

JAVASCRIPT ANTHOLOGY

101 ESSENTIAL TIPS, TRICKS & HACKS



BY JAMES EDWARDS & CAMERON ADAMS

The JavaScript Anthology 101 Essential Tips, Tricks & Hacks (4 Chapter Sample)

Thank you for downloading this four-chapter sample of James Edwards's and Cameron Adams's book, *The JavaScript Anthology:* 101 Essential Tips, Tricks & Hacks, published by SitePoint.

This excerpt includes the Summary of Contents, Information about the Authors, Editors and SitePoint, Table of Contents, Preface, four chapters of the book, and the index.

We hope you find this information useful in evaluating this book.

For more information or to order, visit sitepoint.com

Summary of Contents of this Excerpt

	Preface	xi
	1. Getting Started with JavaScript	1
,	5. Navigating the Document Object Model	79
	7. Working with Windows and Frames	127
	13. Basic Dynamic HTML	229
	Index	565
Sum	mary of Additional Book Contents	
	2. Working with Numbers	31
	3. Working with Strings	45
	4. Working with Arrays	65
	6. Processing and Validating Forms	103
	8. Working with Cookies	143
	9. Working with Dates and Times	151
	10. Working with Images	167
	11. Detecting Browser Differences	191
	12. Using JavaScript with CSS	201
	14. Time and Motion	267
	15. DHTML Menus and Navigation	321
	16. JavaScript and Accessibility	385
	17. Using JavaScript with Flash	457
	18. Building Web Applications with JavaScript	467
	19. Object Orientation in JavaScript	515
	20. Keeping up the Pace	565

The JavaScript Anthology 101 Essential Tips, Tricks & Hacks

by James Edwards and Cameron Adams

The JavaScript Anthology: 101 Essential Tips, Tricks & Hacks

by James Edwards and Cameron Adams

Copyright © 2006 SitePoint Pty. Ltd.

Expert Reviewer: Bobby van der Sluis
Expert Reviewer: Derek Featherstone
Managing Editor: Simon Mackie
Technical Editor: Kevin Yank

Editor: Georgina Laidlaw
Index Editor: Bill Johncocks
Cover Design: Jess Mason
Cover Layout: Alex Walker

Printing History:

First Edition: February 2006

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

424 Smith Street Collingwood VIC Australia 3066. Web: www.sitepoint.com Email: business@sitepoint.com

ISBN 0-9752402-6-9 Printed and bound in the United States of America

About the Authors

James Edwards (aka brothercake¹) is a freelance web developer based in the United Kingdom, specializing in advanced DHTML programming and accessible web site development. He is an outspoken advocate of standards-based development, a part-time forum moderator, and author of the Ultimate Drop Down Menu² system—the first commercial DHTML menu to be WCAG-compliant.

Cameron Adams has a degree in law and one in science; naturally he chose a career in web development. His business cards say, "Web Technologist" because he likes to have a hand in graphic design, JavaScript, CSS, PHP, and anything else that takes his fancy that morning. While running his own business—themaninblue.com³—he's consulted and worked for numerous government departments, nonprofit organizations, large corporations, and tiny startups. Cameron lives in Melbourne, Australia, where, between coding marathons, he likes to play soccer and mix some tunes for his irate neighbors.

About the Expert Reviewers

Bobby van der Sluis lives in the Netherlands and works at Blast Radius⁴ in Amsterdam, where he manages the interface development department. He's a client-side web technologies and design specialist, occasionally writing about these topics on his personal web site. Bobby is an evangelist of unobtrusive JavaScript, progressive enhancement, and the use of best practices, and has contributed to many notable sites, including A List Apart and CSS Zen Garden. He spends his scarce spare time with his wife Anita and newly-born daughter, Sofie.

Derek Featherstone is a well-known instructor, author, speaker, and developer with expertise in web accessibility consulting. Derek delivers technical training that is engaging, informative, and immediately applicable. A high-quality instructor, he draws on his background as a former high school teacher, plus seven years running his web development and accessibility consultancy Further Ahead. Derek blogs at boxofchocolates.ca. ⁷

About the Technical Editor

As Technical Director for SitePoint, Kevin Yank oversees all of its technical publications—books, articles, newsletters, and blogs. He has written over 50 articles for SitePoint, but is best known for his book, Build Your Own Database Driven Website Using PHP &

¹ http://www.brothercake.com/

² http://www.udm4.com/

³ http://themaninblue.com/

⁴ http://www.blastradius.com/

⁵ http://www.bobbyvandersluis.com/

⁶ http://www.furtherahead.com/

⁷ http://boxofchocolates.ca/

MySQL. Kevin lives in Melbourne, Australia, and enjoys performing improvised comedy theatre and flying light aircraft.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our books, newsletters, articles and community forums.

For Kizzy—I missed @media for all the right reasons.

—James

This is for Mum, Dad, Darren, and Davina, who gave me their love and support throughout the writing of this book, even though I had to explain it all using plasticine dinosaurs.

