
goNext() function, 128
gray() method, 839
Greenwich Mean Time (GMT), 116, 892
grouping, regular expressions, BC196
GUID (Globally Unique Identifier), BC96

H

h1 object, 1029
h2 object, 1029
h3 object, 1029
h4 object, 1029
h5 object, 1029
h6 object, 1029
handleApply() function, 452
handleDrop() function, 336

handleEvent() method
discussed, 415–416, 567
form object, 643
handleOK() function, 448
hasAttribute() method, 293
hasAttributeNS() method, 293
hasAttributes() method, 293–294
hasChildNodes() method
discussed, 186
generic objects, 294–295
hasFeature() method, 538
hash property
a element object, 595
area element object, 624
discussed, 488–489
hash symbol (#), 622
hash tables, arrays and, 916
hasOwnProperty() method, 1007
head object, BC4–BC5
headers property, BC57
headers, tables, BC20
headings, scripts in, 48
height
of text, determining, 240
of windows, setting, 383–384
height attribute, 132
height property
applet object, BC169
discussed, 465, 478, 537
embed object, BC178
as generic object, 228
Image object, 610
isindex object, BC8
marquee object, 1037
object object, BC175
screen object, BC100
style object, 826
table object, BC41
td object, BC59
tr object, BC54–BC55
help options, context-sensitive, 342–343
helpers, plug-ins and, 5–6
hexadecimal values
converting to decimal values, 880–881
strings and, 876
hidden object, 689–690
hidden property, BC178
hidden text, 150
hide() method, 482
hideFocus property, 229
hiding
pop-up lists, 483–484
scripts, 50–51, 146–147
statements, 146
hierarchy
containment, 167
element containment, 174–175
inheritance, 167
of nodes, 180–182
object model, 163
table structure, BC19–BC21
high level risks, privilege targets, BC311–BC312
highlight() function, 304
highlighting text, 1040
hilite() function, 290
history object
back() method, 504
current property, 502
forward() method, 504–506
go() method, 506–507
length property, 503–504
next property, 502
overview, 36, 502
previous property, 502–503
roles of, 36
history property, 382
hit counts, Web traffic, 4
home() method, 416
host environments, JavaScript as, 158
host property
a element object, 595
area element object, 624
discussed, 489–492
hostname property
a element object, 595
area element object, 624
discussed, 492
hours, date functions, 117
hr object, 1029–1033

href attribute, 137
href property
  a element object, 595–596
  area element object, 624
  base object, BC6
  discussed, 89, 493–494
  Image object, 610
  link object, BC10
  location object, 89
  styleSheet object, 800
hreflang property
  discussed, 596
  link object, BC10
hsides property, BC39
hspace property
  applet object, BC170
  discussed, 478
  Image object, 610
  isindex object, BC8
  marquee object, 1037
  object object, BC175
HTAs (HTML applications), 177
.htc documents, 177
HTML. *See* Hypertext Markup Language
HTML object, BC3–BC4
htmlFor property
  label element object, 649
  label object, 1034
  script object, BC15
HTMLParagraphElement object, 189
htmlText property, 1079–1080
httpEquiv property, BC14
Hypertext Markup Language (HTML)
  collections, 300
  content-oriented, 5
  defined, 4
  documents, scripts in, 144–146
  DOM (document object model) and, 34–35
  element hierarchy, 35
  HTAs (HTML applications), 177
  new features, 179–180
  node trees, 41
  node types, 182
  tags, defined, 4
hyphenated words, cascading style sheets, 175

I

i variable, 79
ibound() method, 1022
id attribute
  name attribute versus, 40
  naming objects, 39–40
  security issues, B308, BC309
id property
  as generic object, 230
  styleSheet object, 800
IDs (identifiers), 40
ids property, 537
IE. *See* Internet Explorer
IE4+ event object
  altKey property, 755
  altLeft property, 756
  behaviorCookie property, 756
  behaviorPart property, 756
  bookmarks property, 756
  boundElements property, 756
  button property, 757–758
  cancelBubble property, 758
  clientX property, 759
  clientY property, 759
  ctrlKey property, 755
  ctrlLeft property, 756
  dataFld property, 756
  dataTransfer property, 763–765
  fromElement property, 765
  keyCode property, 767–769
  nextPage property, 769
  offsetX property, 759
  offsetY property, 759
  propertyName property, 769
  qualifier property, 756
  reason property, 756
  recordset property, 756
  repeat property, 769
  returnValue property, 770
  saveType property, 770
  screenX property, 759
  screenY property, 759
  shiftKey property, 755–756
  shiftLeft property, 756
  srcElement property, 770–772

srcFilter property, 772
srcUrn property, 772
toElement property, 765–767, 772
type property, 772–773
wheelData property, 773
x property, 759
y property, 759–763
if construction
break statement, 79
control structures, 70
if...else construction, 71
iframe element, 125
iframe object
align property, 476–477
allowTransparency property, 477
contentDocument property, 477
contentWindow property, 477
height property, 478
hspace property, 478
longDesc property, 478
marginHeight property, 478
marginWidth property, 478
name property, 479
scrolling property, 479
src property, 479–480
vspace property, 478
width property, 478
ignoreCase property, BC206
image maps, client-side example, 622–623
images
alignment, 605
caching, 132
document.images property, 91
drawing, BC410
Image object
align property, 604–605
alt property, 606
border property, 606
complete property, 606–607
dynsrc property, 608–609
fileCreatedDate property, 609
fileModifiedDate property, 609
fileSize property, 609–610
fileUpdatedDate property, 609
height property, 610
href property, 610
hspace property, 610
isMap property, 611
longDesc property, 611
loop property, 611
lowsrc property, 612
mimeType property, 612
name property, 612–613
nameProp property, 613
naturalHeight property, 613
naturalWidth property, 613–614
onabort event, 619
onerror property, 619
onload event, 619–620
overview, 602–603
protocol property, 614
src property, 614
start property, 618
useMap property, 618
vspace property, 610–611, 618
width property, 610, 618
x property, 618
y property, 618–619
image rollovers, creating, 134–137
imageLibrary array, 133
imageOff() function, 135
imageOn() function, 135
images property, 537–538
input objects, 669–671
interchangeable, 131–132
motion and still, switching between, 608–609
as objects, 153
precaching, 132–133
rotating, 616–617
scrolling, 434
spacer, 5
swapping, 134
<img> tag
height attribute, 132
src property, 132
width attribute, 132
immediate scripts, BC345
immediate statements, 52
implementation property, 538–539

importing signed scripts, BC315
imports property, 801
in operator, 974
Indent command, documents, 563
indeterminate property, BC8
index values, strings, 115
indexes
  arrays, 76–77
  branching index pages, 151–153
  tabindex property, 256–257
  tabIndex value, 230
  zero-based values, BC55
  zIndex property, 175
index.htm file, BC426–BC430
indexOf() method, 114
in-document scripts, 147
Indterm query command, 569
inheritance hierarchy, 167
inheritance versus containment, frames, 364
init() function, 320, 448
initAudioAPI() function, BC266
initExpand() function, BC400
initial expressions, 72
initializeCookies() function, BC427
initializing variables, 61
inline branching, DHTML, BC320–BC321
inline characters, strings and, 850–851
inline content, 13
inline display and layout properties, style object, 821–825
inline quotation, 1024
innerHeight property, 382
innerHTML property
  Dynamic HTML, 139
  as generic object, 230–232
  read/write access, 174, 189
innerText property
  as generic object, 230–232
  read/write access, 174
innerWidth property, 382
input objects
  checkboxes
    checked property, 658–659
    click() method, 661
    defaultChecked property, 659
    onclick event, 661–663
    overview, 657
    type property, 659
    value property, 659–660
  images, 669–671
  radio objects
    checked property, 665–666
    click() method, 668
    defaultChecked property, 666
    length property, 667
    name property, 667
    onclick event, 668
    overview, 663–664
    type property, 667
    value property, 667
input/output (I/O), 572
input property, BC210
<input> tag
  forms, 637–638
  type attribute, 100
inRange() method, 1093
insertAdjacentElement() method
  discussed, 227
  generic objects, 295–296
insertAdjacentHTML() method, 296
insertAdjacentText() method, 296–298
insertBefore() method
  discussed, 186
  generic objects, 298–299
insertCell() method, BC55–BC56
insertData() method, 1070
insertion pointers, 272
insertNode() method, 1054–1055
insertRow() method, BC46–BC47
insertRule() method, 806–807
inspect() function, 709
installation, plug-ins, BC96
instanceof operator, 974–975
integers, floating-point numbers versus, 65
interactive data, 30–31
interactive user interfaces, 28
interactive value, 249
interCap format, variable names, 61

interchangeable images, 131–132
interfaces, tree views, 28
international characters, security, BC316
Internet bandwidth, 631
Internet Explorer (IE)
  dialog box, 54
  `getIEVersion()` function, 194
  IE4 object model extensions, 174–176
  IE5 object model extensions, 177
  version numbering, 11–12
Internet service providers (ISP), 21
`intersectsNode()` method, 1056
`invert()` method, 839
invoking
  functions, 52, 989
  methods, 44, BC267
I/O (input/output), 572
`iris()` method, 845
`isChar` property, 785
`isCollapsed` property, 1063
`isContentEditable` property
  discussed, 219
  as generic object, 232–233
`isDate()` function, BC235
`isDisabled` property, 233–234
`isDone()` function, BC452
`isempty()` function, BC216–BC217
`isEqual()` method, 1093–1094
`isFinite()` function, 1012
`isindex` object, BC7–BC8
`isInteger()` function, BC218–BC219
`isMap` property, 611
`isMultiLine` property, 234
`isNaN()` function, 1013
`isNumber()` function, BC220–BC221
`isOpen` property, 481–482
ISP (Internet service providers), 21
`isPointInRange()` method, 1056
`isPosInteger()` function, BC217–BC218
`isPrototypeOf()` method, 1007
`isRange()` function, BC221
`isSupported()` method
  discussed, 186
  generic objects, 299
`isTextEdit` property, 234
`isValid()` function, BC361
`isWindows()` function, BC73
`italic` command, 1086
italic text, 219
`item()` method
  discussed, 207
  `Enumerator` object, 1021
  generic objects, 300
  `select` object, 711

J

`JAR` Packager, security, BC307
Java applets. *See* applets
Java in a Nutshell (O'Reilly & Associates, Inc.), BC275
`javaEnabled()` method, BC84
JavaScript language
  advantages, 8
  console window, 55
  disadvantages, 8
  ECMA compatible, 59
  history of, 6–7
  as host environment, 158
  language essentials, 158–161
  overview, 3
  version implementation, 143–144
  when to use, 33–34
JavaScript Scrambler shareware program, 147
`jaws.jar` archive, BC257
joining
  strings, 112–113, 850
  text, 24
`.js` libraries, 197
`JSObject` class, BC256, BC258–BC259
jukebox plug-in example, BC268–BC271
`JustifyCenter` command, documents, 563
`JustifyFull` command, documents, 563
`JustifyLeft` command, documents, 563
`JustifyRight` command, documents, 563
JVM (Java virtual machine), BC244

K

key codes, 344
keyboard characters, specifying, 204

keyboard events. *See* events
keyboard shortcuts, 19
keyCode property
  IE4+ event object, 767–769
  NN6+ event object, 777–779
KeyEvents event, 556
keywords
  const, 1017–1018
  else, 71
  new, 77
  public, BC248
  this, 106
  var
    discussed, 1018–1019
    functions and, 74
    local variables, 74
    variables, creating, 61
  void, BC245

L

label element object, 646–649
label object, 1033–1034
label property
  optgroup object, 715–716
  option object, 714
labeled statements, 944–946
lang property, 235
language attribute, <script> tag, 48, 145
Language parameter, 220
language property
  discussed, 235
  navigator object, BC79
language systems, browsers, 235
lastChild() method, 588
lastChild property
  discussed, 183–184
  as generic object, 226–228, 235
lastIndex property, BC207
lastMatch property, BC210
lastModified property, 539
lastPage() method, BC47
layers
  backgrounds
    colors, BC118
    setting, BC144–BC146
  clipping, BC146–BC147
  collections and, 173
  coordinate systems, BC126–BC128
  defined, 154, BC109
  dragging, BC139–BC140, BC163–BC164
  forms and, BC112–BC113
  layer object
    above property, BC113
    background property, BC116–BC117
    below property, BC113
    bgColor property, BC117–BC118
    clip property, BC118–BC122
    document property, BC122
    left property, BC122
    load() method, BC136–BC137
    moveAbove() method, BC137
    moveBelow() method, BC137–BC138
    moveBy() method, BC138
    moveTo() method, BC138
    moveToAbsolute() method, BC138–BC140
    name property, BC124
    onblur event, BC143
    onfocus event, BC143
    onload event, BC143
    onmouseout event, BC143
    onmouseover event, BC143
    pageX property, BC124
    pageY property, BC124–BC128
    parentLayer property, BC128
    resizeBy() method, BC140
    resizeTo() method, BC140–BC143
    siblingAbove property, BC113, BC128
    siblingBelow property, BC113–BC116, BC128
    src property, BC128–BC131
    top property, BC122–BC124
    visibility property, BC131–BC132
    zIndex property, BC133–BC136
  layers property, 540–542
  layerX property
    NN4 event object, 749
    NN6+ event object, 780
  layerY property
    NN4 event object, 749
    NN6+ event object, 780

references, 172, BC111–BC112
resizing, BC141–BC142, BC164–BC166
source content, BC158–BC159
stacking order, 172, BC160–BC162
tables and, BC113
visibility behaviors, BC159–BC160
layout and inline display properties, `style` object, 821–825
`layoutGrid` property, 823
`layoutGridChar` property, 823
`layoutGridLine` property, 823
`layoutGridMode` property, 823
`layoutGridType` property, 824
`left` property
discussed, 826
`layer` object, BC122
`TextRectangle` object, 1102
`leftContext` property, BC211
`leftMargin` property, 580
`length` property
`Array` object, 917
arrays, 77, 143
`form` object, 641
function object, 986–987
as generic object, 235–236
`history` object, 503–504
`plugin` object, BC92
radio `input` object, 667
`select` object, 704
`string` object, 854
strings, 115
`letterSpacing` property, 817
`lhs` property, BC39
`li` object, BC63–BC64
libraries
scripts and, 147–148, 197
security checks, BC303
`light()` method, 839
line number reporting, errors, BC280–BC281
`lineHeight` property, 817
`lineNumber` property, 958
`link` object, BC8–BC12
`link` property, 578, 580
`linkBreak` property, 817
`linkColor` property, 179, 514, 542
linking behaviors, BC332
`links` property, 542
list properties, `style` object, 833
listings
`accessKey` property, 205–206
`addBehavior()` method, 261–263
`alertUser()` function, 53–54
anchors, Web page navigation, 517
`appendChild()` method, 268–269
applets, starting and stopping, BC246–BC247
`applyElement()` method, 269–270
arrays
`array.reverse()` method, 924
concatenation, 920–921
lookups, 913–914
populating, 911
banners, 403
behaviors
draggable element example, BC335–BC337
text rollovers, BC338–BC339
browsers
nonscriptable, 150
preferences, reading and writing, BC85–BC86, BC887
`canHaveChildren` property, 210–211
`charCode` property, 778–779
checkboxes, 103
`childNodes` property, 212–213
`children` property, 214
`className` property, 215–216
`clientHeight` property, 217
`clientWidth` property, 217
columns, in tables, editing, BC30–BC32
`compareEndPoints()` method, 1082–1084
`componentFromPoint()` method, 279–280
`const` keyword, 1018
`contentEditable` property, 218–219
context-sensitive help, 342–343
cut command, 331–332
data
binding, 221–222
sorting, 223–224
validation, 688