—Cameron



Table of Contents

Preface	xi
Who Should Read this Book?	xi
What's in this Book?	xii
The Book's Web Site	XV
The Code Archive	XV
Updates and Errata	XV
The SitePoint Forums	XV
The SitePoint Newsletters	xvi
Your Feedback	xvi
Acknowledgements	xvi
1. Getting Started with JavaScript	1
JavaScript Defined	
JavaScript's Limitations	
Security Restrictions	3
JavaScript Best Practices	5
Providing for Users who Don't Have JavaScript (Progressive Enhan	.ce-
ment)	5
Separating Content from Behavior (Unobtrusive Scripting)	
Using Braces and Semicolons (Consistent Coding Practice)	11
Adding a Script to a Page	12
Putting HTML Comments Around Code	13
The language Attribute	14
Getting Multiple Scripts to Work on the Same Page	14
Hiding JavaScript Source Code	
Debugging a Script	19
Understanding a Browser's Built-in Error Reporting	
Using alert	
Using try-catch	
Writing to the Page or Window	
Using an External Debugger	
Strict Warnings	
Summary	29
2. Working with Numbers	
Doing Math with JavaScript	
Rounding a Number to x Decimal Places	33
Creating and Constraining Random Numbers	
Converting a Number to a String	
Formatting Currency Values	38

Converting a String to a Number	39
Converting Numbers to Ordinals (-st, -nd, -rd, -th)	42
Summary	
3. Working with Strings	45
Including a Special Character in a String	
Transforming the Character Case of a String	
Encoding a URL	
Comparing Two Strings	
Finding a Substring within a String	
Splitting a String into Substrings	52
Creating a Regular Expression	
Testing whether a String Matches a Regular Expression	
Testing whether a String Contains Only Numeric Data	
Testing whether a String is a Valid Phone Number	
Testing whether a String is a Valid Email Address	
Searching and Replacing Text using a Regular Expression	
Summary	
4. Working with Arrays	65
Using Array-literals	
Creating an Array of Arrays	66
Indexing an Array with Strings Instead of Numbers	
Turning an Array into a String	
Adding or Removing Members from an Array	
Sorting an Array into Alphabetical or Numeric Order	
Sorting a Multi-dimensional Array	
Sorting an Array Randomly	
Summary	
5. Navigating the Document Object Model	79
Accessing Elements	82
Creating Elements and Text Nodes	
Changing the Type of an Element	
Removing an Element or Text Node	
Reading and Writing the Attributes of an Element	
Getting all Elements with a Particular Attribute Value	
Adding and Removing Multiple Classes to/from an Element	
Summary	
6. Processing and Validating Forms 1	103
Reading and Writing the Data in a Text Field	
Reading and Setting the State of a Checkbox	

	Reading and Setting the State of a Radio Button	109
	Reading and Setting the Value of a Select Box	111
	Validating a Mandatory Text Field	113
	Validating a Numeric Field	114
	Validating an Email Address Field	115
	Checking for Unselected Radio Buttons	115
	Stopping a Form Being Submitted Unless all its Fields are Valid	116
	Validating a Form with an Unknown Number of Fields	
	Printing Inline Error Messages when Validating a Form	
	Making Form Fields Appear or Disappear, Based on the Value of	
	other Fields	121
	Summary	125
7 14	Orking with Windows and Frames	127
/ . V	Using Popup Windows	127 128
	What's Wrong with Popups?	
	How Do I Minimize the Problems?	120
	Opening Off-site Links in a New Window	
	Communicating Between Frames	
	Getting the Scrolling Position	
	Making the Page Scroll to a Particular Position	
	Getting the Viewport Size (the Available Space inside the Win-	140
	dow)dow	141
	Summary	
	Sullillary	172
8. W	Orking with Cookies	143
	Writing Cookies	143
	Reading a Cookie	145
	Setting a Cookie to Expire at a Specific Date and Time	146
	Making a Cookie Accessible Only from a Specific Domain or	
	Path	147
	Circumventing Browser Restrictions on the Number of Cookies you	
	can Use	148
	Summary	150
9. W	Orking with Dates and Times	151
J. V.	Getting the Date and Time	
	Formatting a Date into a Sentence	
	Formatting the Time into a 12- or 24-hour Clock	
	Comparing Two Dates	
	Formatting the Difference Between Dates	
	Summary	

10. Working with Images	167
Preloading Images	167
Swapping One Image for Another	169
Displaying an Image at Random	
Making a Slideshow of Several Images	
Making an Image Fade in or out	
Making an Image-based Clock that Updates in Real Time	181
Making a Progress Indicator	186
Summary	
11. Detecting Browser Differences	191
Identifying Support for a Particular Feature	
Identifying a Particular Browser	
Detecting Quirks Mode and Standards Mode	
Summary	
12. Using JavaScript with CSS	
Changing the Style of a Single Element	201
Changing the Style of a Group of Elements	
Retrieving the Computed Style of an Element	
Making a Style Sheet Switcher	
Maintaining Alternate Style Sheet States	
Making a Style Sheet Switcher that Handles Multiple Media	212
Types	215
Reading and Modifying an Existing Style Sheet	217
Adding New Style Sheet Rules	220
Deleting a Rule from a Style Sheet	
Creating a New Style Sheet	224
Summary	
13. Basic Dynamic HTML	
Handling Events	
The Short Way: Using Event Handlers	
The W3C Way (Event Listeners)	233
Finding the Size of an Element	245
Finding the Position of an Element	
Detecting the Position of the Mouse Cursor	
Displaying a Tooltip when you Mouse Over an Element	
Sorting Tables by Column	
Summary	266
14. Time and Motion	
Using setTimeout and setInterval	267