Continued

listings *(continued)*
date calculations, 118
date validation, BC224–BC227
dialog boxes
alert, 405
confirm, 413
modeless, 452–454
`dispatchEvent()` method, 281–282
documents, with anchors, 488–489
`document.write()` method, 92–94
DOM content, adding/replacing, 188
DOM (Document Object Model), 188
drag-and-drop operations, 336–338
dynamic properties, 320–321
errors, script errors, controlling, 387–388
events
canceling, 733–734
event capture and release, 723
event handlers, 45
modifier keys, 743–744
redirecting, 728–730
exceptions, throwing, 953–955
expressions, regular, 861–862
`fireEvent()` method, 283–285
`firstChild` property, 227–228
font properties, changing, 1026–1027
forms
date validation in, 905–907
`reset()` method, 633–634
simple example of, 535–536
`submit()` method, 633–634
submitting, 109
frames
blank, 365
forcing to load, 363
navigation bars, 127
functions
calling, 73
form elements and objects, passing, 107–108
generalizable, 993–994
recursion in, 992
`getBoundingClientRect()` function, 290–291
global variables, 75
image maps, 622–623
images
`align` property, 605
image rollovers, 134
motion and still, switching between, 608–609
precaching, 133
rotating, 616–617
still and motion, switching between, 608–609
indexes, branching index pages, 151–153
`innerHTML` property, 231–232
`innerText` property, 231–232
`insertBefore()`, 298–299
`isempty()` function, BC216–BC217
`isInteger()` function, BC219
`isNumber()` function, BC220
`isPosInteger()` function, BC217–BC218
`keyCode` property, 778–779
`lastChild` property, 227–228
layers
backgrounds, color, BC118
backgrounds, setting, BC116–BC117, BC144–BC146
coordinate systems, BC126–BC128
dragging, BC139–BC140, BC163–BC164
resizing, BC141–BC142, BC164–BC166
source content, BC158–BC159
stacking order, BC160–BC162
visibility behaviors, BC159–BC160
local variables, 75
`location.replace()` method, 501
`marquee` object, 1035–1036
`mergeAttributes()` method, 301
message status, setting default, 373
mime types, BC94
mouse, tracking, 561–562
node-related objects, 313–314
numbers, decimal to hexadecimal conversion, 880–881
`offsetParent` property, 243
`onbeforecopy`, 324
`onblur` event, 326–327

onclick event, 329
ondblclick event, 329
onfilterchange event, 340–341
onload event, 53
onmousedown event, 350
onmouseup event, 350
onpropertychange event, 355–356
onselectstart event, 357–358
opener property, 391
optgroup object, 715–716
outerHTML property, 244–245
outerText property, 244–245
pageXOffset property, 393–394
pageYOffset property, 393–394
parentTextEdit property, 248
paste command, 331–332
placeholders, 545
plug-ins, verification, BC94–BC95
pop-up lists
  color changes, 712
  hiding and showing, 483–484
radio objects, 104
read-only properties, 190
recordNumber property, 250–251
relatedTarget property, 786–787
releaseCapture() function, 305–306
reloading, hard versus soft, 500
removeBehavior() method, 262–263
scripts
  in body, 49–50
  in headings, 48
  hiding, 50
  immediate statements, 52
  <noscript> tag, 149
scroll values, determining, 582
scrolled images, 434
select object, 105–106
selection lists, 706
self property, 400–401
setCapture() method, 305–306
setInterval() method, 439–441
simple script example, 21–22
srcElement property, 771–772
statements
  immediate, 52
  labeled, 945–946
strings
  nested, 875
  prototypes, 855
  reading, 869–870
  replacing, BC204–BC205
  slicing, 866–867
  utility functions, 873–874
style sheets, applying and creating, 558–559
switch statement, 947–948
tabindex property, 255–257
tables
  cells, replacing data in, BC23–BC25
  DHTML, BC349–BC352
  dynamic, BC346–BC347
  rows, inserting and removing, BC27–BC28, BC30–BC31
  static, BC343–BC344
target property, 788–789
text
  fields, selecting, 684–685
  retrieving selected, 566–567
  values, getting and setting, 681–682
TextRectangle object, 1102–1103
time stamps, 540
timers, countdown, 409–410
timeStamp property, 789–790
title property, 258–259
trace() function, BC295
uniqueID property, 259–260
userProfile object, BC104–BC105
validations
  cross-confirmation, BC240
  dispatch lookup tables, BC229
  simple example of, BC231–BC232
Web pages
  file information, displaying, 533–534
  referrer pages, 548
window object, 86

Continued

listings *(continued)*
windows
chrome, controlling, 376–377
click events, 408
closing, checking before, 372
creating, 425–426
height and width, setting, 383–384
main window document, 129
moving, 417–419
resizing, 431–432
subwindow document, 129
lists
ordered, BC60
unordered, 289, BC62
`listStyle` property, 833
`listStyleImage` property, 833
`listStylePosition` property, 833
`listStyleType` property, 833
literal notation, arrays, 911–912
literals, strings, 112, 850
LiveConnect features, applets and, BC243–BC244
LiveScript language, 7
`load` event, 264
`load()` method, BC136–BC137
`loadCached()` function, 133
`loaded` value, 249
`loadedCache()` function, 615
`loadIndividual()` function, 615
loading
frames, 363
windows, 384
`loading` property, 384
`loading` value, 249
`loadOuter()` method, BC158
`loadXMLDoc()` function, BC397
local scope, variables, 75
local variables, 74–75
`localName` property
discussed, 183
as generic object, 236
`location` object
`assign()` method, 499
`document` object versus, 89
`hash` property, 488–489
`host` property, 489–492
`hostname` property, 492
`href` property, 89, 493–494
`newWindow` variable, 90
overview, 36, 486
`pathname` property, 494
`port` property, 494–495
`protocol` property, 495
`reload()` method, 499–500
`replace()` method, 500–501
roles of, 36
`search` property, 495
security checks, BC302
`userAgent` property, 90
`location` property, 385, 543
`locationbar` property, 375, 385
`location.hash` property, frames, 128
`location.href` property, 128
`location.ref` property, BC344
`location.replace()` method, 501
locking forms, 225
logos, positioning, 217
long string variables, 850
`longDesc` property
discussed, 465, 478
`Image` object, 611
lookups
arrays, 913–914
small data, 29
loops
`do-while`, 942
`for`, 72, 936–937
`loop` property
`Image` object, 611
`isindex` object, BC8
`marquee` object, 1038
repeat, 71–72, 534
`while`, 940–942
lowest common denominators, 165
`lowsrc` property
`Image` object, 612
`isindex` object, BC8

M

MacOS X, authoring environment, setting up, 19
macro targets, privileges, BC311
mailto file, 495
makeAreas() function, 625
makeHTML() function, BC398–BC399
makeNewWindow() function, 86
makeTitleRow() function, BC370
manipulation, strings, 873–874
map element object, 624–627
margin property, 831
marginBottom property, 831
marginHeight property, 466, 478
marginLeft property, 831
marginRight property, 831
marginTop property, 831
marginWidth property, 466, 478
markerOffset property, 824
markers, browsers, 10
marks property, 824
marquee object, 1034–1038
MASK constant, 752
mask() method, 839
maskfilter() method, 845
matching techniques, regular expressions, BC200–BC203
Math object
  methods, list of, 882, 884
  properties, list of, 883–884
  random() method, 115–116
  trigonometric functions, 115
maxHeight property, 824
maximize() method, 167
maxLength property
  isindex object, BC8
  text object, 679
maxValue parameter, 72
MAX_VALUE property, 887
maxWidth property, 824
media property
  discussed, 545
  link object, BC10–BC11
  style element object, 797
  styleSheet object, 801
medium level risks, privilege targets, BC311–BC312
memory
  allocation, 160
  RAM (random access memory), 36, 60
  script control, 160
menu object, BC65–BC66
menubar property, 375
menu-driven systems, events, 719–720
mergeAttributes() method, 209, 300–301
message notification, errors, BC278–BC279
message property, 958–959
message status, setting default, 373
meta object, BC12–BC14
<meta> tag
  alt attribute, 153
  content attribute, 151–153
metacharacters, BC194–BC195
metadata, BC13
metaKey property, 775
method attribute, 99
method property, 642
methods. *See also* commands; functions
  abort(), BC188
  add(), 710
  addBehavior()
    discussed, 177, BC332
    generic objects, 261–264
  AddDesktopComponent(), 378
  addEventListener(), 191, 264
  addImport(), 804–805
  adding to objects, 159
  addRange(), 1065
  addReadRequetst(), BC105–BC106
  addRule() styleSheet object, 805
  alert()
    functions and, 74
    strings, 112
    window object, 45, 87–88, 404–405
  appendChild()
    discussed, 139, 185
    generic objects, 267–269
  appendData(), 1070

Continued

methods *(continued)*
apply(), 987
applyElement(), 269–270
arguments, 44
array.concat(), 918–921
array.join(), 921–922
array.pop(), 923
array.push(), 923
array.reverse(), 923–925
array.shift(), 923
array.slice(), 925
array.sort(), 925–926
array.splice(), 929–930
array.toLocaleString(), 930
array.toString(), 930
array.unshift(), 923
assign(), 499
atEnd(), 1021
attachEvent(), 264, 270
back(), 406, 504
BasicImage(), 844
blendTrans(), 840
Blinds(), 844
blur()
filter object, 839
generic objects, 271
text object, 683
window object, 272
calculate(), BC370
call(), 987–988, BC258
captureEvents(), 406–408, 551–552
charAt(), 114
Checkerboard(), 844
chroma(), 839, 844
clear(), 552–553, 1065
clearAttributes(), 275
clearData(), 370
clearInterval(), 408–409
clearRequest(), BC106
clearTimeout(), 409–411
click()
button object, 655
checkbox input object, 661
generic objects, 276
radio input object, 668
cloneContents(), 1046
cloneNode(), 185, 276–277
cloneRange(), 1046
close()
discussed, 553–554
window object, 85–86, 411–412
collapse()
Range object, 1047
selection object, 1066
TextRange object, 1080–1081
collapseToEnd(), 1066
collapseToStart(), 1066
collisions between, 189
compareBoundaryPoints(), 1047–1048
compareEndPoints(), 1081–1084
comparePoint(), 1051
compile(), BC207
componentFromPoint(), 277–280
confirm(), 412–413
contains(), 279–280
containsNode(), 1066
createAttribute(), 307, 554
createCaption(), BC45
createComment(), 554
createContextualFragment(), 1051–1052
createControlRange(), 583
createDocumentFragment(), 555
createElement(), 186, 555–556
createEvent(), 556
createEventObject(), 557
createPopup(), 413
createRange(), 234, 557, 1066
CreateStyleSheet(), 557–559
createTextNode(), 186, 559–560
createTextRange(), 234, 583, 693
createTFoot(), BC46
createTHead(), BC46
createTreeWalker(), 560
custom methods, 999–1000
Date object, 895–896
Date.parse(), 899
dates, 117
defined, 44
defineGetter(), 189

defineSetter(), 189
deleteCaption(), BC45–BC46
deleteCell(), BC55
deleteContents(), 1052–1053
deleteData(), 1070
deleteFromDocument(), 1066
deleteRow(), BC46
deleteRule(), 806
deleteTFoot(), BC46
deleteTHead(), BC46
detach(), 1053–1054
detachEvent(), 270–271, 280
dimensions(), 1022
dispatchEvent(), 271, 280–282
document object, accessing, 90
document.createElement(), 94
document.createTextNode(), 94
document.getElementById(), 94–95
document.write(), 90, 92–94
doReadRequest(), BC107–BC108
doScroll(), 583–584
doValidate(), BC230
dropEffect(), 764
dropShadow(), 839
duplicate(), 1084–1085
effectAllowed(), 764
elementFromPoint(), 560–562
empty(), 1067
enablePrivilege(), BC310
errors in, BC282
events as, 168
exec(), BC208
execCommand()
  discussed, 562–564
  TextRange object, 1085–1087
execScript(), 414
expand(), 1087–1088
expression(), 570
extend(), 1067
extendRow(), BC367
extractContents(), 1054
Fade(), 844
fetchText(), BC254
find(), 414–415
findText(), 1088–1092
fireEvent(), 271, 282–285
firstChild(), 588
firstPage(), BC47
flipH(), 839
flipV(), 839
focus()
  discussed, 168
  generic objects, 271, 285
  text object, 683–684
forward()
  history object, 406, 504–505
  window object, 415
getAdjacentText(), 285–286
getAllResponseHeaders(), BC188
getAttribute()
  generic objects, 286–287
  userProfile object, BC108
getAttributeNode(), 287–288
getAttributeNodeNS(), 288–289
getAttributeNS(), 289
getAttributes(), 208–209
getBookmark(), 1092
getBoundingClientRect(), 289–291
getClientRects(), 292
getData(), 370
getDate(), 117
getDay(), 117
getElementById(), 90, 564–565
getElementsByName(), 565
getElementsByTagName(), 38, 292
getElementsByTagNameNS(), 292–293
getExpression, 293
getFile(), BC254
getFullYear(), 117, 897
getHours(), 117
getItem(), 1022
getMember(), BC258
getMinutes(), 117
getMonth(), 117
getRangeAt(), 1067
getResponseHeader(), BC188
getSeconds(), 117
getSelection(), 566–567
getSlot(), BC258

Continued

methods *(continued)*
getTime(), 117
getWindow(), BC258
getYear(), 117
glow(), 839
go(), 506–507
gray(), 839
handleEvent()
discussed, 415–416, 567
form object, 643
hasAttribute(), 293
hasAttributeNS(), 293
hasAttributes(), 293–294
hasChildNodes()
discussed, 186
generic objects, 294–295
hasFeature(), 538
hasOwnProperty(), 1007
hide(), 482
home(), 416
ibound(), 1022
indexOf(), 114
init(), 320
inRange(), 1093
insertAdjacentElement(), 227, 295–296
insertAdjacentHTML(), 296
insertAdjacentText(), 296–298
insertBefore(), 186, 298–299
insertCell(), BC55–BC56
insertData(), 1070
insertNode(), 1054–1055
insertRow(), BC46–BC47
insertRule(), 806–807
intersectsNode(), 1056
invert(), 839
invoking, 44, BC267
iris(), 845
isEqual(), 1093–1094
isPointInRange(), 1056
isPrototypeOf(), 1007
isSupported(), 186, 299
item()
discussed, 207
Enumerator object, 1021
generic objects, 300
select object, 711
javaEnabled(), BC84
lastChild(), 588
lastPage(), BC47
light(), 839
load(), BC136–BC137
loadOuter(), BC158
location.replace(), 501
mask(), 839
maskfilter(), 845
Math object, 882, 884
maximize(), 167
mergeAttributes(), 209, 300–301
motionblur(), 845
move(), 1094–1095
moveAbove(), BC137
moveBelow(), BC137–BC138
moveBy(), 416, BC138
moveEnd(), 1095
moveFirst(), 1021
moveNext(), 1021
moveRow(), BC47
moveStart(), 1095
moveTo(), 416–419, BC138
moveToAbsolute(), BC138–BC140
moveToBookmark(), 1095
moveToElementText(), 1095
moveToPoint(), 1096–1097
namedItem() method, 711
navigate(), 419–420
NavigateAndFind(), 378
navigator.Enabled(), 149
nextNode(), 588
nextPage(), BC47
nextSibling(), 588
node object, 185–186
normalize(), 302
objects and, 167
onerrorupdate(), 685–686
open()
discussed, 420, 567–568
xml object, BC189
openWindow(), 164

options(), 711
parameters, 44
parentElement(), 1097–1098
parentNode(), 588
pasteHTML(), 1098–1099
pop-up lists, 482–484
preference(), BC84–BC87
preventDefault(), 191, 280
previousNode(), 588–589
previousPage(), BC47–BC48
previousSibling(), 588
print(), 426–428
prompt(), 428–429
propertyIsEnumerable(), 1008
queryCommandCommandState(), 569
queryCommandEnabled(), 569, 1099
queryCommandIndeterm(), 1099
queryCommandIndterm(), 569
queryCommandState(), 1099
queryCommandSupported(), 569, 1099
queryCommandText(), 569, 1099
queryCommandValue(), 569–570, 1099
random(), 115–116
RandomDissolve(), 845
recalc(), 319, 570
refresh()
  plugin object, BC92
  table object, BC48
releaseCapture(), 302–303, 305–306
releaseEvents(), 429–430, 570
reload(), 499–500
remove(), 710–711
removeAllRanges(), 1067
removeAttribute(), 209, 306–307
removeAttributeNode(), 307
removeAttributeNS(), 308
removeBehavior(), 262–263
removeChild(), 186, 267, 309
removeEventListener(), 191, 264–267, 309
removeExpression(), 293, 309–310
removeNode(), 310–312
removeRange(), 1067
removeRule(), 805–806
replaceAdjacentText(), 298, 311–312
replaceChild(), 186–188, 267, 312
replaceData(), 1070
replaceNode(), 312–315
reset(), 643–644
resizeBy(), 430, BC140
resizeTo()
  discussed, 430–432
  layer object, BC140–BC143
revealTrans(), 840
revertPrivilege(), BC310
routeEvent(), 432–433, 571
scroll(), 433–435
scrollBy(), 435
scrollIntoView(), 204, 206, 315
scrollTo(), 435–437
select()
  text object, 684–685
  TextRange object, 1099
selectAllChildren(), 1068
selectNode(), 1056
selectNodeContents(), 1056–1057
send(), BC189
setActive(), 315–316
setAttribute()
  discussed, 187, 207, 209
  generic objects, 316–317
setAttributeNode(), 317
setAttributeNS(), 318
setCapture()
  discussed, 264
  generic objects, 302–306, 318
setColor(), BC145
setdata(), 330
setDate(), 117
setDay(), 117
setEnd(), 1057
setEndAfter(), 1059
setEndBefore(), 1059
setEndPoint(), 1100
setExpression()
  discussed, 293
  generic objects, 318–321

Continued

methods *(continued)*
setHours(), 117
setInterval(), 437–441
setMember(), BC258
setMinutes(), 117
setMonth(), 117
setSeconds(), 117
setSlot(), BC258
setStart(), 1057–1058
setStartAfter(), 1059
setStartBefore(), 1059
setTime(), 117
setTimeout(), 273, 371, 441–444
setYear(), 117
shadow(), 839
show(), 482–484
showHelp(), 444
showModalDialog(), 397, 444
showModelessDialog(), 444–447
sizeToContent(), 455
splitText(), 1073
start(), 1038
stop(), 455–456, 1038
stopPropagation(), 736
string.charAt(), 855–856
string.charCodeAt(), 856–857
string.concat(), 859
string.indexOf() method, 113–114, 859
string.lastIndexOf(), 860
string.localeCompare(), 860
string.match(), 861–862
string.replace(), 862–865
string.search(), 865
string.slice(), 865–867
string.split(), 868
string.substr(), 868–870
string.substring(), 114–115, 870–871
string.toLocaleLowerCase(), 871
string.toLocaleUpperCase(), 871
string.toLowerCase(), 872
string.toString(), 872
string.toUpperCase(), 872
string.valueOf(), 872
submit(), 108–109, 643–644
substring(), 524
substringData(), 1070
subWrite(), 93
surroundContents(), 1059–1060
swapNode(), 312, 321
tags, 321–322
taintEnabled(), BC87
test(), BC208–BC209
toArray(), 1022
toString()
discussed, 959
function object, 988
Range object, 1061
selection object, 1068
ubound(), 1022
urns, 322
valueOf(), 988
wave(), 839
window.confirm(), 88
window.open(), 85, 128
window.prompt(), 88–89
windows, accessing, 84–85
write(), 44, 571
writeln(), 571–575
xRay(), 839
Methods property, 596
Microsoft Script Debugger tool, BC290
mime types, verifying, BC94
mimeType object, BC87–BC90
mimeType property
a element object, 596
discussed, 546
Image object, 612
navigator object, BC79–BC80
minHeight property, 824
minimum system requirements, 1143
minutes, date functions, 117
MIN_VALUE property, 887
minWidth property, 824
minYear parameter, BC237
modeless dialog boxes, 452–454
modification methods, table rows, BC26
modifier keys, events, 743–744
modifiers property, 752

modulus (%) operator, 71
monospace fonts, 1028
months, date functions, 117
motion images, switching to still images, 608–609
motionblur() method, 845
mouse
  double click events, 332–333
  rollovers, 13
  tracking, 561–562
mousedown event, 146, 264
MouseEvents event, 556
mousemove event, 264
mouseout event, 264
mouseover event, 264
mouseup event, 264
move() method, 1094–1095
moveAbove() method, BC137
moveBelow() method, BC137–BC138
moveBy() method, 416, BC138
moveEnd() method, 1095
moveFirst() method, 1021
moveNext() method, 1021
moveRow() method, BC47
moveStart() method, 1095
moveTo() method, 416–419, BC138
moveToAbsolute() method, BC138–BC140
moveToBookmark() method, 1095
moveToElementText() method, 1095
moveToPoint() method, 1096–1097
moving
  table rows, BC47
  windows, 417–419
mozBorderRadius property, 831
mozBorderRadiusBottomLeft property, 831
mozBorderRadiusBottomRight property, 831
mozBorderRadiusTopLeft property, 831
mozBorderRadiusTopRight property, 831
Mozilla browsers, 10
mozOpacity property, 831
multidimensional arrays, 915–916
multi-frame navigation, 8
multiline property, BC207
multiple frames
  controlling, navigation bars, 126–128
  discussed, 31–32
multiple property, 704
multiplication (*) operator, 66
MutationEvents event, 556

N

name attribute
  form object, 99
  <form> tag, 91, 97
  id attribute versus, 40
name property
  a element object, 596–597
  applet object, BC171
  button object, 654
  discussed, 466, 479
  embed object, BC178
  Error object, 959
  form object, 642
  Image object, 612–613
  isindex object, BC8
  layer object, BC124
  meta object, BC14
  object object, BC176
  plugin object, BC92
  radio input object, 667
  text object, 679–680
  window object, 385
named node maps, 287
namedItem() method, 711
nameProp property
  a element object, 597
  discussed, 546
  Image object, 613
namespaces
  behaviors and, BC334
  namespaces property, 546
  namespaceURI property, 183, 236
naming
  cookies, 522
  functions, 73

Continued

naming *(continued)*
objects, 39–40
parameters, 106
strings, 134
variables, 61–62
windows, 385, 424
NaN property, 887–888
naturalHeight property, 613
naturalWidth property, 613–614
navigate() method, 419–420
NavigateAndFind() method, 378
navigation
multi-frame, 8
windows, 385
navigation bars, multiple frames, controlling, 126–128
navigator object
appCodeName property, BC69
appMinorVersion property, BC76
appName property, 44, 90, BC69
appVersion property, 44, 90, BC69
browserLanguage property, BC77
cookieEnabled property, BC78
cpuClass property, BC78–BC79
javaEnabled() method, BC84
language property, BC79
mimeTypes property, BC79–BC80
onLine property, BC80
oscpu property, BC80
overview, 36, BC68–BC69
platform property, BC81
plugins property, BC81–BC82
preference() method, BC84–BC87
product property, BC82
productSub property, BC82
roles of, 36
securityPolicy property, BC83
systemLanguage property, BC83
taintEnabled() method, BC87
userAgent property, BC69, BC83
userLanguage property, BC83
userProfile property, BC83
vendor property, BC82, BC84
vendorSub property, BC82, BC84
navigator property, 385
navigator.Enabled() method, 149
  (nonbreaking space character), BC22
NEGATIVE_INFINITY property, 887
nested elements, 209
nested functions, 983–984
nested strings, 73, 875–876
Netscape browsers, 10
new keyword, 77
newline (\n) character, 161, 692
newsgroups, 1139–1140
newWind variable, 371
newWindow() function, 128
newWindow variable
location object, 90
window object, 86
next property, 502
nextField() function, 221
nextNode() method, 588
nextPage() method, BC47
nextPage property, 769
nextSibling() method, 588
nextSibling property, 183–184, 237
nextWeek() function, 118
NN4 event object
data property, 748–749
layerX property, 749
layerY property, 749
modifiers property, 752
pageX property, 749
pageY property, 749
screenX property, 749
screenY property, 749–751
target property, 752
type property, 753
which property, 753
NN6+ event object
altKey property, 775
bubbles property, 776
button property, 776
cancelable property, 776–777
cancelBubble property, 77
charCode property, 77
clientX property, 780

clientY property, 780
ctrlKey property, 775
currentTarget property, 783–785
detail property, 785
eventPhase property, 785
isChar property, 785
keyCode property, 777–779
layerX property, 780
layerY property, 780
metaKey property, 775
originalTarget property, 785–786
pageX property, 780
pageY property, 780
relatedTarget property, 786–787
screenX property, 780
screenY property, 780
shiftKey property, 775–776
target property, 787–789
timeStamp property, 789–790
node maps, 226, 287
node trees, HTML structure, 41
nodeName property
discussed, 183–184
as generic object, 237–238
nodes
child
appendChild() method, 139
defined, 42
discussed, 227
content, replacing, 187–189
creatTextNode() method, 186
defined, 41
document, 42
hierarchy of, 180–182
HTML-related, 182
new content, generating, 186–187
node trees, 41
object methods, list of, 185–186
object properties, list of, 183
parent, 42
root, 1043
text, 38–39
nodeType property
discussed, 184
as generic object, 238–239
nodeValue property
discussed, 183–184
as generic object, 239–240
nonbreaking space character ( ), BC22
non-repeating objects, 250
nonscriptable browsers, 50–51, 149–150
noResize property, 466
normalize() method, 302
<noscript> tag, 148–149
noShade property, 1032
not equal to (!=) operator, 67
noWrap property
discussed, 580–581
td object, BC59–BC60
null value, 86
Number() function, 1013
Number object
methods, list of, 888–890
overview, 886
properties, list of, 887–888
number property, 959
numbers
adding, 64
converting
to strings, 65–66, 882
strings to, 65, 881–882
decimal to hexadecimal conversion, 880–881
errors in, BC282
floating-point, 65, 877–879
lowest common denominators, 165
maxValue parameter, 72
random number generation, 115–116
Roman numerals, BC64
startValue parameter, 72

O

object model, frames, 360–361
object object
align property, BC173
alt property, BC173
altHTML property, BC173
BaseHref property, BC173

Continued

object object *(continued)*
baseURI property, BC173
border property, BC174
classid property, BC174
code property, BC174
codeBase property, BC174
codeType property, BC175
contentDocument property, BC175
data property, BC175
discussed, 1006–1008
form property, BC175
height property, BC175
hspace property, BC175
name property, BC176
object property, BC176
overview, BC171–BC172
type property, BC176
useMap property, BC176
vspace property, BC175–BC176
width property, BC175–BC176
object property
applet object, BC171
object object, BC176
object-oriented (OO) programming, 185
objects
a element, 595
ActiveXObject, 1019–1020
applet, BC167–BC171
application level, BC68
area element
overview, 621–622
properties, list of, 621–622
Array
array.concat() method, 918–921
array.join() method, 921–922
array.pop() method, 923
array.push() method, 923
array.reverse() method, 923–925
array.shift() method, 923
array.slice() method, 925
array.sort() method, 925–926
array.splice() method, 929–930
array.toLocaleString() method, 930
array.toString() method, 930
array.unshift() method, 923
properties, list of, 916–918
in arrays, 80
automation, 1019
base, BC5–BC6
basefont, BC6–BC7
blockquote, 1024
body element
aLink property, 578
background property, 579
bgColor property, 578–579
bgProperties property, 579
bottomMargin property, 580
createControlRange() method, 583
createTextRange() method, 583
doScroll() method, 583–584
leftMargin property, 580
link property, 578, 580
noWrap property, 580–581
onafterprint property, 583–584
onbeforeprint property, 584
onscroll property, 585
overview, 577
rightMargin property, 580–581
scroll property, 581
scrollLeft property, 581
scrollTop property, 581–582
text property, 578, 583
topMargin property, 580, 583
vLink property, 583
Boolean, 890
br, 1024–1025
built-in, 158
button
click() method, 655
form property, 653–654
name property, 654
onclick event, 655–656
onmousedown event, 656
onmouseup event, 656
overview, 651–652
type property, 654
value property, 654

buttons as, 102
checkboxes as, 102–103
clientInformation, BC68
col, BC50–BC52
colgroup, BC50–BC52
core language, 111
cssRule, 807–810
Date
  calculations, 117–118
  creating, 893–894
  form validation, 904–907
  getMonth() method, 117
  GMT (Greenwich Mean Time), 892
  methods, list of, 117, 895–896
  references, 1108
  setMonth() method, 117
  snapshots of, 116
  strings and, 898
  time zones, 891–892
  today variable, 116
dd, BC64–BC65
detection, DHTML, BC323–BC324
Dictionary, 1020
dir, BC65–BC66
dl, BC64–BC65
document
  activeElement property, 513
  alinkColor property, 179, 514
  anchors property, 516–517
  applets property, 517–518
  in arrays, 80
  bgColor property, 179, 514, 518
  body property, 518
  captureEvents() method, 551–552
  characterSet property, 519
  charset property, 518–519
  clear() method, 552–553
  close() method, 553–554
  compatMode property, 519–520
  cookie property, 520–521
  createAttribute() method, 554
  createComment() method, 554
  createDocumentFragment() method, 555
createElement() method, 555–556
createEvent() method, 556
createEventObject() method, 557
createRange() method, 557
createStyleSheet() method, 557–559
createTextNode() method, 559–560
createTreeWalker() method, 560
defaultCharset property, 529
defaultView property, 529
designMode property, 530
doctype property, 530
document.createElement() method, 94
document.createTextNode() method, 94
documentElement property, 530–531
document.forms property, 91
document.getElementById() method, 94–95
document.images property, 91
document.write() method, 90, 92–94
domain property, 531
elementFromPoint() method, 560–562
embeds property, 531
execCommand() method, 562–564
expando property, 532
fgColor property, 514, 532
fileCreatedDate property, 532
fileModifiedDate property, 532
fileSize property, 532–534
forms property, 534–536
frames property, 536–537
getElementById() function, 90
getElementById() method, 564–565
getElementsByName() method, 565
getSelection() method, 566–567
handleEvent() method, 567
height property, 537
ids property, 537
images property, 537–538
implementation property, 538–539
lastModified property, 539
layers property, 540–542

Continued

objects, document *(continued)*
linkColor property, 179, 514, 542
links property, 542
location object versus, 89
location property, 543
media property, 545
mimeType property, 546
nameProp property, 546
namespaces property, 546
onselectionchange event, 575
onstop event, 575
open() method, 567–568
overview, 36, 512
parentWindow property, 546
plugins property, 546–547
properties and methods, accessing, 90
protocol property, 547
queryCommandCommandState() method, 569
queryCommandEnabled() method, 569
queryCommandIndterm() method, 569
queryCommandSupported() method, 569
queryCommandText() method, 569
queryCommandValue() method, 569–570
recalc() method, 570
referrer property, 547–548
releaseEvents() method, 570
roles of, 36
routeEvent() method, 571
scripts property, 548–549
security checks, BC303
security property, 549
selection property, 549–550
styleSheets property, 550
subWrite() method, 93
syntax, 90
tags property, 550
title property, 550
URL property, 543–545, 551
URLUnencoded property, 551
vlinkColor property, 179, 514–516, 551
width property, 537, 551
write() method, 44, 571
writeln() method, 571–575
document object hierarchy, 164–165
dt, BC64–BC65
embed, BC177–BC179
Error
methods, list of, 959
overview, 957
properties, 958–959
event handlers and, 168–169
expected errors, BC281
fieldset element, 646–647
file, 717–718
filter
filter names, 843–846
overview, 837–838
reveal transitions, 841–842
static filter types, 838–839
subproperties, reading and writing, 838
syntax changes, 842–843
transition filters, 840
font, 1025–1028
form
acceptCharset property, 638
action attribute, 99, 108
action property, 638
autocomplete property, 639
elements property, 639–640
encoding property, 641
enctype attribute, 99
enctype property, 641
form.elements property, 99
handleEvent() method, 643
length property, 641
method attribute, 99
method property, 642
name attribute, 99
name property, 642
onreset() method, 645–646
onsubmit event, 646
processData() function, 107
reset() method, 643–644
security checks, BC303
submit() method, 108–109, 644

target attribute, 99
target property, 642–643
frame
allowTransparency property, 463
borderColor property, 463
contentDocument property, 463–464
contentWindow property, 464
frameBorder property, 464–465
height property, 465
longDesc property, 465
marginHeight property, 466
marginWidth property, 466
name property, 466
noResize property, 466
scrolling property, 467
src property, 468
width property, 465
frame element objects versus frames, 365–366
frameset
border property, 470
borderColor property, 471
cols property, 471
frameBorder property, 474
frameSpacing property, 474
rows property, 471–473
function
apply() method, 987
arguments property, 984–985
arity property, 985
call() method, 987–988
caller property, 985–986
constructor property, 986
length property, 986–987
prototype property, 987
toString() method, 988
valueOf() method, 988
h1, 1029
h2, 1029
h3, 1029
h4, 1029
h5, 1029
h6, 1029
head, BC4–BC5
hidden, 689–690
history
back() method, 503–504
current property, 502
forward() method, 504–506
go() method, 506–507
length property, 503–504
next property, 502–503
overview, 36
previous property, 502–503
roles of, 36
hr, 1029–1033
HTML, BC3–BC4
HTMLParagraphElement, 189
identifiers, 40
IE4+ event object
altKey property, 755
altLeft property, 756
behaviorCookie property, 756
behaviorPart property, 756
bookmarks property, 756
boundElements property, 756
button property, 757–758
cancelBubble property, 758
clientX property, 759
clientY property, 759
ctrlKey property, 755
ctrlLeft property, 756
dataFld property, 756
dataTransfer property, 763–765
fromElement property, 765
keyCode property, 767–769
nextPage property, 769
offsetX property, 759
offsetY property, 759
propertyName property, 769
qualifier property, 756
reason property, 756
recordset property, 756
repeat property, 769
returnValue property, 770
saveType property, 770
screenX property, 759