	Making an Object Move Along a Set Path	
	Making Animation Less Jerky	278
	Animation Frame Times	279
	Changing Between Frames	279
	Complexity of the Animation	280
	The Speed of the Computer	
	The Speed of the Browser	
	Implementing Drag-and-drop Behavior	
	Reordering a List Using Drag-and-drop Functionality	
	Making a Scrolling News Ticker	
	Creating Clip-based Transition Effects	
	Making a Slider Control	
	Summary	
15.	DHTML Menus and Navigation	
	Making a Drop-down or Fly-out Menu	
	Adding Arrows to Indicate the Presence of a Submenu	
	Adding Timers so the Menus Don't Open and Close so Abruptly	338
	Making Sure the Menus Stay Inside the Window	
	Making the Menus Display Over select Elements	
	Making a Folder Tree or Expanding Menu	
	Indicating Expanded Branches in a Menu	
	Allowing Only One Menu Branch to Be Open at Any Time	
	Opening the Current Sub-branch Automatically	
	Summary	383
16	avaScript and Accessibility	205
10.	Is JavaScript Inaccessible?	386
	What is Accessibility?	386
	Who are the Affected Users?	
	Making Scripts Accessible to the Keyboard	
	Using Device-independent Event Handlers	
	Making Scripts Accessible to the Keyboard as well as the Mouse	
	Rollovers and Revealing Content	396
	Form Validation	
	Drag-and-drop Functionality	
	AJAX and other Remote Scripting Techniques	
	Making title Attribute Tooltips Display on Focus	
	Making a DHTML Menu Accessible to the Keyboard	
	Making a DHTML Menu Usable via the Keyboard	
	Making a DHTML Slider Control Accessible to the Keyboard	
	Making Scripts Accessible to Screen Readers	

JavaScript Behaviors	438
Tricks and Hacks	449
Towards Best Practice	453
Summary	
17. Using JavaScript with Flash	457
Detecting whether Flash is Installed in a Browser	
Communicating Between JavaScript and Flash	
FSCommand	
Flash/JavaScript Integration Kit	464
Summary	
18. Building Web Applications with JavaScript	467
Retrieving Data Using XMLHttpRequest	
Requesting Data from a Server	470
Parsing the Data	
Caching	
AJAX Frameworks	
Retrieving Data without Using XMLHttpRequest	
Creating Custom Dialogs (Such as Popup Forms)	
Creating Editable Elements	489
Controlling Text Selections	
Creating an Auto-complete Text Field	502
Summary	
19. Object Orientation in JavaScript	515
What's so Good about Object Orientation?	
Abstraction	
Encapsulation	516
Class Inheritance	
Polymorphism	518
Object Based Code vs Object Oriented Code	518
Writing an Object Oriented Script	519
Creating Methods for an Object	521
Prototype-based Method Creation	522
Modelling Ínheritance	526
Understanding Scope	528
Implementing Namespaces	
Summary	
20. Keeping up the Pace	535
Making Scripts Run Faster	536
Saving References to Objects you Use Frequently	

Using Ternary Operators and Switch Statements	539
Optimizing Loops	542
Avoiding eval	
Avoiding Strict Warnings	544
Optimizing for a Particular Browser	545
Writing Scripts Using Less Code	
Dividing Tasks into Functions (or Using OO)	548
Using Arrays and Iteration to Avoid Code Repetition	550
Compacting Conditions and Return Statements	551
Optimizing Scripts for the Web	552
Removing Comments and Unnecessary Whitespace	
Compacting the Names of Variables and Properties	555
Avoiding Memory Leaks	
Avoiding Circular References	557
Cleaning Up After the Fact	558
Making Scripts Run Before the Load Event	560
Summary	563
Index	

Preface

To many people, the word "JavaScript" conjures up memories of annoying popups, irritating mouse-trails, and frustrating no-right-click scripts. If you've ever been on the receiving end of such a script, you'll know how tedious they can be. Yet JavaScript is a mature, professional scripting language that's used on the majority of modern web sites, and is a key component in almost all web-based applications. Hang on! Are we talking about the same technology here?

As with so many histories, both perceptions are reasonably accurate: JavaScript does have a dubious reputation, which it earned mainly in the first dot com boom when it was used for little else than opening popups, shielding code from casual scrutiny, and adding pointless whizz-bang effects. And in recent years, as both the web development community and the world at large have become more aware of accessibility issues, JavaScript has been singled out as a cause of many problems, though in reality, it's not the technology itself that's at fault—it's the poorly planned and careless use that has given JavaScript this reputation.