Continued

objects, IE4+ event object *(continued)*
screenY property, 759
shiftKey property, 755–756
shiftLeft property, 756
srcElement property, 770–772
srcFilter property, 772
srcUrn property, 772
toElement property, 765–767, 772
type property, 772–773
wheelData property, 773
x property, 759
y property, 759–763
IE4 extensions, 174–176
IE5 extensions, 177
Image
align property, 604–605
alt property, 606
border property, 606
complete property, 606–607
dynsrc property, 608–609
fileCreatedDate property, 609
fileModifiedDate property, 609
fileSize property, 609–610
fileUpdatedDate property, 609
height property, 610
href property, 610
hspace property, 610
isMap property, 611
longDesc property, 611
loop property, 611
lowsrc property, 612
mimeType property, 612
name property, 612–613
nameProp property, 613
naturalHeight property, 613
naturalWidth property, 613–614
onabort event, 619
onerror property, 619
onload event, 619–620
overview, 602–603
protocol property, 614
src property, 614
start property, 618
useMap property, 618
vspace property, 610–611, 618
width property, 610, 618
x property, 618
y property, 618–619
images as, 153
input
checkboxes, 657–661
images, 669–671
radio objects, 663–667
isindex, BC7–BC8
label, 1033–1034
label element, 646–649
layer
above property, BC113
background property, BC116–BC117
below property, BC113
bgColor property, BC117–BC118
clip property, BC118–BC122
document property, BC122
left property, BC122
load() method, BC136–BC137
moveAbove() method, BC137
moveBelow() method, BC137–BC138
moveBy() method, BC138
moveTo() method, BC138
moveToAbsolute() method, BC138–BC140
name property, BC124
onblur event, BC143
onfocus event, BC143
onload event, BC143
onmouseout event, BC143
onmouseover event, BC143
pageX property, BC124
pageY property, BC124–BC128
parentLayer property, BC128
resizeBy() method, BC140
resizeTo() method, BC140–BC143
siblingAbove property, BC113, BC128
siblingBelow property, BC113–BC116, BC128
src property, BC128–BC131
top property, BC122–BC124
visibility property, BC131–BC132
zIndex property, BC133–BC136

li, BC63–BC64
link, BC8–BC12
location
  assign() method, 499
  document object versus, 89
  hash property, 488–489
  host property, 489–492
  hostname property, 492
  href property, 89, 493–494
  newWindow variable, 90
  overview, 36, 486
  pathname property, 494
  port property, 494–495
  protocol property, 495
  reload() method, 499–500
  replace() method, 500–501
  roles of, 36
  search property, 495
  security checks, BC302
  userAgent property, 90
map element, 624–627
marquee, 1034–1038
Math
  methods, list of, 882, 884
  properties, list of, 883–884
  random() method, 115–116
  trigonometric functions, 115
menu, BC65–BC66
meta, BC12–BC14
methods, adding, 159
methods and, 167
mimeType, BC87–BC90
naming, 39–40
navigator
  appCodeName property, BC69
  appMinorVersion property, BC76
  appName property, 44, 90, BC69
  appVersion property, 44, 90, BC69
  browserLanguage property, BC77
  cookieEnabled property, BC78
  cpuClass property, BC78–BC79
  javaEnabled() method, BC84
  language property, BC79
  mimeTypes property, BC79–BC80
  onLine property, BC80
  oscpu property, BC80
  overview, 36, BC68–BC69
  platform property, BC81
  plugins property, BC81–BC82
  preference() method, BC84–BC87
  product property, BC82
  productSub property, BC82
  roles of, 36
  securityPolicy property, BC83
  systemLanguage property, BC83
  taintEnabled() method, BC87
  userAgent property, BC69, BC83
  userLanguage property, BC83
  userProfile property, BC83
  vendor property, BC82
  vendorSub property, BC82, BC84
NN4 event object
  data property, 748–749
  layerX property, 749
  layerY property, 749
  modifiers property, 752
  pageX property, 749
  pageY property, 749
  screenX property, 749
  screenY property, 749–751
  target property, 752
  type property, 753
  which property, 753
NN6+ event object
  altKey property, 775
  bubbles property, 776
  button property, 776
  cancelable property, 776–777
  cancelBubble property, 77
  charCode property, 77
  clientX property, 780
  clientY property, 780
  ctrlKey property, 775
  currentTarget property, 783–785
  detail property, 785
  eventPhase property, 785
  isChar property, 785
  keyCode property, 777–779
  layerX property, 780

Continued

objects, NN6+ event object *(continued)*
layerY property, 780
metaKey property, 775
originalTarget property, 785–786
pageX property, 780
pageY property, 780
relatedTarget property, 786–787
screenX property, 780
screenY property, 780
shiftKey property, 775–776
target property, 787–789
timeStamp property, 789–790
node object properties, list of, 183
non-repeating, 250
Number
methods, list of, 888–890
overview, 886
properties, list of, 887–888
object
align property, BC173
alt property, BC173
altHTML property, BC173
BaseHref property, BC173
baseURI property, BC173
border property, BC174
classid property, BC174
code property, BC174
codeBase property, BC174
codeType property, BC175
contentDocument property, BC175
data property, BC175
discussed, 1006–1008
form property, BC175
height property, BC175
hspace property, BC175
name property, BC176
object property, BC176
overview, BC171–BC172
type property, BC176
useMap property, BC176
vspace property, BC175–BC176
width property, BC175–BC176
object detection, 153–155
object model families, 170
object model hierarchy, 163
ol, BC60–BC62
optgroup, 714–716
option, 713
param, BC179
plugin, BC90–BC92
properties, 166
prototypes of, 159
radio, 103–104
Range
cloneContents() method, 1046
cloneRange() method, 1046
collapse() method, 1047
collapsed property, 1043
commonAncestorContainer property, 1043–1044
compareBoundaryPoints() method, 1047–1048
comparePoint() method, 1051
createContextualFragment() method, 1051–1052
deleteContents() method, 1052–1053
detach() method, 1053–1054
endContainer property, 1044
endOffset property, 1044
extractContents() method, 1054
insertNode() method, 1054–1055
intersectsNode() method, 1056
isPointInRange() method, 1056
overview, 1039–1040
selectNode() method, 1056
selectNodeContents() method, 1056–1057
setEnd() method, 1057
setEndAfter() method, 1059
setEndBefore() method, 1059
setStart() method, 1057–1058
setStartAfter() method, 1059
setStartBefore() method, 1059
startContainer property, 1044
startOffset property, 1044–1045
surroundContents() method, 1059–1060
toString() method, 1061

referencing, 40
RegExp
  compile() method, BC207
  constructor property, BC206
  exec() method, BC208
  global property, BC206
  ignoreCase property, BC206
  input property, BC210
  lastIndex property, BC207
  lastMatch property, BC210
  leftContext property, BC211
  multiline property, BC207, BC210
  prototype property, BC211
  rightContext property, BC211
  source property, BC207
  test() method, BC208–BC209
roles of, 42–45
rule, 807–810
screen
  availHeight property, BC100
  availLeft property, BC101
  availTop property, BC101–BC102
  availWidth property, BC100
  bufferDepth property, BC102
  colorDepth property, BC102
  discussed, 397
  fontSmoothingEnabled property, BC103
  height property, BC100
  overview, 36, BC99–BC100
  pixelDepth property, BC102–BC103
  roles of, 36
  updateInterval property, BC103
  width property, BC100–BC101
script, BC14–BC16
select
  add() method, 710
  item() method, 711
  length property, 704
  multiple property, 704
  namedItem() method, 711
  navigating with, 105–106
  onchange event, 711–712
  options() method, 711
  options property, 705–708
  overview, 695–696
  pop-up lists, 104–105
  remove() method, 710–711
  selectedIndex property, 105, 709
  size property, 710
  text property, 105
  type property, 710
  value property, 710
selection
  addRange() method, 1065
  anchorNode property, 1063
  anchorOffset property, 1063
  clear() method, 1065
  collapse() method, 1066
  collapseToEnd() method, 1066
  collapseToStart() method, 1066
  containsNode() method, 1066
  createRange() method, 1066
  deleteFromDocument() method, 1066
  empty() method, 1067
  extend() method, 1067
  focusNode property, 1063
  focusOffset property, 1063
  getRangeAt() method, 1067
  isCollapsed property, 1063
  overview, 1061–1062
  rangeCount property, 1063
  removeAllRanges() method, 1067
  removeRange() method, 1067
  selectAllChildren() method, 1068
  toString() method, 1068
  type property, 1064–1065
static, 189–190
string
  constructor property, 853–854
  length property, 854
  prototype property, 854–855
  string.charAt() method, 855–856
  string.charCodeAt() method, 856–857
  string.concat() method, 859
  string.indexOf() method, 859

Continued

objects, string *(continued)*
string.lastIndexOf() method, 860
string.localeCompare() method, 860
string.match() method, 861–862
string.replace() method, 862–865
string.search() method, 865
string.slice() method, 865–867
string.split() method, 868
string.substr() method, 868–870
string.substring() method, 870–871
string.toLocaleLowerCase() method, 871
string.toLocaleUpperCase() method, 871
string.toLowerCase() method, 872
string.toString() method, 872
string.toUpperCase() method, 872
string.valueOf() method, 872
style
aural properties, 836–837
background properties, 828–829
border and edge properties, 829–833
font properties, 815–816
inline display and layout properties, 821–825
list properties, 833
page and printing properties, 835–836
positioning properties, 826–828
scroll bar properties, 834
table properties, 834–835
text properties, 817–820
style element, 796–797
styleSheet
addImport() method, 804–805
addRule() method, 805
cssRules property, 799
cssText property, 799–800
deleteRule() method, 806
disabled property, 800
href property, 800
id property, 800
imports property, 801
insertRule() method, 806–807
media property, 801
ownerNode property, 801
ownerRule property, 802
owningElement property, 802
pages property, 802–803
readOnly property, 803
removeRule() method, 805–806
rules property, 803–804
title property, 804
type property, 804
table
align property, BC35
background property, BC35
bgColor property, BC36
border property, BC36
borderColor property, BC36
borderColorDark property, BC36
borderColorLight property, BC36–BC37
caption property, BC37
cellPadding property, BC37
cells property, BC38
cellSpacing property, BC37–BC38
cols property, BC38
createCaption() method, BC45
createTFoot() method, BC46
createTHead() method, BC46
dataPageSize property, BC39
deleteCaption() method, BC45–BC46
deleteRow() method, BC46
deleteTFoot() method, BC46
deleteTHead() method, BC46
firstPage() method, BC47
frame property, BC39–BC41
height property, BC41
insertRow() method, BC46–BC47
lastPage() method, BC47
moveRow() method, BC47
nextPage() method, BC47
previousPage() method, BC47–BC48
refresh() method, BC48
rows property, BC42
rules property, BC42–BC44
summary property, BC44
tBodies property, BC44–BC45

tFoot property, BC45
tHead property, BC45
width property, BC41, BC45
tbody, BC48–BC50
td, BC56–BC60
text
blur() method, 683
defaultValue property, 677–678
focus() method, 683–684
form property, 678
maxLength property, 679
name property, 679–680
onafterupdate event, 685
onbeforeupdate event, 685
onblur event, 686
onchange event, 688
onerrorupdate() method, 685–686
onfocus event, 686
onselect event, 686–687
overview, 674
readOnly property, 680
select() method, 684–685
size property, 680
type property, 681
value property, 681–683
text fields as, 28
TextNode, 1069–1073
TextRange
boundingHeight property, 1077
boundingLeft property, 1077
boundingTop property, 1077
boundingWidth property, 1077–1079
collapse() method, 1080–1081
compareEndPoints() method, 1081–1084
duplicate() method, 1084–1085
execCommand() method, 1085–1087
expand() method, 1087–1088
findText() method, 1088–1092
getBookmark() method, 1092
htmlText property, 1079–1080
inRange() method, 1093
isEqual() method, 1093–1094
move() method, 1094–1095
moveEnd() method, 1095
moveStart() method, 1095
moveToBookmark() method, 1095
moveToElementText() method, 1095
moveToPoint() method, 1096–1097
overview, 1074–1075
parentElement() method, 1097–1098
pasteHTML() method, 1098–1099
queryCommandEnabled() method, 1099
queryCommandIndeterm() method, 1099
queryCommandState() method, 1099
queryCommandSupported() method, 1099
queryCommandText() method, 1099
queryCommandValue() method, 1099
select() method, 1099
setEndPoint() method, 1100
text property, 1080
TextRectangle, 1101–1103
tfoot, BC48–BC50
th, BC56–BC60
thead, BC48–BC50
title, BC16–BC17
top-level, 37
tr
cells property, BC54
deleteCell() method, BC55
height property, BC54–BC55
insertCell() method, BC55–BC56
overview, BC52–BC53
rowIndex property, BC55
sectionRowIndex property, BC55
TreeWalker
methods, 588–589
overview, 586
properties, 587
ul, BC62–BC63
userProfile
addReadRequetst() method, BC105–BC106
clearRequest() method, BC106
data, accessing, BC104–BC105

Continued

objects, userProfile *(continued)*
doReadRequest() method, BC107–BC108
getAttribute() method, BC108
overview, BC103–BC104
value property and, 166
VBArray, 1021–1022
window
alert() method, 45, 87–88, 404–405
appCore property, 369
back() method, 406
blur() method, 272
captureEvents() method, 406–408
clearInterval() method, 408–409
clearTimeout() method, 409–411
clientInformation property, 370
clipboardData, 370–371
close() method, 85–86, 411–412
closed property, 371–372
closeNewWindow() function, 86
Components property, 369, 372
confirm() method, 412–413
controllers property, 369, 372
createPopup() method, 413
crypto property, 373
defaultStatus property, 373–374
dialogArguments property, 374
dialogHeight property, 374
dialogLeft property, 375
dialogTop property, 375
dialogWidth property, 374–375
directories property, 375
document property, 377–378
DOM (document object model), 35
event property, 378
events and, 176
execScript() method, 414
external property, 378–379
find() method, 414–415
forward() method, 406, 415
frameElement property, 379
frames property, 379–381
fullScreen() function, 167
handleEvent() method, 415–416
history property, 382
home() method, 416
innerHeight property, 382
innerWidth property, 382
loading property, 384
location property, 385
locationbar property, 375, 385
makeNewWindow() function, 86
maximize() method, 167
menubar property, 375
moveBy() method, 416
moveTo() method, 416–419
name property, 385
navigate() method, 419–420
navigator property, 385
newWindow() function, 128
newWindow variable, 86
offscreenBuffering property, 386
onafterprint event, 456
onbeforeprint event, 456
onbeforeunload event, 456–457
ondragdrop event, 457–458
onerror event, 458
onerror property, 386–388
onhelp event, 458–459
onload event, 52, 89, 459–460
onmove event, 460
onresize event, 460–461
onunload event, 461
open() method, 420
opener property, 128, 389–391
openWindow() method, 164
outerHeight property, 382, 392
outerWidth property, 382–384, 392
overview, 36, 84, 368–369
pageXOffset property, 392
pageYOffset property, 392–394
parent property, 394–397
personalbar property, 375, 397
pkcs11 property, 373
print() method, 426–428
prompt() method, 428–429
prompter property, 369
releaseEvents() method, 429–430

resizeBy() method, 430
resizeTo() method, 430–432
returnValue property, 397
roles of, 36
routeEvent() method, 432–433
screen property, 397
screenLeft property, 398
screenTop property, 398
screenX property, 398
screenY property, 398–399
scroll() method, 433–435
scrollbars property, 375, 399
scrollBy() method, 435
scrollMaxX property, 399
scrollMaxY property, 399
scrollTo() method, 435–437
scrollX property, 399
scrollY property, 399
self property, 400–401
self value, 85
setInterval() method, 437–440
setTimeout() method, 441–444
showHelp() method, 444
showModalDialog() method, 444
showModelessDialog() method, 444–445
sidebar property, 369–370, 401
sizeToContent() method, 455
status property, 401–404
statusbar property, 375, 404
stop() method, 455–456
subWindow variable, 85
toolbar property, 375–377, 404
top property, 404
window property, 404
window.confirm() method, 88
window.open() method, 85, 128
window.prompt() method, 88–89
window.status property, 87
xml, BC181–BC184
XMLHttpRequest, BC185–BC189
offscreenBuffering property, 386
offset parameter, 1057
offset properties, 174
offsetHeight property, 229, 240
offsetLeft property, 241
offsetParent property, 242–243
offsetTop property, 241
offsetWidth property, 229, 240–241
offsetX property, 759
offsetY property, 759
OKToTest flag, BC266
ol object, BC60–BC62
onabort event, 619
onactivate event, 323
onafterprint event, 456, 584
onafterupdate event, 685
onbeforecopy event, 324
onbeforecut event, 324–325
onbeforedeactivate event, 323, 325
onbeforeeditfocus event, 325
onbeforepaste event, 325–326
onbeforeprint event, 456, 584
onbeforeunload event, 456–457
onbeforeupdate event, 685
onblur event
discussed, 326–327
layer object, BC143
text object, 686
onbounce event, 1039
onchange event
select object, 711–712
text object, 688
this keyword, 106
onclick event
button object, 655–656
checkbox input object, 661–663
discussed, 168, 329
generic objects, 327–329
radio input object, 668
oncontextmenu event, 330
oncontrolselect event, 330
oncopy event, 330
oncut event, 330–331
ondblclick event, 329, 332–333
ondeactivate event, 323
ondrag event, 333–337
ondragdrop event, 457–458

ondragend event, 333–337
ondragenter event, 338
ondragleave event, 338
ondragover event, 338–339
ondragstart event, 333–337, 339
ondrop event, 339–340
onerror property, 386–388, 619
onerrorupdate() method, 685–686
onfilterchange event, 340–341
onfinish event, 1039
onfocus event
  discussed, 341–342
  layer object, BC143
  text object, 686
onhelp event, 342–343, 458–459
onkeydown event, 343
onkeypress event, 343
onkeyup event, 343–344
online documentation, as resource, 1140–1141
onLine property, BC80
onload event
  <body> tag, 168
  frames, 125
  Image object, 619–620
  layer object, BC143
  link object, BC12
  running scripts from, 53
  window object, 52, 89
onlosecapture event, 348–349
onmousedown event, 349–350, 656
onmouseenter event, 351
onmouseleave event, 351
onmousemove event, 351
onmouseout event, 153, 352, BC143
onmouseover event, 153, 351, BC143
onmouseup event, 349–351, 656
onmove event, 460
onpaste event, 353–355
onpropertychange event, 355–356
onreadystatechange event, 356–357
onreset event, 645–646
onresize event, 357, 460–461
onresizeend event, 357
onresizestart event, 357
onscroll event, 584
onselect event, 686–687
onselectionchange event, 575
onselectstart event, 357–358
onstart event, 1039
onstop event, 575
onsubmit event
  form object, 646
  <form> tag, 102
onTarget() function, BC453
onunload event, 376, 461
OO (object-oriented) programming, 185
open() method
  discussed, 420, 567–568
  xml object, BC189
opener property, window object, 128, 389–391
openWindow() method, 164
Opera browsers, 7, 10
operators
  arithmetic, 66
  assignment, 61, 967–968
  bitwise, 972–973
  Boolean, 968–972
  calculations, 66
  categories, 961–962
  comparison, 66–67, 962–963
  connubial, 965–967
  delete, 973–974
  equality comparisons, 963–964
  in, 974
  instanceof, 974–975
  operations, 66
  operator precedence, 978–980
  short-circuit evaluation, 980
  subtract-by-value, 128
  this, 975–976
  typeof, 977–978
  unary, 966
OPML (Outline Processor Markup Language), BC394–BC395
optgroup object, 714–716
option object, 713
options() method, 711

options property, 705–708
ordered lists, BC60
O'Reilly & Associates, Inc. *(Java in a Nutshell)*, BC275
origin checks, security, BC302
originalTarget property, 785–786
orphans property, 835
oscpu property, BC80
Outdent command, documents, 563
outerHeight property, 382, 392
outerHTML property
  as generic object, 244–245
  read/write access, 174
outerText property
  as generic object, 244–245
  read/write access, 174
outerWidth property, 382–384, 392
Outline Processor Markup Language (OPML), BC394–BC395
outline property, 832
outlineColor property, 832
outlineStyle property, 832
outlineWidth property, 832
output stream, browsers, 92
output variable, 75
overflow property, 824
overflowX property, 825
overflowY property, 825
overlapped elements, 196
ownerDocument property, 183, 245–246
ownerNode property, 801
ownerRule property, 802
owningElement property, 802

P

<p> tag, 34, 38
packages, classes and, BC275
padding property, 832
paddingBottom property, 833
paddingLeft property, 833
paddingRight property, 833
paddingTop property, 833
page and printing properties, style object, 835–836
page property, 835
pageBreakAfter property, 835
pageBreakBefore property, 835
pageBreakInside property, 835
pages. *See* Web pages
pages property, 802–803
pageX property
  layer object, BC124
  NN4 event object, 749
  NN6+ event object, 780
pageXOffset property, 392–394
pageY property
  layer object, BC124–BC128
  NN4 event object, 749
  NN6+ event object, 780
pageYOffset property, 392–394
paragraphs
  content, adding, 187
  text, 38
parallel arrays, 78–80, 912–913
param object, BC179
parameters. *See also* arguments
  afterBegin, 285
  afterEnd, 285
  beforeBegin, 285
  beforeEnd, 285
  CharSet, 220
  DataURL, 220
  defined, 24, 44, 74
  EscapeChar, 220
  FieldDelim, 220
  function parameters
    overview, 984, 992
    passing, 438–439
  functions and, 73–74
  Language, 220
  maxValue, 72
  minYear, BC237
  naming, 106
  offset, 1057
  passing, 442
  relativeNumber, 506
  RowDelim, 220

Continued

parameters *(continued)*
startValue, 72
strings as, 74
TextQualifier, 220
this.form, 633
UseHeader, 220
windowFeatures, 420–421
parent nodes
defined, 42
siblings, 181
parent property, 394–397
parent window, frames, 122–123
parentElement() method, 1097–1098
parentElement property, 246
parentFrame property, 379
parentheses (( )), 24
parentLayer property, BC128
parentNode() method, 588
parentNode property
discussed, 183–184
as generic object, 247
parentStyleSheet property, 808
parentTextEdit property, 247–249
parent-to-child references, frames, 123–124, 362
parentWindow property, 546
parseFloat() function
converting strings to number, 65
discussed, 428
global functions, 1013
parseInt() function
converting strings to numbers, 65
discussed, 375, 428
global functions, 1013–1014
passing
events, 725
forms to functions, 631–634
function parameters, 438–439
text to functions, 682
passwords, throwaway, 521
paste command, 331–332, BC325
pasteHTML() method, 1098–1099
path settings, cookies, 523
pathname property
a element object, 595
area element object, 624
discussed, 494
patterns, regular expressions and, BC191–BC192
pause property, 837
pauseAfter property, 837
pauseBefore property, 837
PDAs (personal digital assistants), 151–153
percentages
calculations, 240
% (percent character), 1033
personal digital assistants (PDAs), 151–153
personalbar property, 375, 397
phantom page syndrome, cascading style sheets, 175
pipe character (|), 407
pitch property, 837
pitchRange property, 837
pixelBottom property, 827
pixelDepth property, BC102–BC103
pixelHeight property, 827
pixelLeft property, 827
pixelRight property, 827
pixels, coordinate points, 45
pixelTop property, 827
pixelWidth property, 827
pkcs11 property, 373
placeholders
creating, 236
data types, 60
discussed, 545
platform equivalency, DHTML, BC321–BC322
platform property, BC81
playDuring property, 837
plugin object, BC90–BC92
plug-ins
ActiveX controls, 6
audio and video, 6
controlling, BC263–BC267
detection, BC96–BC99
embedded media example, BC271–BC275

Flash, 6
helpers and, 5–6
installation, BC96
jukebox example, BC268–BC271
verifying, BC94
plugins property
discussed, 546–547
navigator object, BC81–BC82
pluginspace property, BC178
plus sign (+), 24, 66
pointer control, 1021
populating
arrays, 911
table cells, BC21–BC23
windows, 424–425
pop-up lists
color changes, 712
creating, 413
hiding and showing, 483–484
methods, list of, 482–484
properties, list of, 481–482
select object, 104–105
port property
a element object, 595
area element object, 624
overview, 494–495
posBottom property, 827
posHeight property, 827
position property, 827
positioning properties, style object, 826–828
POSITIVE_INFINITY property, 887
posLeft property, 827
posTop property, 827
posWidth property, 827
precaching images, 132–133
preference() method, BC84–BC87
preferences, browsers, reading and writing, BC85–BC86, BC887
prefix property, 183, 236
prefsDlog variable, 451
preventDefault() method, 191, 280
prevention, frames, 362–363
prevField() function, 221
previous property, 502–503
previousNode() method, 588–589
previousPage() method, BC47–BC48
previousSibling() method, 588
previousSibling property
discussed, 183–184
as generic object, 249
primitive targets, privileges, BC311
print() method, 426–428
printEvaluator() function, BC291–BC292
printFrames() function, 426
printing and page properties, style object, 835–836
privileges
gaining, BC310
manager errors, BC314–BC315
target specification, BC310–BC312
problem prevention, BC296–BC297
processData() function, 107
processing data, 107
product property, BC82
productSub property, BC82
profile property, BC5
programming
object-oriented, 185
scripting versus, 55–56
programs
back-end, 5
repetition in, 70
prompt dialog boxes, 88
prompt() method, 428–429
prompt property, BC8
prompter property, 369
propagation models, events, 190
properties. *See also* attributes
abbr, BC57
above, BC113
AbsolutePosition, 250
acceptCharset, 638
access to, 43
accessKey
discussed, 205–206
as generic object, 204–206
label object, 1033–1034

Continued

properties *(continued)*
activeElement, 513
adding to objects, 159
align
applet object, BC169
discussed, 476–477
embed object, BC178
hr object, 1030
Image object, 604–605
object object, BC173
table object, BC35
aLink, 578
alinkColor, 179, 514
all, 206–207
allowTransparency, 463, 477
alt
applet object, BC170
area element object, 623
Image object, 60
<meta> tag, 153
object object, BC173
altHTML
applet object, BC169
object object, BC173
altKey
IE4+ event object, 755
NN6+ event object, 775
altLeft, 756
anchorNode, 1063
anchorOffset, 1063
anchors, 516–517
appCodeName, BC69
appCore, 369
applets, 517–518
appMinorVersion, BC76
appName, 44, 90, BC69
appVersion, 44, 90, BC69
archive
applet object, BC170
security, BC307–BC308
areas, 625
arguments, 984–985
arity, 985
as arrays, 166
in arrays, 80
attribute, 207–209
attributes, 183
aural, style object, 836–837
autocomplete, 639
availHeight, BC100
availLeft, BC101
availTop, BC101–BC102
availWidth, BC100
axis, BC57
BackCompat, 519
background
layer object, BC116–BC117
overview, 579
table object, BC35
BaseHref, BC173
baseURI, BC173
behavior, 1035
behaviorCookie, 756
behaviorPart, 756
below, BC113
bgColor
body element object, 578–579
discussed, 179, 514, 518
document object, 179
layer object, BC117–BC118
marquee object, 1037
table object, BC36
bgcolor, 166
bgProperties, 579
body, 518
bookmarks, 756
border
discussed, 470
Image object, 606
object object, BC174
table object, BC36
border and edge, style object, 829–833
borderColor, 463, 471, BC36
borderColorDark, BC36
borderColorLight, BC36–BC37
bottom, 1102

bottomMargin, 580
boundElements, 756
boundingHeight, 1077
boundingLeft, 1077
boundingTop, 1077
boundingWidth, 1077–1079
box, BC39
browserLanguage, BC77
bubbles, 776
bufferDepth, BC102
button
  IE4+ event object, 757–758
  NN6+ event object, 776
caller, 985–986
cancelable, 776–777
cancelableFlag, 791
cancelBubble
  IE4+ event object, 758
  NN6+ event object, 77
canHaveChildren, 209–211
canHaveHTML, 211
caption, BC37
case-sensitivity, 166
cellIndex, BC58
cellPadding, BC37
cells
  table object, BC38
  tr object, BC54
cellSpacing, BC37–BC38
ch, BC49
characterSet, 519
charCode, 77, 778–779
charset
  a element object, 595
  discussed, 518–519
  link object, BC10
  meta object, BC13
checked
  checkbox input object, 658–659
  discussed, 102
  radio input object, 665–666
childNodeDetail, 190
childNodes, 183–184, 211–213
children, 213–214
chOff, BC49
cite, 1024
classid, BC174
className
  cascading style sheets, 138
  as generic object, 215–216
clear, 1025
clientHeight, 216–217
clientInformation, 370
clientLeft, 217–218
clientTop, 217–218
clientWidth, 216–217
clientX
  IE4+ event object, 759
  NN6+ event object, 780
clientY
  IE4+ event object, 759
  NN6+ event object, 780
clip, BC118–BC122
clipboardData, 370–371
closed, 370–371
code
  applet object, BC170
  object object, BC174
codeBase
  applet object, BC170
  object object, BC174
codeType, BC175
collapsed, 1043
color
  basefont object, BC7
  cascading style sheets, 138
  font object, 1026
  hr object, 1032
colorDepth, BC102
cols
  discussed, 471
  frames, 364
  table object, BC38
  textarea object, 692
colSpan, BC58

Continued

properties *(continued)*
commonAncestorContainer, 1043–1044
compact, BC61
compatMode, 519–520
complete
image input object, 670
Image object, 606–607
Components, 369, 372
constructor
Array object, 916
Boolean object, 890
Error object, 958
function object, 986
RegExp object, BC206
string object, 853–854
content, BC13
contentDocument, 463–464, 477, BC175
contentDocument, frames, 125
contentEditable, 218–219, 233
contentWindow, 464, 477
controllers, 369, 372
cookie, 520–521
cookieEnabled, BC78
coords, 595, 623
cpuClass, BC78–BC79
crypto, 373
cssRules, 799
cssText
cssRule object, 808
styleSheet object, 799–800
ctrlKey
IE4+ event object, 755
NN6+ event object, 775
ctrlLeft, 756
current, 502
currentNode, 587
currentStyle, 219
currentTarget, 783–785
cursor, 446
data
NN4 event object, 748–749
object object, BC175
Text object, 1069
dataFld, 219, 756
dataFormatAs, 219
dataPageSize, BC39
dataSrc, 219–223
dataTransfer, 763–765
defaultCharset, 529
defaultChecked
checkbox input object, 659
radio input object, 666
defaultStatus, 373–374
defaultValue, 677–678
defaultView, 529
defer, BC15
defined, 43
deprecated, BC4
description
discussed, 958
mimeType object, BC88
plugin object, BC92
designMode, 530
detail, 785
dialogArguments, 374
dialogHeight, 374
dialogLeft, 374
dialogTop, 374
dialogWidth, 374–375
dir, 225
direction, 1037
directories, 374
disabled
as generic object, 225–226, 234
link object, BC10
styleSheet object, 800
doctype, 530
document, 377–378, BC122
document object, accessing, 90
documentElement, 530–531
document.forms, 91
document.images, 91
domain, 531
dot notation, 43
dynamic, 320–321
dynsrc, 608–609
elements, 639–640
embeds, 531

enabledPlugin, BC88–BC89
encoding, 641
enctype, 99, 641
endContainer, 1044
endOffset, 1044
errors in, BC282
event, 378, BC15
event handlers as, 168–169
eventPhase, 785
events as, 168–169
expandEntityReference, 587
expando, 532
external, 378–379
face
  basefont object, BC7
  font object, 1027–1028
fgColor, 514, 532
fileCreatedDate, 532, 609
fileModifiedDate, 532, 609
fileName, 958
filename, BC92
fileSize, 532–534, 609–610
fileUpdatedDate, 609
filter, 587
filters, 226
firstChild, 183–184, 226–228
focusNode, 1063
focusOffset, 1063
font, style object, 815–816
fontSmoothingEnabled, BC103
fontStyle, 219
form
  button object, 653–654
  label object, 1034
  object object, BC175
  text object, 678
form.elements, 99
forms, 534–536
frame, BC39–BC41
frameBorder, 464–465, 474
frameElement, 379
frames, 379–381, 536–537
frameSpacing, 474
fromElement, 765
hash
  a element object, 595
  area element object, 624
  discussed, 488–489
headers, BC57
height
  applet object, BC170
  discussed, 465, 478, 537
  embed object, BC178
  as generic object, 228
  Image object, 610
  marquee object, 1037
  object object, BC175
  screen object, BC100
  table object, BC41
  tr object, BC54–BC55
hidden, BC178
hideFocus, 229
history, 382
host
  a element object, 595
  area element object, 624
  discussed, 489–492
hostname
  a element object, 595
  area element object, 624
  discussed, 492
href
  a element object, 595–596
  area element object, 624
  base object, BC6
  discussed, 89, 493–494
  Image object, 610
  link object, BC10
  location object, 89
  styleSheet object, 800
hreflang, 596, BC10
hsides, BC39
hspace
  applet object, BC170
  Image object, 610
  marquee object, 1037
  object object, BC175
  overview, 478

Continued

properties *(continued)*
htmlFor
label element object, 649
label object, 1034
script object, BC15
htmlText, 1079–1080
httpEquiv, BC14
id
as generic object, 230
styleSheet object, 800
ids, 537
ignoreCase, BC206
images, 537–538
implementation, 538–539
imports, 801
inline display and layout, 821–825
innerHeight, 382
innerHTML
Dynamic HTML, 139
as generic object, 230–232
read/write access, 174, 189
innerText
as generic object, 230–232
read/write access, 174
innerWidth, 382
input, BC210
isChar, 785
isCollapsed, 1063
isContentEditable, 219, 232–233
isDisabled, 233–234
isMap, 611
isMultiLine, 234
isOpen, 481–482
isTextEdit, 234
keyCode
IE4+ event object, 767–769
NN6+ event object, 777–779
label
optgroup object, 715–716
option object, 714
lang, 235
language
as generic object, 235
navigator object, BC79
lastChild
discussed, 183–184, 235
as generic object, 226–228
lastIndex, BC207
lastMatch, BC210
lastModified, 539
layers, 540–542
layerX
NN4 event object, 749
NN6+ event object, 780
layerY
NN4 event object, 749
NN6+ event object, 780
left
layer object, BC122
TextRectangle object, 1102
leftContext, BC211
leftMargin, 580
length
Array object, 917
arrays, 77, 143
form object, 641
function object, 986–987
as generic object, 235–236
history object, 503–504
plugin object, BC92
radio input object, 667
select object, 704
string object, 854
lhs, BC39
lineNumber, 958
link, 578, 580
linkColor, 179, 514, 542
links, 542
lists, style object, 833
loading, 384
localName, 183, 236
location, 385, 543
location object versus, 89
locationbar, 374, 385
location.hash, frames, 128
location.href, frames, 128
location.ref, BC344
longDesc, 465, 478, 611

loop
  Image object, 611
  marquee object, 1038
lowsrc, 612
marginHeight, 466, 478
marginWidth, 466, 478
Math object, 883–884
maxLength, 679
MAX_VALUE, 887
media
  link object, BC10–BC11
  overview, 545
  style element object, 797
  styleSheet object, 801
menubar, 374
message, 958
metaKey, 775
method, 642
Methods, 596
mimeType
  a element object, 596
  Image object, 612
  overview, 546
mimeTypes, BC79–BC80
MIN_VALUE, 887
modifiers, 752
multiline, BC207, BC210
multiple, 704
name
  a element object, 596–597
  applet object, BC171
  button object, 654
  discussed, 385, 466, 479
  embed object, BC178
  Error object, 959
  form object, 642
  Image object, 612–613
  layer object, BC124
  meta object, BC14
  object object, BC176
  plugin object, BC92
  radio input object, 666
  text object, 679–680
  window object, 385
nameProp
  a element object, 597
  Image object, 613
  overview, 546
namespaces, 546
namespaceURI, 183, 236
NaN, 887–888
naturalHeight, 613
naturalWidth, 613–614
navigator, 385
NEGATIVE_INFINITY, 887
newWindow variable, 90
next, 502
nextPage, 769
nextSibling, 183–184, 237
node objects, list of, 183
nodeName, 183–184, 237–238
nodeType, 184, 238–239
nodeValue, 183–184, 239–240
noResize, 466
noShade, 1032
noWrap, 580–581, BC59–BC60
number, 959
object
  applet object, BC171
  object object, BC176
object properties, 166
offscreenBuffering, 386
offset, 174
offsetHeight, 229, 240
offsetLeft, 241
offsetParent, 42–243
offsetTop, 241
offsetWidth, 229, 240–241
offsetX, 759
offsetY, 759
onerror, 386–388, 619
onLine, BC80
opener, window object, 128, 389–391
options, 705–708
originalTarget, 785–786
oscpu, BC80
outerHeight, 382, 392