Yet with the increasing popularity of remote scripting techniques (popularly referred to as "AJAX"), JavaScript is enjoying something of a renaissance. Designers, developers, and programmers from many different disciplines are becoming interested in—and impressed by—what was once the domain of specialists. Browser vendors and other technology companies are taking another look at the potential of this powerful language, as the line between the Web and the desktop becomes increasingly blurred.

JavaScript is a key component in the development of a raft of new applications, and there's never been a better time to take an interest in it.

Who Should Read this Book?

Anyone who's involved or interested in building web sites or web applications should read this book.

If you're a webmaster looking for copy-and-paste solutions to everyday needs, we have those solutions for you. If you're already an experienced JavaScript programmer, you'll find in this book scripts and discussions that sit on the bleeding edge of current practice. If you're a designer with an interest in the coding side of things, or a student who's just beginning to get into it, you'll find many rich and beautiful examples to give you insight and ideas.

Whatever your current JavaScript knowledge, we hope you'll find this book a useful and inspirational resource for modern, best practice scripting.

What's in this Book?

Chapter 1: Getting Started With JavaScript

This chapter, which is slightly more theoretical than the rest, provides an overview of JavaScript's capabilities and limitations, and introduces some core best practices that we'll be using through the rest of the book. It's not a beginners' tutorial, nor a ground-up summary of the language, but it focuses on finding the best ways to perform basic tasks, including practical solutions for the problems that are encountered as we try to make scripts work together.

Chapter 2: Working with Numbers

This chapter looks at techniques for using and processing numbers in JavaS-cript. It covers basic computation, number rounding, the generation and constraint of random numbers, and the use of currency values, ordinals, and other formatted numbers.

Chapter 3: Working with Strings

Text is the meat and drink of the Web, and processing text is one of the most common tasks in web scripting. This chapter looks at ways of manipulating strings to find information, store data, and prepare text for output, and includes a thorough introduction to regular expressions in JavaScript.

Chapter 4: Working with Arrays

This chapter introduces one the most powerful data-storage structures in JavaScript: the array. We'll talk about reading and writing data from an array, sorting and processing arrays, and using multidimensional arrays. We'll also discuss a similar data structure: the object literal.

Chapter 5: Navigating the Document Object Model

The DOM is an interface for manipulating individual parts of a document. This chapter introduces and explores the DOM, and looks at how to create and read the data from elements, attributes, and text.

Chapter 6: Processing and Validating Forms

In this chapter, we look at reading and writing data from different kinds of form widget, address the tasks of validating and processing form data, and discuss techniques for improving the usability of form-based interfaces.

Chapter 7: Working with Windows and Frames

This chapter takes a cautious look at manipulating windows and scripting across frames. These are the most controversial parts of the language, as they have the potential to create serious usability and accessibility barriers, so this chapter is centered firmly on techniques that try to avoid or alleviate these problems.

Chapter 8: Working with Cookies

Cookies are the simplest and most reliable method for maintaining statepersistence in JavaScript—they allow pages and applications to "remember" who you are and what you're doing. In this chapter, we introduce cookies and show you how to use them effectively.

Chapter 9: Working with Dates and Times

It won't win any prizes for glamour, but this chapter shows you how to get the date and time in JavaScript, how to compare and process dates and times, and how to output the final data in different formats and conventions.

Chapter 10: Working with Images

Images are an important part of most web designs, and this chapter explores the basic techniques involved in scripting for them. We move from simple tasks like preloading, randomly selecting, swapping, and cross-fading images, to more complex slide show, progress indicator, and image-based clock scripts.

Chapter 11: Detecting Browser Differences

This short chapter outlines techniques for dealing with different browsers and rendering modes. In it, we explain when and where it's appropriate to use browser detection and object detection, and how you can combine these techniques to get the most robust information.

Chapter 12: Using JavaScript with CSS

In this chapter, we look at how to read and write the styles from a single element or group of elements, how to read and write CSS rules to an existing or created style sheet, and how to build a style sheet switcher.

Chapter 13: Basic Dynamic HTML

DHTML uses HTML, the DOM, and CSS to bring static content to life, and although the term DHTML is disparaged in some quarters, we still believe it's a useful and relevant way of describing this kind of scripting. In this chapter, we cover event-handling in all its flavors, detecting the position and size of an object, tracking the mouse, and making elements appear and dis-

appear. We'll also begin to look at rearranging the DOM dynamically with a neat table-sorting script.

Chapter 14: Time and Motion

This chapter advances the ideas from Chapter 13 into more complex forms of scripting that use motion and animation. We'll look at timers in JavaScript, and learn how to use them for both simple and more sophisticated animations. We'll also cover drag-and-drop functionality, and put it to use selecting and sorting information, as well as creating scrollers, sliders, and transition effects.

Chapter 15: DHTML Menus and Navigation

This chapter enters the complex arena of DHTML menus with two major scripts—a drop-down or fly-out menu, and a folder tree or expanding menu. For each menu, we'll create a core navigation structure using clean, semantic code. Then, we'll improve on each script with usability and accessibility enhancements, including submenu indicator arrows, open and close timers, and automatic repositioning (so that a menu never runs off the page's edge). This chapter also includes solutions for the problem of menus overlapping select elements in Windows IE 5 and IE 6.

Chapter 16: JavaScript and Accessibility

This chapter provides an overview of the current state of play regarding JavaScript and accessibility. It's focused on ideas and techniques for making scripts accessible to the keyboard, and also touches on how scripting may impact on people with learning or cognitive disabilities. We'll also examine a range of different scripts, including AJAX applications, to see how they behave with screen readers.

Chapter 17: Using JavaScript with Flash

In this chapter, we look at the narrow alliance between these two technologies, learning to detect whether a user has the Flash plugin, and mastering communication between JavaScript and Flash.

Chapter 18: Building Web Applications with JavaScript

This chapter delves into the exciting area of online application design, including data retrieval using XMLHttpRequest, as well as the older technique of using iframes. We'll also talk about creating custom dialogs, building editable elements like rich-text entry fields, and controlling and creating text selections to generate an auto-complete search field.

Chapter 19: Object Orientation in JavaScript

Object oriented programming is generally considered the best approach to large-scale programming projects, and in this chapter we introduce OOP, exploring its core concepts and benefits. We'll cover the practical techniques involved in creating an object oriented or object based script, and we'll talk about scope, inheritance, and object namespacing.

Chapter 20: Keeping up the Pace

The final chapter looks at everyday techniques for writing faster, more efficient code that's shorter and uses less memory. We'll also cover more brutal techniques for optimizing and obfuscating production code, but with the warning that some optimizations are more trouble than they're worth!

The Book's Web Site

Located at http://www.sitepoint.com/books/jsant1, the web site supporting this book will give you access to the following facilities.

The Code Archive

As you progress through the text, you'll note a number of references to the code archive. This is a downloadable ZIP archive that contains complete code for all the examples presented in this book. You can grab it on the book's web site at http://www.sitepoint.com/books/jsantl/code.php.

Updates and Errata

The Errata page on the book's web site will always have the latest information about known typographical and code errors, and necessary updates for changes to technologies. Visit it at http://www.sitepoint.com/books/jsantl/errata.php.

The SitePoint Forums

While we've made every attempt to anticipate any questions you may have, and answer them in this book, there's no way that *any* book could teach you everything you'll ever need to know about using JavaScript in your web development projects. If you have a question about anything in this book, the best place to go for a quick answer is http://www.sitepoint.com/forums/—SitePoint's vibrant and knowledgeable community.

The SitePoint Newsletters

In addition to books like this one, SitePoint offers free email newsletters.

The SitePoint Tech Times covers the latest news, product releases, trends, tips, and techniques for all technical aspects of web development. The long-running SitePoint Tribune is a biweekly digest of the business and moneymaking aspects of the Web. Whether you're a freelance developer looking for tips to score that dream contract, or a marketer striving to keep abreast of changes to the major search engines, this is the newsletter for you. The SitePoint Design View is a monthly compilation of the best in web design. From new CSS layout methods to subtle Photoshop techniques, SitePoint's chief designer shares his years of experience in its pages.

Browse the archives or sign up to any of SitePoint's free newsletters at http://www.sitepoint.com/newsletter/.

Your Feedback

If you can't find an answer through the forums, or you wish to contact us for any other reason, the best place to write is books@sitepoint.com. We have a well-manned email support system set up to track your inquiries, and if our support staff are unable to answer your question, they send it straight to us. Suggestions for improvement, as well as notices of any mistakes you may find, are especially welcome.

Acknowledgements

I'd like to thank all those who helped and supported me while writing this book, particularly to Eddie and Debi, Jon and Kim, who provided as much encouragement as they did practical support. I'd also like to thank Dave Evans, a significant influence from my early days as a developer.

—James Edwards

1

Getting Started with JavaScript

As we hope to demonstrate in many practical solutions throughout this book, JavaScript is an amazingly useful language that offers many unique benefits. With a little consideration for how scripted functionality degrades, you can use Java-Script to bring a whole range of functional, design and usability improvements to your web sites.

Let's begin with an introduction to JavaScript, exploring what it's for, and how we can use it.

JavaScript Defined

JavaScript is a scripting language that's used to add interactivity and dynamic behaviors to web pages and applications. JavaScript can interact with other components of a web page, such as HTML and CSS, to make them change in real time, or respond to user events.

You'll undoubtedly have seen JavaScript in the source code of web pages. It might have been inline code in an HTML element, like this:

```
<a href="page.html" onclick="open('page.html'); return false;">
```

It might have appeared as a script element linking to another file:

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

Or it may have had code directly inside it:

```
<script type="text/javascript">
function saySomething(message)
{
   alert(message);
}
saySomething('Hello world!');
</script>
```

Don't worry about the differences between these snippets yet. There are quite a few ways—both good and bad—in which we can add JavaScript to a web page. We'll look at these approaches in detail later in this chapter.