Continued

properties *(continued)*
outerHTML
as generic object, 244
read/write access, 174
outerText
as generic object, 244–245
read/write access, 174
outerWidth, 382–384, 392
ownerDocument, 183, 245–246
ownerNode, 801
ownerRule, 802
owningElement, 802
page and printing, style object, 835–836
pages, 802–803
pageX
layer object, BC124
NN4 event object, 749
NN6+ event object, 780
pageXOffset, 392–394
pageY
layer object, BC124–BC128
NN4 event object, 749
NN6+ event object, 780
pageYOffset, 392–394
parent, 394–397
parentElement, 246
parentFrame, 379
parentLayer, BC128
parentNode, 183–184, 247
parentStyleSheet, 808
parentTextEdit, 247–249
parentWindow, 546
pathname
a element object, 595
area element object, 624
discussed, 494
personalbar, 374, 397
pixelDepth, BC102–BC103
pkcs11, 373
platform, BC81
plugins, 546–547, BC81–BC82
pluginspace, BC178
pop-up lists, 481–482
port
a element object, 595
area element object, 624
discussed, 494–495
positioning, style object, 826–828
POSITIVE_INFINITY, 887
prefix, 183, 236
previous, 502–503
previousSibling, 183–184, 249
product, BC82
productSub, BC82
profile, BC5
prompter, 369
propertyName, 769
protocol
a element object, 595
area element object, 624
discussed, 495, 547
Image object, 614
protocolLang, 597
prototype
Array object, 917–918
Boolean object, 890
function object, 987
Math object, 888
RegExp object, BC211
string object, 854–855
prototypes, 855
qualifier, 756
rangeCount, 1063
read-only, 190
readOnly
cssRule object, 808–809
styleSheet object, 803
text object, 680
read/write status, 166
readyState
as generic object, 249–250
xml object, BC186–BC187
reason, 756
recordNumber, 250–251
recordset, 756
referrer, 547–548

rel, 597, BC11
relatedTarget, 786–787
repeat, 769
replace, 499–500
responseText, BC187
responseXML, BC187
returnValue, 397, 770
rev, 597, BC11
rhs, BC39
right, 1102
rightMargin, 580–581
root, 587
rowIndex, BC55
rows
  discussed, 471–473
  frames, 364
  table object, BC42
  textarea object, 692–693
rowSpan, BC58–BC60
rules
  styleSheet object, 803–804
  table object, BC42–BC44
runtimeStyle, 251–252
saveType, 770
scopeName, 236, 252
screen, 397
screenLeft, 398
screenTop, 398
screenX
  discussed, 398
  IE4+ event object, 759
  NN4 event object, 749
  NN6+ event object, 780
screenY
  discussed, 398–399
  IE4+ event object, 759
  NN4 event object, 749–751
  NN6+ event object, 780
script-detectable, 171
scripts, 548–549
scroll, 581
scrollAmount, 1038
scrollbar, style object, 833
scrollbars, 374, 399
scrollDelay, 1038
scrollHeight, 252
scrolling, 467, 479
scrollLeft
  body element object, 581
  as generic object, 252–253
scrollMaxX, 399
scrollMaxY, 399
scrollTop
  body element object, 581–582
  as generic object, 253–254
scrollWidth, 252–253
scrollX, 399
scrollY, 399
search
  a element object, 595
  area element object, 624
  discussed, 495
sectionRowIndex, BC55
security, 549
securityPolicy, BC83
selectedIndex, 105, 709
selection, 549–550
selectorText, 808–809
self, 400–401
shape, 595, 623–624
sheet, BC11
shiftKey
  IE4+ event object, 755–756
  NN6+ event object, 775–776
shiftLeft, 756
siblingAbove, BC113, BC128
siblingBelow object, BC113–BC116, BC128
sidebar, 369–370, 401
size
  basefont object, BC7
  hr object, 1032
  select object, 710
  text object, 680
source, BC207
sourceIndex, 254–255
span, BC52

Continued

properties *(continued)*
src
discussed, 468, 479–480
embed object, BC179
image input object, 670
Image object, 614
<img> tag, 132
layer object, BC128–BC131
script object, BC16
xml object, BC184
srcElement, 770–772
srcFilter, 772
srcUrn, 772
start
Image object, 618
ol object, BC61
startContainer, 1044
startOffset, 1044–1045
status, 401–404, BC187
statusbar, 374, 404
statusText, BC188
style
cssRule object, 809
as generic object, 255
styleSheet, BC11
styleSheets, 550
subproperties, reading and writing, 838
suffixes, BC90
summary, BC44
systemLanguage, BC83
tabIndex, 255–257
table, style object, 834–835
tagName, 257–258
tags, 550
tagUrn, 236, 258
target
area element object, 624
base object, BC6
discussed, 598
form object property, 642–643
link object, BC11–BC12
NN4 event object, 752
NN6+ event object, 787–789
tBodies, BC44–BC45
text
a element object, 598
body element object, 578, 583
script object, BC16
select object, 105
TextRange object, 1080
title object, BC17
text, style object, 817–818, 820
textAlign, 175
tFoot, BC45
tHead, BC45
timeStamp, 789–790
title
discussed, 550
as generic property, 258–259
styleSheet object, 804
toElement, 765–767, 772
toolbar, 375–377, 404
top
layer object, BC122–BC124
overview, 404
TextRectangle object, 1102
topMargin, 580, 583
trueSpeed, 1038
type
a element object, 598
button object, 654
checkbox input object, 659
cssRule object, 809
defined, 43
IE4+ event object, 772–773
image input object, 670
li object, BC64
link object, BC12
mimeType object, BC89–BC90
NN4 event object, 753
object object, BC176
ol object, BC61–BC62
radio input object, 667
script object, BC16
select object, 710
selection object, 1064–1065

style element object, 797
styleSheet object, 804
text object, 681
ul object, 680
undoBuffer, 1055
uniqueID, 259–260
units, BC179
updateInterval, BC103
URL, 543–545, 551
URLUnencoded, 551
urn, 599
useMap
Image object, 618
object object, BC176
userAgent
location object, 90
navigator object, BC69, BC83
userLanguage, BC83
userProfile, BC83
vAlign, BC49
value
button object, 654
checkbox input object, 659–660
checkboxes, 102
defined, 43
forms, 100
li object, BC64
objects and, 166
radio input object, 667
select object, 710
text object, 681–683
vendor, BC84
vendorSub, BC82, BC84
version, BC4
visibility, BC131–BC132
visible, 376
vLink, 583
vlinkColor, 179, 514–516, 551
vsides, BC39
vspace
applet object, BC170–BC171
discussed, 478
Image object, 610–611, 618
marquee object, 1037
object object, BC175–BC176
whatToShow, 587
wheelData, 773
which, 753
width
applet object, BC170–BC171
colgroup object, BC52
discussed, 465, 478, 537, 551
embed object, BC178
as generic object, 228
hr object, 1032
Image object, 610, 618
marquee object, 1037
object object, BC175–BC176
screen object, BC100–BC101
table object, BC41, BC45
td object, BC60
window, 404
windows, accessing, 84–85
window.status, 87
wrap, 693
x
discussed, 599
IE4+ event object, 759
Image object, 618
XMLDocument, BC184
y
discussed, 599
IE4+ event object, 759–763
Image object, 618–619
zIndex
discussed, 175
layer object, BC133–BC136
propertyIsEnumerable() method, 1008
propertyName property, 769
protected property security example, BC313–BC314
protocol property
a element object, 595
area element object, 624
discussed, 495, 547
Image object, 614

protocolLang property, 597
prototypes
  CGI, 33
  objects, 159
  properties, 855
  prototype inheritance, 1003–1004
  prototype property
    Array object, 917–918
    Boolean object, 890
    function object, 987
    Math object, 888
    RegExp object, BC211
    string object, 854–855
pseudo-URLs, 137
public keyword, BC248

Q

qualifier property, 756
query commands, 569
queryCommandCommandState() method, 569
queryCommandEnabled() method, 569, 1099
queryCommandIndterm() method, 569
queryCommandState() method, 1099
queryCommandSupported() method, 569, 1099
queryCommandText() method, 569, 1099
queryCommandValue() method, 569–570, 1099
quoted pairs, strings, 111–112
quotes property, 817

R

radio objects
  discussed, 103–104
  input objects
    checked property, 665–666
    click() method, 668
    defaultChecked property, 666
    length property, 667
    name property, 667
    onclick event, 668–669
    overview, 663–664
    type property, 667
    value property, 667
random access memory (RAM), 36, 60
random() method, 115–116
random number generation, 115–116
RandomDissolve() method, 845
Range object
  cloneContents() method, 1046
  cloneRange() method, 1046
  collapse() method, 1047
  collapsed property, 1043
  commonAncestorContainer property, 1043–1044
  compareBoundaryPoints() method, 1047–1048
  comparePoint() method, 1051
  createContextualFragment() method, 1051–1052
  deleteContents() method, 1052–1053
  detach() method, 1053–1054
  endContainer property, 1044
  endOffset property, 1044
  extractContents() method, 1054
  insertNode() method, 1054–1055
  intersectsNode() method, 1056
  isPointInRange() method, 1056
  overview, 1039–1040
  selectNode() method, 1056
  selectNodeContents() method, 1056–1057
  setEnd() method, 1057
  setEndAfter() method, 1059
  setEndBefore() method, 1059
  setStart() method, 1057–1058
  setStartAfter() method, 1059
  setStartBefore() method, 1059
  startContainer property, 1044
  startOffset property, 1044–1045
  surroundContents() method, 1059–1060
  toString() method, 1061
rangeCount property, 1063
reading
  browser preference, BC85–BC86, BC887
  elements, 216
  strings, 869–870
read-only properties, 190

readOnly property
cssRule object, 808
isindex object, BC8
styleSheet object, 803
text object, 680
read/write status, properties
discussed, 166
innerHTML property, 174, 189
innerText property, 174
outerHTML property, 174
outerText property, 174
readyState property
as generic object, 249–250
xml object, BC186–BC187
real-time validation triggers, BC213–BC215
reason property, 756
recalc() method, 319, 570
recordNumber property, 250–251
recordset property, 756
rectangles
bounding, 289–291
styles, 814
recursion, in functions, 992
redirecting events, 728–730
references
FAQs (Frequently Asked Questions), 1140
layers, 172, BC111–BC112
referencing
events, 720
frames, 362
objects, 40
referrer pages, 548
referrer property, 547–548
Refresh command, documents, 563
refresh() method
plugin object, BC92
table object, BC48
RegExp object
compile() method, BC207
constructor property, BC206
exec() method, BC208
global property, BC206
ignoreCase property, BC206
input property, BC210
lastIndex property, BC207
lastMatch property, BC210
leftContext property, BC211
multiline property, BC207, BC210
prototype property, BC211
rightContext property, BC211
source property, BC207
test() method, BC208–BC209
regular expressions
backreferencing, BC196
discussed, 861–862
grouping, BC196
matching techniques, BC200–BC203
object relationships, BC196–BC200
patterns and, BC191–BC192
search-and-replace operations, BC203–BC205
special characters, BC193–BC196
string replacement, BC203–BC205
rel property, 597, BC11
relatedTarget property, 786–787
relative units, defined, 812
relativeNumber parameter, 506
release() function, BC452
releaseCapture() method, 302–303, 305–306
releaseEvents() method, 429–430, 570
reload, soft, 499
reload() method, 499–500
reloading, hard versus soft, 500
reloading issues, authoring environment setup, 20
remove() method, 710–711
removeAllRanges() method, 1067
removeAttribute() method, 209, 306–307
removeAttributeNode() method, 307
removeAttributeNS() method, 308
removeBehavior() method, 262–263, 308–309, BC332
removeChild() method, 186, 267, 309
removeEventListener() method, 191, 264–267, 309
removeExpression() method, 293, 309–310
RemoveFormat command, documents, 563

removeMember() function, BC258
removeNode() method, 310–312
removeRange() method, 1067
removeRule() method, 805–806
repeat loops, 71–72, 534
repeat property, 769
repetition, in programs, 70
replace property, 499
replaceAdjactentText() method, 298, 311–312
replaceChild() method, 186–188, 267, 312
replaceData() method, 1070
replaceNode() method, 312–315
replacing text, 311
reset event, 264
reset() method, 643–644
resetField() function, 677
resetTab() function, 257
resistors, defined, 30
resizable value, 446
resize event, 264
resizeBy() method, 430, BC140
resizeTo() method
  discussed, 430–432
  layer object, BC140–BC143
resizing layers, BC141–BC142, BC164–BC166
resizing windows, 383, 431–432
resources
  behaviors, BC340
  newsgroups, 1139–1140
  online documentation, 1140–1141
  Web sites, 1141
responses, events, 45
responseText property, BC187
responseXML property, BC187
restore() function, 310, 376
return (\r) character, 692
return values, arrays, 926
returning values, 85
returnValue property, 397, 770
rev property, 597, BC11
revealClip() function, BC147
revealTrans() method, 840
revertPrivilege() method, BC310
rhs property, BC39
richness property, 837
right property, 826, 1102
rightContext property, BC211
rightMargin property, 580–581
risks, privilege targets, BC311–BC312
rng variable, 1042
rollovers, mouse, 13
root nodes, 1043
root property, 587
rotating images, 616–617
routeEvent() method, 432–433, 571
RowDelim parameter, 220
rows property
  discussed, 471–473
  frames, 364
  table object, BC42
  textarea object, 692–693
rowSpan property, BC58–BC60
rows, tables
  deleting, BC46
  editing, BC25–BC29
  inserting, BC46–BC47
moving, BC47
rubyAlign property, 817
rubyOverhand property, 817
rubyPosition property, 817
rule object, 807–810
rules property
  styleSheet object, 803–804
  table object, BC42–BC44
running scripts, 51
runtime errors
  exception handling, 948–949
  syntax errors versus, BC277
runtimeStyle property, 251–252

S

Safari browser
  Debug menu, 55
  overview, 7–8
same origin security policy, BC301–BC303
same word entries, 43

sans serif fonts, 1028
saveCurrentVisit() function, BC418
saveType property, 770
saving cookies, 522
scopeName property, 236, 252
scr property, BC16
screen object
availHeight property, BC100
availLeft property, BC101
availTop property, BC101–BC102
availWidth property, BC100
bufferDepth property, BC102
colorDepth property, BC102
discussed, 397
fontSmoothingEnabled property, BC103
height property, BC100
overview, 36, BC99–BC100
pixelDepth property, BC102–BC103
roles of, 36
updateInterval property, BC103
width property, BC100–BC101
screenLeft property, 398
screenSize variable, BC101
screenTop property, 398
screenX property
IE4+ event object, 759
NN4 event object, 749
NN6+ event object, 780
overview, 398
screenY property
IE4+ event object, 759
NN4 event object, 749–751
NN6+ event object, 780
overview, 398–399
Script Debugger tool (Microsoft), BC290
script object, BC14–BC16
script-detectable properties, 171
scripters, BBEdit, 18
scripting techniques, frames, 125
scripts
in body, 49–50
browser sniffer, 152
client-side scripting, 8
compatibility issues, 145
copyright notifications, 147
deferred, 52–54, BC345
errors
controlling, 387–388
viewing, 54–55
functions and, 52
in headings, 48
hiding, 50–51, 146–147
in HTML documents, 144–146
immediate, BC345
in-document, 147
libraries, 147–148, 197
<noscript> tag, 148–149
programming versus, 55–56
running, 51
<script> tag
event attribute, 145, 176
for attribute, 145, 176
.htc document, 177
language attribute, 48, 145
overview, 22
src attribute, 47–48
type attribute, 47, 145
scripts property, 548–549
signed, 421–422
simple example of
errors, 24
overview, 20
source code, 21–22
text, displaying, 23–24
sniffer script, browers, 151–153
script-to-applet communication, BC257–BC258
scroll bar properties, style object, 834
scroll event, 264
scroll() method, 433–435
scroll property, 581
scroll values, 582
scrollAmount property, 1038
scrollbar3dLightColor property, 834
scrollbarArrowColor property, 834
scrollbarBaseColor property, 834
scrollbarDarkShadowColor property, 834
scrollbarFaceColor property, 834