JavaScript was developed by Netscape and implemented in Netscape 2, although it was originally called LiveScript. The growing popularity of another language, Java, prompted Netscape to change the name in an attempt to cash in on the connection, as JavaScript provided the ability to communicate between the browser and a Java applet.

But as the language was developed both by Netscape, in its original form, and by Microsoft, in the similar-but-different JScript implementation, it became clear that web scripting was too important to be left to the wolves of vendor competition. So, in 1996, development was handed over to an international standards body called ECMA, and JavaScript became ECMAScript or ECMA-262.

Most people still refer to it as JavaScript, and this can be a cause of confusion: apart from the name and similarities in syntax, Java and JavaScript are nothing alike.

JavaScript's Limitations

JavaScript is most commonly used as a **client-side language**, and in this case the "client" refers to the end-user's web browser, in which JavaScript is interpreted and run. This distinguishes it from **server-side** languages like PHP and ASP, which run on the server and send static data to the client.

Since JavaScript does not have access to the server environment, there are many tasks that, while trivial when executed in PHP, simply cannot be achieved with JavaScript: reading and writing to a database, for example, or creating text files. But since JavaScript *does* have access to the client environment, it can make de-

cisions based on data that server-side languages simply don't have, such as the position of the mouse, or the rendered size of an element.



What About ActiveX?

If you're already quite familiar with Microsoft's JScript, you might be thinking "but JavaScript can do some of these things using ActiveX," and that's true—but ActiveX is not part of ECMAScript. ActiveX is a Windowsspecific mechanism for allowing Internet Explorer to access COM (the Component Object Model at the heart of Windows scripting technology) and generally only runs in trusted environments, such as an intranet. There are some specific exceptions we'll come across—examples of ActiveX controls that run without special security in IE (such as the Flash plugin, and XMLHttpRequest)—but for the most part, scripting using ActiveX is outside the scope of this book.

Usually, the computer on which a client is run will not be as powerful as a server, so JavaScript is not the best tool for doing large amounts of data processing. But the immediacy of data processing on the client makes this option attractive for small amounts of processing, as a response can be received straight away; form validation, for instance, makes a good candidate for client-side processing.

But to compare server-side and client-side languages with a view to which is "better" is misguided. Neither is better—they're tools for different jobs, and the functional crossover between them is small. However, increased interactions between client-side and server-side scripting are giving rise to a new generation of web scripting, which uses technologies such as XMLHttpRequest to make requests for server data, run server-side scripts, and then manage the results on the client side. We'll be looking into these technologies in depth in Chapter 18.

Security Restrictions

As JavaScript operates within the realm of highly sensitive data and programs, its capabilities have been restricted to ensure that it can't be used maliciously. As such, there are many things that JavaScript simply is not allowed to do. For example, it cannot read most system settings from your computer, interact directly with your hardware, or cause programs to run.

Also, some specific interactions that would normally be allowed for a particular element are not permitted within JavaScript, because of that element's properties. For example, changing the value of a form <input> is usually no problem, but if it's a file input field (e.g., <input type="file">), writing to it is not allowed at

all—a restriction that prevents malicious scripts from making users upload a file they didn't choose.

There are quite a few examples of similar security restrictions, which we'll expand on as they arise in the applications we'll cover in this book. But to summarize, here's a list of JavaScript's major limitations and security restrictions, including those we've already seen. JavaScript cannot:

open and read files directly (except under specific circumstances, as detailed in Chapter 18).
create or edit files on the user's computer (except cookies, which are discussed in Chapter 8).
read HTTP POST data.
read system settings, or any other data from the user's computer that is not made available through language or host objects. ¹
modify the value of a file input field.
alter a the display of a document that was loaded from a different domain.
close or modify the toolbars and other elements of a window that was not opened by script (i.e., the main browser window).

Ultimately, JavaScript might not be supported at all.

It's also worth bearing in mind that many browsers include options that allow greater precision than simply enabling or disabling JavaScript. For example, Opera includes options to disallow scripts from closing windows, moving windows, writing to the status bar, receiving right-clicks ... the list goes on. There's little you can do to work around this, but mostly, you won't need to—such options have evolved to suppress "annoying" scripts (status bar scrollers, no-right-click scripts, etc.) so if you stay away from those kinds of scripts, the issue will come up only rarely.

¹Host objects are things like **Window** and **Screen**, which are provided by the environment rather than the language itself.

JavaScript Best Practices

JavaScript best practices place a strong emphasis on the question of what you should do for people whose browsers don't support scripting, who have scripting turned off, or who are unable to interact with the script for another reason (e.g., the user makes use of an assistive technology that does not support scripting).

That final issue is the most difficult to address, and we'll be focusing on solutions to this problem in Chapter 16. In this section, I'd like to look at three core principles of good JavaScript:

progressive enhancement providing for users who don't have JavaScript

unobtrusive scripting separating content from behavior

consistent coding practice using braces and semicolon terminators

The first principle ensures that we're thinking about the bigger picture whenever we use a script on our site. The second point makes for easier maintenance on our end, and better usability and **graceful degradation**² for the user. The third principle makes code easier to read and maintain.