scrollbarHighlightColor property, 834
scrollbars
  chrome elements, 368
  scrollbars property, 375, 399
  vertical, 253
scrollbarTrackColor property, 834
scrollBy() method, 435
scrollDelay property, 1038
scrollHeight property, 252
scrolling images, 434
scrolling property, 467, 479
scrollIntoView() method, 204, 206, 315
scrollJump variable, 441
scrollLeft property
  body element object, 581
  as generic object, 252–253
scrollMaxX property, 399
scrollMaxY property, 399
scrollMsg() function, 403
scrollShadowColor property, 834
scrollSpeed variable, 441
scrollTo() method, 435–437
scrollTop property
  body element object, 581–582
  as generic object, 253–254
scrollWidth property, 252–253
scrollX property, 399
scrollY property, 399
search property
  a element object, 595
  area element object, 624
  overview, 495
search-and-replace operations, regular expressions and, BC203–BC205
searches
  strings, 113–114
  Web traffic, 4
searchOnReturn() function, BC362
seconds, date functions, 117
security
  applets, BC300–BC301
  archive property, BC307–BC308
  codebase principal, BC306
  cookies, 523
  digital certificates, BC305–BC306
  id attribute, B308, BC309
  international characters, BC316
  JAR Packager, BC307
  origin checks, BC302
  privileges, gaining, BC310
  protected property example, BC313–BC314
  reasons for, BC299–BC300
  same origin policy, BC301–BC303
  security property, 549
  securityPolicy property, BC83
  signed script policy, BC304–BC305
  SignTool program, BC307
select event, 264
select() method
  text object, 684–685
  TextRange object, 1099
select object
  add() method, 710
  item() method, 711
  length property, 704
  multiple property, 704
  namedItem() method, 711
  navigating with, 105–106
  onchange event, 711–712
  options() method, 711
  options property, 705–708
  overview, 695–696
  pop-up lists, 104–105
  remove() method, 710–711
  selectedIndex property, 105, 709
  size property, 710
  text property, 105
  type property, 710
  value property, 710
SelectAll command, documents, 563
selectAllChildren() method, 1068
selectChunk() function, 247
selectedIndex property, 105, 709
selecting
  browsers, 18
  text editors, 17–18

selection lists, 706
selection object
  addRange() method, 1065
  anchorNode property, 1063
  anchorOffset property, 1063
  clear() method, 1065
  collapse() method, 1066
  collapseToEnd() method, 1066
  collapseToStart() method, 1066
  containsNode() method, 1066
  createRange() method, 1066
  deleteFromDocument() method, 1066
  empty() method, 1067
  extend() method, 1067
  focusNode property, 1063
  focusOffset property, 1063
  getRangeAt() method, 1067
  isCollapsed property, 1063
  overview, 1061–1062
  rangeCount property, 1063
  removeAllRanges() method, 1067
  removeRange() method, 1067
  selectAllChildren() method, 1068
  toString() method, 1068
  type property, 1064–1065
selection property, 549–550
selectNode() method, 1056
selectNodeContents() method, 1056–1057
selectorText property, 808–809
selectSort() function, BC467
self property, 400–401
self value, 85
send() method, BC189
serif fonts, 1028
server scripting, 5
serverless CGIs, 33
serverless databases, BC355
servers
  access, reducing, BC386
  busy servers, offloading, 33–34
setActive() method, 315–316
setAttribute() method
  discussed, 187, 207, 209
  generic objects, 316–317
setAttributeNode() method, 317
setAttributeNS() method, 318
setBGColor() function, BC328
setCapture() method
  discussed, 264
  generic objects, 302–306, 318
setClip() function, BC147
setColor() method, BC145
setCookie() function, BC427
setCookieData() function, BC417
setCount() function, 700
setdata() method, 330
setDate() method, 117
setDay() method, 117
setDocCapture() function, 724
setEnd() method, 1057
setEndAfter() method, 1059
setEndBefore() method, 1059
setEndPoint() method, 1100
setExpression() method
  discussed, 293
  generic objects, 318–321
setFontAttr() function, 1026
setHours() method, 117
setImagePosition() function, 242
setInitialColor() function, 262
setInnerPage() function, BC152
setInterval() method, 437–441
setLang() function, 700
setMember() method, BC258
setMinutes() method, 117
setMonth() method, 117
setMsg() function, 136
setOuterPage() function, BC152
setSeconds() method, 117
setSelected() function, 448
setSlot() method, BC258
setStart() method, 1057–1058
setStartAfter() method, 1059
setStartBefore() method, 1059
setTime() method, 117
setTimeout() method, 273, 371, 441–444
setupDrag() function, 335–336
setWinWidth() function, BC456

setYear() method, 117
setZIndex() function, BC328
SGML (Standard Generalized Markup Language), 4
shading effects, 1032
shadow() method, 839
shape property
  area element object, 623–624
  discussed, 595
shareware editors, TextPad, 18
shareware programs, JavaScript Scrambler, 147
sheet property, BC11
shiftBy() function, BC328
shiftKey property
  IE4+ event object, 755–756
  NN6+ event object, 775–776
shiftLeft property, 756
shiftTo() function, BC327
short-circuit evaluation, operators, 980
show() method, 482–484
showCountDown() function, 410
showHelp() method, 444
showModalDialog() method, 397, 444
showModelessDialog() method, 444–447
showOffsets() function, 393
showValues() function, BC119
siblingAbove property, BC113
siblingBelow property, BC113–BC116
siblings, parent nodes, 181
sidebar property, 369–370, 401
signed scripts
  exporting and importing, BC315
  overview, 421–422
  singed script security policy, BC304–BC305
SignTool security program, BC307
single quotes (‘), 87
sites. *See* Web sites
size property
  basefont object, BC7
  hr object, 1032
  select object, 710
  style object, 836
  text object, 680
sizeToContent() method, 455
slicing strings, 866–867
small data lookups, 29
smart Web pages, 34
snapshots
  Date object, 116
  debugging techniques, BC292
sniffer script, browsers, 151–153
soft cookies, BC414
soft reload, defined, 499
sorting data, 223–224
source code
  debugging, BC287
  frames, 365
  simple script example, 21–22
source property, BC207
sourceIndex property, 254–255
space in Web pages, saving, 23
spacer images, 5
span element
  cascading style sheets, 172
  overview, 138
span property, BC52
speak property, 837
speakHeader property, 837
speakNumeral property, 837
speakPunctuation property, 837
special characters, BC193–BC196
specification levels, DOM, 178
speechRate property, 837
splitText() method, 1073
spreadsheets, arrays and, 76
square value, BC63
src attribute, <script> tag, 47–48
src property
  embed object, BC179
  image input object, 670–671
  Image object, 614
  overview, 468, 479–480
  xml object, BC184
src property, <img> tag, 132
srcElement property, 770–772
srcFilter property, 772
srcUrn property, 772

stacking order, layers, 172, BC160–BC162
Standard Generalized Markup Language (SGML), 4
standards
  browsers, 10
  EMCA, 59
  EMCAScript, 144
start() method, 1038
start property
  Image object, 618
  ol object, BC61
startContainer property, 1044
starting applets, BC246–BC247
startOffset property, 1044–1045
startTimer() function, 410
startValue parameter, 72
statements
  assignment, 112
  behaviors and, BC333
  break, 79, 939–940
  comments, 51
  const keyword, 1017–1018
  debugging, BC289
  defined, 51
  executing, 51
  hiding, 146
  immediate, 52
  labeled, 944–946
  roles of, 51
  switch, 946–948
  uses for, 51
  variables as, 62
  with, 943
static events, 721
static filter types, 838–839
static objects, 189–190
static tables, calendars and, BC342–BC344
static text, 24
status bar messages, 258
status property
  discussed, 401–404
  xml object, BC187
statusbar property, 375, 404
statusText property, BC188
still images, switching to motion images, 608–609
stop() method
  marquee object, 1038
  overview, 455–456
stopping applets, BC246–BC247
stopPropagation() method, 736
stopTimer() function, 410
storage, cookies, BC379
stress property, 837
string object
  constructor property, 853–854
  length property, 854
  prototype property, 854–855
  string.charAt() method, 855–856
  string.charCodeAt() method, 856–857
  string.concat() method, 859
  string.indexOf() method, 859
  string.lastIndexOf() method, 860
  string.localeCompare() method, 860
  string.match() method, 861–862
  string.replace() method, 862–865
  string.search() method, 865
  string.slice() method, 865–867
  string.split() method, 868
  string.substr() method, 868–870
  string.substring() method, 870–871
  string.toLocaleLowerCase() method, 871
  string.toLocaleUpperCase() method, 871
  string.toLowerCase() method, 872
  string.toString() method, 872
  string.toUpperCase() method, 872
  string.valueOf() method, 872
string.charAt() method, 855–856
string.charCodeAt() method, 856–857
string.concat() method, 859
string.indexOf() method, 113–114
string.lastIndexOf() method, 860
string.localeCompare() method, 860
string.match() method, 861–862
string.replace() method, 862–865

strings. *See also* text
alert() method, 112
case, changing, 113
case-sensitivity, 292
character and substring copies, extracting, 114–115
character limitations, 112
comma-delimited, BC13
concatenation, 66, 112
converting
to numbers, 65, 881–882
numbers to, 65–66, 882
dates as, 898
defined, 111, 849
encoding/decoding, 876
formatting methods, 874–876
hexadecimal values and, 876
index values, 115
inline characters, 850–851
joining, 112–113, 850
length property, 115
literals, 112, 850
long variables, 850
manipulation, 873–874
naming, 134
nested, 73, 875–876
as parameters, 74
quoted pairs, 111–112
reading, 869–870
replacement, regular expressions and, BC203–BC205
searches, 113–114
slicing, 866–867
utility functions, 873–874
values to variables, accessing, 112
string.search() method, 865
string.slice() method, 865–867
string.split() method, 868
string.substr() method, 868–870
string.substring() method, 114–115, 870–871
string.toLocaleLowerCase() method, 871
string.toLocaleUpperCase() method, 871
string.toLowerCase() method, 872
string.toString() method, 872
string.toUpperCase() method, 872
string.valueOf() method, 872
stripZeros() function, BC360
style element object, 796–797
style property
cssRule object, 809
as generic object, 255
styleFloat property, 825
style-oriented tags, 113
styles, dynamic, 194
styleSheet object
addImport() method, 804–805
addRule() method, 805
cssRules property, 799
cssText property, 799–800
deleteRule() method, 806
disabled property, 800
href property, 800
id property, 800
imports property, 801
insertRule() method, 806–807
media property, 801
ownerNode property, 801
ownerRule property, 802
owningElement property, 802
pages property, 802–803
readOnly property, 803
removeRule() method, 805–806
rules property, 803–804
title property, 804
type property, 804
styleSheet property, BC11
styleSheets property, 550
stylesheets, system-specific, BC9
submission validations, 192
submit event, 264
submit() method, 108–109, 643–644
submitting forms, 636–637
subproperties, reading and writing, 838
substring() method, 524
substringData() method, 1070

subtract-by-value operator, 128
subtraction (-) operator, 66
subWindow variable, 85
subWrite() method, 93
suffixes property, BC90
summary property, BC44
Supported query command, 569
surroundContents() method, 1059–1060
swapNode() method, 312, 321
swapping images, 134
switch statement, 946–948
synchronization, frames, 364–365
syntax changes, 842–843
syntax versus runtime errors, BC277
system requirements, 1143
system-specific stylesheets, BC9
systemLanguage property, BC83

T

tabbing sequences, 256
tab-delimited data, 221
tabIndex property, 255–257
tabIndex value, 230
table object
  align property, BC35
  background property, BC35
  bgColor property, BC36
  border property, BC36
  borderColor property, BC36
  borderColorDark property, BC36
  borderColorLight property, BC36–BC37
  caption property, BC37
  cellPadding property, BC37
  cells property, BC38
  cellSpacing property, BC37–BC38
  cols property, BC38
  createCaption() method, BC45
  createTFoot() method, BC46
  createTHead() method, BC46
  dataPageSize property, BC39
  deleteCaption() method, BC45–BC46
  deleteRow() method, BC46
  deleteTFoot() method, BC46
  deleteTHead() method, BC46
  firstPage() method, BC47
  frame property, BC39–BC41
  height property, BC41
  insertRow() method, BC46–BC47
  lastPage() method, BC47
  moveRow() method, BC47
  nextPage() method, BC47
  previousPage() method, BC47–BC48
  refresh() method, BC48
  rows property, BC42
  rules property, BC42–BC44
  summary property, BC44
  tBodies property, BC44–BC45
  tFoot property, BC45
  tHead property, BC45
  width property, BC41, BC45
table of contents, 28–29
tableLayout property, 835
table-oriented containers, BC21
tables
  cells
    blank space between, BC38
    content, replacing, BC23–BC25
    data in, editing, BC22–BC25
    deleting, BC55
    inserting, BC55–BC56
    populating, BC21–BC23
  classes, BC33
  columns, editing, BC30–BC32
  dispatch lookup, BC228–BC229
  DHTML, BC348–BC352
  dynamic tables, calendars and, BC345–BC347
  footers, BC20
  hash tables, arrays and, 916
  headers, BC20
  hierarchy, BC19–BC21
  layers and, BC113
  modification methods, BC26
  properties, style object, 834–835

Continued

tables *(continued)*
rows
deleting, BC46
editing, BC25–BC29
inserting, BC46–BC47
moving, BC47
static tables, calendars and, BC342–BC344
truth, 969
Tabular Data Control (TDC), 220
`tagName` property, 257–258
tags
`<a>`, 137
`<area>`, 137
`<body>`, 168
`<br>`, 24
closing, 47
comment, 23
debugging, BC286–BC287
defined, 4
`<em>`, 38
`<form>`
`name` attribute, 91, 97
`onsubmit` event, 102
`<img>`
`height` attribute, 132
`src` property, 132
`width` attribute, 132
`<input>`
forms, 637–638
`type` attribute, 100
`<meta>`
`alt` attribute, 153
`content` attribute, 151–153
`<noscript>`, 148–149
`<p>`, 34, 38
positions, 48–50
`<script>`
`for` attribute, 145, 176
`event` attribute, 145, 176
`.htc` document, 177
`language`, 48
overview, 22
`src` attribute, 47–48
`type` attribute, 47, 145
style-oriented, 113
`tags()` method, 321–322
`tags` property, 550
`tagURN` property, 236
`tagUrn` property, 258
`taintEnabled()` method, BC87
`target` attribute
`form` object, 99
`window.open()` method, 85
`target` property
`area` element object, 624
`base` object, BC6
`form` object, 642–643
`link` object, BC11–BC12
NN4 event object, 752
NN6+ event object, 787–789
overview, 598
target specification, privileges, BC310–BC312
`tBodies` property, BC44–BC45
`tbody` object, BC48–BC50
`td` object, BC56–BC60
TDC (Tabular Data Control), 220
temporary cookies, BC414
`test()` method, BC208–BC209
testing
expression evaluation, 63
`OKToTest` flag, BC266
reasons for, BC297–BC298
`testValues()` function, 990–991
text. *See also* strings
bidirectional, 820
blinking, 34
boldface, 1045
capitalization, 138
color of, changing, 248
concatenation, 24
displaying, 23–24
fields, selecting, 684–685
height of, determining, 240
hidden, 150
highlighting, 1040
insertion pointers, 272
italics, 219
joining, 24

objects as, 28
paragraph, 38
passing to functions, 682
replacing, 311
retrieving selected, 566–567
static, 24
uppercase, 101
values, getting and setting, 681–682
width of, determining, 240
wrapping, BC59
text boxes
discussed, 219
Enter/Return key and, 676–677
text editors
selection considerations, 17–18
switching between, 19
text information, data as, 59
text nodes, 38–39
text object
blur() method, 683
defaultValue property, 677–678
focus() method, 683–684
form property, 678
maxLength property, 679
name property, 679–680
onafterupdate event, 685
onbeforeupdate event, 685
onblur event, 686
onchange event, 688
onerrorupdate() method, 685–686
onfocus event, 686
onselect event, 686–687
overview, 674
readOnly property, 680
select() method, 684–685
size property, 680
type property, 681
value property, 681–683
text property
a element object, 598
body element object, 578, 583
script object, BC16
select object, 105
style object, 817–820
TextRange object, 1080
title object, BC17
Text query command, 569
textAlign property, 175, 818
textAlignLast property, 818
textarea element, 100
textarea object, 690–693
textareas, carriage returns in, 692
textAutospace property, 818
textDecoration property, 818
textDecorationBlink property, 818
textDecorationLineThrough property, 818
textDecorationNone property, 818
textDecorationOverline property, 818
textDecorationUnderline property, 818
textIndent property, 818
textJustify property, 819
textJustifyTrim property, 819
TextNode object, 1069–1073
textOverflow property, 819
TextPad shareware editor, 18
TextQualifier parameter, 220
TextRange object
boundingHeight property, 1077
boundingLeft property, 1077
boundingTop property, 1077
boundingWidth property, 1077–1079
collapse() method, 1080–1081
compareEndPoints() method, 1081–1084
duplicate() method, 1084–1085
execCommand() method, 1085–1087
expand() method, 1087–1088
findText() method, 1088–1092
getBookmark() method, 1092
htmlText property, 1079–1080
inRange() method, 1093
isEqual() method, 1093–1094
move() method, 1094–1095
moveEnd() method, 1095
moveStart() method, 1095
moveToBookmark() method, 1095
moveToElementText() method, 1095
moveToPoint() method, 1096–1097

Continued

TextRange object *(continued)*
overview, 1074–1075
parentElement() method, 1097–1098
pasteHTML() method, 1098–1099
queryCommandEnabled() method, 1099
queryCommandIndeterm() method, 1099
queryCommandState() method, 1099
queryCommandSupported() method, 1099
queryCommandText() method, 1099
queryCommandValue() method, 1099
select() method, 1099
setEndPoint() method, 1100
text property, 1080
TextRectangle object, 1101–1103
text-related objects, 100–102
textShadow property, 819
textToWrite variable, 64
textTransform property, 819
textUnderlinePosition property, 819
tfoot object, BC48–BC50
tFoot property, BC45
th object, BC56–BC60
Thawte Digital Certificate Services Web site, BC305
thead object, BC48–BC50
tHead property, BC45
this keyword, 106
this operator, 975–976
this.form parameter, 633
throwaway passwords, 521
throwing exceptions, 953–957
time, date functions, 117
time intervals, Date object, 118
time stamps, 540
time zones, Date object, 891–892
timeIt() function, 336
timers, countdown, 409–410
timeStamp property, 789–790
timing problems, debugging, BC287–BC288
title object, BC16–BC17
title property
discussed, 550
as generic object, 258–259
styleSheet object, 804
toArray() method, 1022
today variable, Date object, 116
toElement property, 765–767, 772
toggle() function, BC380
toggleBar() function, 376
toggleColor() function, BC261
toggleComplete() function, 223
toggleDetails() function, 374
toolbar property, 375–377, 404
toolbars, chrome elements, 368
tooltips, 258
top object, 123
top property
layer object, BC122–BC124
overview, 404
style object, 826
TextRectangle object, 1102
top windows, 124
top-level objects, 37
topMargin property, 580, 583
toString() function, 1014–1016
toString() method
discussed, 959
function object, 988
Range object, 1061
selection object, 1068
toUpperCase() function, 101
tr object
cells property, BC54
deleteCell() method, BC55
height property, BC54–BC55
insertCell() method, BC55–BC56
overview, BC52–BC53
rowIndex property, BC55
sectionRowIndex property, BC55
trace() function, BC293–BC295
traffic
busy servers, offloading, 33–34
hit counts, 4
searches, 4
transition filters, 840
tree views, interfaces, 28

TreeWalker object
methods, 588–589
overview, 586
properties, 587
trigonometric functions, 115
troubleshooting. *See* debugging
true value
checked property, 102
comparison operators, 66
conditions, 70
discussed, 56
if constructions, 70
trueSpeed property, 1038
truth tables, 969
try construction, 950–952
turnOn() function, 262
two-dimensional arrays, BC356
type attribute
<input> tag, 100
<script> tag, 47, 145
type property
a element object, 598
button object, 654
checkbox input object, 659
cssRule object, 809
defined, 43
IE4+ event object, 772–773
image input object, 671
li object, BC64
link object, BC12
mimeType object, BC89–BC90
NN4 event object, 753
object object, BC176
ol object, BC61–BC62
radio input object, 667
script object, BC16
select object, 710
selection object, 1064–1065
style element object, 797
styleSheet object, 804
text object, 681
ul object, BC63
typeof operator, 977–978

U

ubound() method, 1022
UIEvents event, 556
ul object, BC62–BC63
unary operators, 966
UnBookmark command, documents, 563
undefined data type, 155
undefined errors, BC281–BC282
underline command, 1086
underscore (_), 991
undoBuffer property, 1055
unescape() function, 493, 1011–1012
unhighlight() function, 304
unicodeBidi property, 820
Uniform Resource Names (URNs), 209
uninitialized value, 249
uniqueID property, 259–260
units property, BC179
Unlink command, documents, 563
unload event, 264
unordered lists, 289, BC2
Unselect command, documents, 563
unwatch() function, 1016
update expressions, 72
updateInterval property, BC103
updateTime() function, 443
upgrades, browsers, 14–15
uppercase text, 101, 113
URL property, 543–545, 551
URLUnencoded property, 551
urn property, 599
urns() method, 322
URNs (Uniform Resource Names), 209
UseHeader parameter, 220
useMap property
Image object, 618
object object, BC176
user interfaces, interactive, 28
userAgent property, 90
location object, 90
navigator object, BC69
userLanguage property, BC83

userProfile object
addReadRequetst() method, BC105–BC106
clearRequest() method, BC106
data, accessing, BC104–BC105
doReadRequest() method, BC107–BC108
getAttribute() method, BC108
overview, BC103–BC104
userProfile property, BC83
utility functions, strings, 873–874

V

validations
batch mode, BC215
cross-confirmation fields, BC239–BC240
custom functions, BC221–BC222
data, 688
data entry, 33
date, BC224–BC227
dispatch lookup tables, BC228–BC229
forms, 30, 108–109
forms, Date object, 904–907
real-time triggers, BC213–BC215
simple example of, BC230–BC234
submission, 192
validation functions, combining, BC222
validDate() function, BC224
vAlign property, BC49–BC50
value property
button object, 654
checkbox input object, 659–660
checkboxes, 102
defined, 43
forms, 100
li object, BC64
objects and, 166
radio input object, 667
select object, 710
text object, 681–683
Value query command, 569
valueOf() method, 988
values
assigning to variables, 61
Boolean, 56
center, 446
circle, BC63
complete, 249
data types, 60
defined, 59
disc, BC63
edge, 446
false
checked property, 102
comparison operators, 66
conditions, 70
discussed, 56
else keyword, 71
if constructions, 70
interactive, 249
loaded, 249
loading, 249
null, 86
resizable, 446
returning, 85
self, 85
square, BC63
tabIndex, 230
true
checked property, 102
comparison operators, 66
conditions, 70
discussed, 56
if constructions, 70
uninitialized, 249
var keyword
discussed, 1018–1019
functions and, 74
local variables, 74
variables, creating, 61
variable scope
behaviors, BC333
functions, 989–992
variables
assigning, 64
branching, 194
creating, 61
currTitle, frames, 128
declaring, 61

defined, 60
expressions and, 64
foundArray, BC198
foundMatch, BC362
global, 74–75, 989
i, parallel arrays, 79
initializing, 61
local, 74–75
local scope, 75
long string, 850
naming, 61–62
newWind, 371
newWindow
location object, 90
window object, 86
output, 75
prefsDlog, 451
rng, 1042
screenSize, BC101
scrollJump, 441
scrollSpeed, 441
as statements, 62
subWindow, 85
textToWrite, 64
today, Date object, 116
values, assigning, 61
xDoc, BC397
VBArray object, 1021–1022
VBScript (Visual Basic Script) language, 7
vendor property, BC84
vendorSub property, BC82, BC84
Venkman debugger tool, BC290
verification
mime types, BC94
plug-ins, BC94–BC95
verifyData() function, BC296
Verisign Web site, BC305
version detection, browsers, 148–150
version implementation, JavaScript, 143–144
version numbering, browser compatibility issues, 11–12
version property, BC4
vertical scrollbars, 253
verticalAlign property, 825
video and audio plug-ins, 6
viewing errors, 54–55
visibility behaviors, layers, BC159–BC160
visibility property
discussed, 825
layer object, BC131–BC132
visible property, 376
Visual Basic Script (VBScript) language, 7
vLink property, 583
vlinkColor property, 179, 514–516, 551
voiceFamily property, 837
void keyword, BC245
volume property, 837
vsides property, BC39
vspace property
applet object, BC170–BC171
Image object, 610–611, 618
marquee object, 1037
object object, BC175
overview, 478

W

walkChildNodes() function, 212
warnings
access denied, 487
errors and, BC286
watch() function, 1016
wave() method, 839
Web pages
branching index pages, 151–153
compatibility issues, 155
file information, displaying, 533–534
navigating, using anchors, 517
referrer pages, 548
smart, 34
space in, saving, 23
Web sites
competitors, 4
resources, 1141
Thawte Digital Certificate Services, BC305
Venkman debugger tool, BC290
Verisign, BC305

Web traffic
busy servers, offloading, 33–34
hit counts, 4
searches, 4
weeks, date functions, 118
What You See Is What You Get (WYSIWYG) authoring tool, 17
whatToShow property, 587
wheelData property, 773
which property, 753
while loop, 940–942
white space, 160
whiteSpace property, 820
widgets, BC373
widows property, 835
width
of text, determining, 240
of windows, setting, 383–384
width attribute, <img> tag, 132
width property
applet object, BC170–BC171
colgroup object, BC52
discussed, 465, 478, 537, 551
embed object, BC178
as generic object, 228
hr object, 1032–1033
Image object, 610, 618
marquee object, 1037
object object, BC175–BC176
screen object, BC100–BC101
style object, 826
table object, BC41, BC45
td object, BC59–BC60
window object
alert() method, 45, 87–88, 404–405
appCore property, 369
back() method, 406
blur() method, 272
captureEvents() method, 406–408
clearInterval() method, 408–409
clearTimeout() method, 409–411
clientInformation property, 370
clipboardData, 370–371
close() method, 85–86, 411–412
closed property, 371–372
closeNewWindow() function, 86
Components property, 369, 372
confirm() method, 412–413
controllers property, 369, 372
createPopup() method, 413
crypto property, 373
defaultStatus property, 373–374
dialogArguments property, 374
dialogHeight property, 374
dialogLeft property, 375
dialogTop property, 375
dialogWidth property, 374–375
directories property, 375
document property, 377–378
DOM (document object model), 35
event property, 378
events and, 176
execScript() method, 414
external property, 378–379
find() method, 414–415
forward() method, 406, 415
frameElement property, 379
frames property, 379–381
fullScreen() function, 167
handleEvent() method, 415–416
history property, 382
home() method, 416
innerHeight property, 382
innerWidth property, 382
loading property, 384
location property, 385
locationbar property, 375, 385
makeNewWindow() function, 86
maximize() method, 167
menubar property, 375
moveBy() method, 416
moveTo() method, 416–419
name property, 385
navigate() method, 419–420
navigator property, 385
newWindow() function, 128
newWindow variable, 86
offscreenBuffering property, 386

onafterprint event, 456
onbeforeprint event, 456
onbeforeunload event, 456–457
ondragdrop event, 457–458
onerror event, 458
onerror property, 386–388
onhelp event, 458–459
onload event, 52, 89, 459–460
onmove event, 460
onresize event, 460–461
onunload event, 461
open() method, 420
opener property, 128, 389–391
openWindow() method, 164
outerHeight property, 382, 392
outerWidth property, 382–384, 392
overview, 36, 84, 368–369
pageXOffset property, 392
pageYOffset property, 392–394
parent property, 394–397
personalbar property, 375, 397
pkcs11 property, 373
print() method, 426–428
prompt() method, 428–429
prompter property, 369
releaseEvents() method, 429–430
resizeBy() method, 430
resizeTo() method, 430–432
returnValue property, 397
roles of, 36
routeEvent() method, 432–433
screen property, 397
screenLeft property, 398
screenTop property, 398
screenX property, 398
screenY property, 398–399
scroll() method, 433–435
scrollbars property, 375, 399
scrollBy() method, 435
scrollMaxX property, 399
scrollMaxY property, 399
scrollTo() method, 435–437
scrollX property, 399
scrollY property, 399
self property, 400–401
self value, 85
setInterval() method, 437–440
setTimeout() method, 441–444
showHelp() method, 444
showModalDialog() method, 444
showModelessDialog() method, 444–445
sidebar property, 369–370, 401
sizeToContent() method, 455
status property, 401–404
statusbar property, 375, 404
stop() method, 455–456
subWindow variable, 85
toolbar property, 375–377, 404
top property, 404
window property, 404
window.confirm() method, 88
window.open() method, 85, 128
window.prompt() method, 88–89
window.status property, 87
window.confirm() method, 88
windowFeatures parameter, 420–421
window.open() method, 85, 128
window.prompt() method, 88–89
windows
chrome, controlling, 376–377
click events, 408
closing, checking before, 372, 411–412
creating, 85–86, 425–426
height and width, setting, 383–384
loading, 384
main window document example, 129
moving, 417–419
naming, 385, 424
navigating, 385
pop-up, 413
populating, 424–425
properties and methods, accessing, 84–85
resizing, 383, 431–432
status bars, 404
subwindow document example, 129
top, 124

window.status property, 87
Winhelp file, 444
with statement, 943
wordBreak property, 820
wordSpacing property, 820
wordWrap property, 820
World Wide Web Consortium (W3C)
  browser compatibility issues, 13
  W3C DOM
    discussed, 177
    DOM content, adding/placing, 188
    DOM levels, 178
    element referencing, 180
    object-oriented, 185
    static objects, 189–190
wrap property, 693
wrapping text, BC59
write() method, 44, 571
writeln() method, 571–575
writing browser preferences, BC85–BC86, BC887
writingMode property, 820
W3C. *See* World Wide Web Consortium
WYSIWYG (What You See Is What You Get) authoring tool, 17

X

x property
  discussed, 599
  IE4+ event object, 759
  Image object, 618
xDoc variable, BC397
XML (eXtensible Markup Language), 5
xml object, BC181–BC184
XMLDocument property, BC184
XMLHttpRequest object, BC185–BC189
xRay() method, 839

Y

y property
  IE4+ event object, 759–763
  Image object, 618–619
  overview, 599
years, date functions, 117, 897

Z

zero-based index values, BC55
zIndex property
  layer object, BC133–BC136
  overview, 175
  style object, 828
zoom property, 826

Wiley Publishing, Inc.
End-User License Agreement

READ THIS. You should carefully read these terms and conditions before opening the software packet(s) included with this book "Book". This is a license agreement "Agreement" between you and Wiley Publishing, Inc. "WPI". By opening the accompanying software packet(s), you acknowledge that you have read and accept the following terms and conditions. If you do not agree and do not want to be bound by such terms and conditions, promptly return the Book and the unopened software packet(s) to the place you obtained them for a full refund.

1. **License Grant.** WPI grants to you (either an individual or entity) a nonexclusive license to use one copy of the enclosed software program(s) (collectively, the "Software") solely for your own personal or business purposes on a single computer (whether a standard computer or a workstation component of a multi-user network). The Software is in use on a computer when it is loaded into temporary memory (RAM) or installed into permanent memory (hard disk, CD-ROM, or other storage device). WPI reserves all rights not expressly granted herein.
2. **Ownership.** WPI is the owner of all right, title, and interest, including copyright, in and to the compilation of the Software recorded on the disk(s) or CD-ROM "Software Media". Copyright to the individual programs recorded on the Software Media is owned by the author or other authorized copyright owner of each program. Ownership of the Software and all proprietary rights relating thereto remain with WPI and its licensers.
3. **Restrictions on Use and Transfer.**
   - **(a)** You may only (i) make one copy of the Software for backup or archival purposes, or (ii) transfer the Software to a single hard disk, provided that you keep the original for backup or archival purposes. You may not (i) rent or lease the Software, (ii) copy or reproduce the Software through a LAN or other network system or through any computer subscriber system or bulletin-board system, or (iii) modify, adapt, or create derivative works based on the Software.
   - **(b)** You may not reverse engineer, decompile, or disassemble the Software. You may transfer the Software and user documentation on a permanent basis, provided that the transferee agrees to accept the terms and conditions of this Agreement and you retain no copies. If the Software is an update or has been updated, any transfer must include the most recent update and all prior versions.
4. **Restrictions on Use of Individual Programs.** You must follow the individual requirements and restrictions detailed for each individual program in the About the CD-ROM appendix of this Book. These limitations are also contained in the individual license agreements recorded on the Software Media. These limitations may include a requirement that after using the program for a specified period of time, the user must pay a registration fee or discontinue use. By opening the Software packet(s), you will be agreeing to abide by the licenses and restrictions for these individual programs that are detailed in the About the CD-ROM appendix and on the Software Media. None of the material on this Software Media or listed in this Book may ever be redistributed, in original or modified form, for commercial purposes.

5. **Limited Warranty.**

   (a) WPI warrants that the Software and Software Media are free from defects in materials and workmanship under normal use for a period of sixty (60) days from the date of purchase of this Book. If WPI receives notification within the warranty period of defects in materials or workmanship, WPI will replace the defective Software Media.

   (b) WPI AND THE AUTHOR(S) OF THE BOOK DISCLAIM ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SOFTWARE, THE PROGRAMS, THE SOURCE CODE CONTAINED THEREIN, AND/OR THE TECHNIQUES DESCRIBED IN THIS BOOK. WPI DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE SOFTWARE WILL BE ERROR FREE.

   (c) This limited warranty gives you specific legal rights, and you may have other rights that vary from jurisdiction to jurisdiction.

6. **Remedies.**

   (a) WPI's entire liability and your exclusive remedy for defects in materials and workmanship shall be limited to replacement of the Software Media, which may be returned to WPI with a copy of your receipt at the following address: Software Media Fulfillment Department, Attn.: *JavaScript Bible, 5th Ed.*, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, or call 1-800-762-2974. Please allow four to six weeks for delivery. This Limited Warranty is void if failure of the Software Media has resulted from accident, abuse, or misapplication. Any replacement Software Media will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer.

   (b) In no event shall WPI or the author be liable for any damages whatsoever (including without limitation damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising from the use of or inability to use the Book or the Software, even if WPI has been advised of the possibility of such damages.

   (c) Because some jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation or exclusion may not apply to you.

7. **U.S. Government Restricted Rights.** Use, duplication, or disclosure of the Software for or on behalf of the United States of America, its agencies and/or instrumentalities "U.S. Government" is subject to restrictions as stated in paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013, or subparagraphs (c) (1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, and in similar clauses in the NASA FAR supplement, as applicable.

8. **General.** This Agreement constitutes the entire understanding of the parties and revokes and supersedes all prior agreements, oral or written, between them and may not be modified or amended except in a writing signed by both parties hereto that specifically refers to this Agreement. This Agreement shall take precedence over any other documents that may be in conflict herewith. If any one or more provisions contained in this Agreement are held by any court or tribunal to be invalid, illegal, or otherwise unenforceable, each and every other provision shall remain in full force and effect.