Providing for Users who Don't Have JavaScript (Progressive Enhancement)

There are several reasons why users might not have JavaScript:

- ☐ They're using a device that doesn't support scripting at all, or supports it in a limited way.
- ☐ They're behind a proxy server or firewall that filters out JavaScript.
- ☐ They have JavaScript switched off deliberately.

The first point covers a surprisingly large and ever-growing range of devices, including small-screen devices like PDAs, mid-screen devices including WebTV

²Graceful degradation means that if JavaScript is not supported, the browser can naturally fall back on, or "degrade" to, non-scripted functionality.

and the Sony PSP, as well as legacy JavaScript browsers such as Opera 5 and Netscape 4.

The last point in the list above is arguably the least likely (apart from other developers playing devil's advocate!), but the reasons aren't all that important: some users simply don't have JavaScript, and we should accommodate them. There's no way to quantify the numbers of users who fall into this category, because detecting JavaScript support from the server is notoriously unreliable, but the figures I've seen put the proportion of users who have JavaScript switched off between 5% and 20%, depending on whether you describe search engine robots as "users."

Solution

The long-standing approach to this issue is to use the HTML noscript element, the contents of which are rendered by browsers that don't support the script element at all, and browsers that support it but have scripting turned off.

Although it's a sound idea, in practice this solution has become less useful over time, because noscript cannot differentiate by capability. A browser that offers limited JavaScript support is not going to be able to run a complicated script, but such devices are script-capable browsers, so they won't parse the noscript element either. These browsers would end up with nothing.

A better approach to this issue is to begin with static HTML, then use scripting to modify or add dynamic behaviors within that static content.

Let's look at a simple example. The preferred technique for making DHTML menus uses an unordered list as the main menu structure. We'll be devoting the whole of Chapter 15 to this subject, but this short example illustrates the point:

```
    <a href="/">Home</a>
    <a href="/about/">About</a>
    <a href="/contact/">Contact</a>

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

The list of links is plain HTML, so it exists for all users, whether or not they have scripting enabled. If scripting *is* supported, our menu. js script can apply dynamic behaviors, but if scripting isn't supported, the content still appears. We haven't differentiated between devices explicitly—we've just provided content that's dynamic if the browser can handle it, and static if not.

Discussion

The "traditional" approach to this scenario would be to generate a separate, dynamic menu in pure JavaScript, and to have fallback static content inside a noscript element:

```
<script type="text/javascript" src="menu.js"></script>
<noscript>

     <a href="/">Home</a>
     <a href="/about/">About</a>
     <a href="/contact/">Contact</a>

</
```

But, as we've already seen, a wide range of devices will fall though this net, because JavaScript support is no longer an all-or-nothing proposition. The above approach provides default content to *all* devices, and applies scripted functionality only if it works.

This scripting approach is popularly referred to as **progressive enhancement**, and it's a methodology we'll be using throughout this book.



Don't Ask!

Neither this technique nor the noscript element should be used to add a message that reads, "Please turn on JavaScript to continue." At best, such a message is presumptuous ("Why should I?"); at worst it may be unhelpful ("I can't!") or meaningless ("What's JavaScript?"). Just like those splash pages that say, "Please upgrade your browser," these messages are as useful to the average web user as a road sign that reads, "Please use a different car."

Occasionally, you may be faced with a situation in which equivalent functionality simply *cannot* be provided without JavaScript. In such cases, I think it's okay to have a static message that informs the user of this incompatibility (in nontechnical terms, of course). But, for the most part, try to avoid providing this kind of message unless it's literally the only way.

Separating Content from Behavior (Unobtrusive Scripting)

Separating content from behavior means keeping different aspects of a web page's construction apart. Jeffrey Zeldman famously refers to this as the "three-legged stool" of web development³—comprising content (HTML), presentation (CSS), and behavior (JavaScript)—which emphasizes not just the difference in each aspect's functioning, but also the fact that they should be separated from one another.

Good separation makes for sites that are easier to maintain, are more accessible, and degrade well in older or lower-spec browsers.

Solution

At one extreme, which is directly opposed to the ideal of separating content from behavior, we can write inline code directly inside attribute event handlers. This is very messy, and generally should be avoided:

```
<div id="content"
  onmouseover="this.style.borderColor='red'"
  onmouseout="this.style.borderColor='black'">
```

We can improve the situation by taking the code that does the work and abstracting it into a **function**:

```
<div id="content"
  onmouseover="changeBorder('red')"
  onmouseout="changeBorder('black')">
```

Defining a function to do the work for us lets us provide most of our code in a separate JavaScript file:

```
File: separate-content-behaviors.js (excerpt)

function changeBorder(element, to)

{
    element.style.borderColor = to;
}
```

³Zeldman, J. Designing with Web Standards. New Riders, 2003.

But a much better approach is to avoid using inline event handlers completely. Instead, we can make use of the Document Object Model (DOM) to bind the event handlers to elements in the HTML document. The DOM is a standard programming interface by which languages like JavaScript can access the contents of HTML documents, removing the need for any JavaScript code to appear in the HTML document itself. In this example, our HTML code would look like the following:

```
<div id="content">
```

Here's the scripting we'd use:

```
file: separate-content-behaviors.js

function changeBorder(element, to)
{
    element.style.borderColor = to;
}

var contentDiv = document.getElementById('content');

contentDiv.onmouseover = function()
{
    changeBorder('red');
};

contentDiv.onmouseout = function()
{
    changeBorder('black');
};
```

This approach allows us to add, remove, or change event handlers without having to edit the HTML, and since the document itself does not rely on or refer to the scripting at all, browsers that don't understand JavaScript will not be affected by it. This solution also provides the benefits of reusability, because we can bind the same functions to other elements as needed, without having to edit the HTML.

This solution hinges on our ability to access elements through the DOM, which we'll cover in depth in Chapter 5.



The Benefits of Separation

By practicing good separation of content and behavior, we gain not only a practical benefit in terms of smoother degradation, but also the advantage of *thinking* in terms of separation. Since we've separated the HTML and

JavaScript, instead of combining them, when we look at the HTML we're less likely to forget that its core function should be to describe the *content* of the page, independent of any scripting.

Andy Clarke refers to the **web standards trifle**, ⁴ which is a useful analogy, A trifle looks the way a good web site should: when you look at the bowl, you can see all the separate layers that make up the dessert. The opposite of this might be a fruit cake: when you look at the cake, you can't tell what each different ingredient is. All you can see is a mass of cake.

Discussion

It's important to note that when you bind an event handler to an element like this, you can't do it until the element actually exists. If you put the preceding script in the head section of a page as it is, it would report errors and fail to work, because the content div has not been rendered at the point at which the script is processed.

The most direct solution is to put the code inside a load event handler. It will always be safe there because the load event doesn't fire until after the document has been fully rendered:

```
window.onload = function()
{
  var contentDiv = document.getElementById('content');
  :
};
```

Or more clearly, with a bit more typing:

```
window.onload = init;
function init()
{
  var contentDiv = document.getElementById('content');
  :
}
```

The problem with the load event handler is that only one script on a page can use it; if two or more scripts attempt to install load event handlers, each script will override the handler of the one that came before it. The solution to this

⁴ http://www.stuffandnonsense.co.uk/archives/web_standards_trifle.html

problem is to respond to the load event in a more modern way; we'll look at this shortly, in "Getting Multiple Scripts to Work on the Same Page".

Using Braces and Semicolons (Consistent Coding Practice)

In many JavaScript operations, braces and semicolons are optional, so is there any value to including them when they're not essential?

Solution

Although braces and semicolons are often optional, you should always include them. This makes code easier to read—by others, and by yourself in future—and helps you avoid problems as you reuse and reorganize the code in your scripts (which will often render an optional semicolon essential).

For example, this code is perfectly valid:

```
File: semicolons-braces.js (excerpt) if (something) alert('something') else alert('nothing')
```

This code is valid thanks to a process in the JavaScript interpreter called **semi-colon insertion**. Whenever the interpreter finds two code fragments that are separated by one or more line breaks, and those fragments wouldn't make sense if they were on a single line, the interpreter treats them as though a semicolon existed between them. By a similar mechanism, the braces that normally surround the code to be executed in **if-else** statements may be inferred from the syntax, even though they're not present. Think of this process as the interpreter adding the missing code elements for you.

Even though these code elements are not always necessary, it's easier to remember to use them when they *are* required, and easier to read the resulting code, if you do use them consistently.

Our example above would be better written like this:

```
File: semicolons-braces.js (excerpt) if (something) { alert('something'); } else { alert('nothing'); }
```

This version represents the ultimate in code readability:

```
File: semicolons-braces.js (excerpt)
if (something)
{
    alert('something');
}
else
{
    alert('nothing');
}
```



Using Function Literals

As you become experienced with the intricacies of the JavaScript language, it will become common for you to use **function literals** to create anonymous functions as needed, and assign them to JavaScript variables and object properties. In this context, the function definition should be followed by a semicolon, which terminates the variable assignment:

```
var saySomething = function(message)
{
    :
};
```

Adding a Script to a Page

Before a script can begin doing exciting things, you have to load it into a web page. There are two techniques for doing this, one of which is distinctly better than the other.

Solution

The first and most direct technique is to write code directly inside a **script** element, as we've seen before:

```
<script type="text/javascript">
function saySomething(message)
{
   alert(message);
}
```

```
saySomething('Hello world!');
</script>
```

The problem with this method is that in legacy and text-only browsers—those that don't support the script element at all—the contents may be rendered as literal text.

A better alternative, which avoids this problem, is always to put the script in an external JavaScript file. Here's what that looks like:

```
<script type="text/javascript" src="what-is-javascript.js"
></script>
```

This loads an external JavaScript file named what-is-javascript.js. The file should contain the code that you would otherwise put inside the script element, like this:

```
File: what-is-javascript.js

function saySomething(message)
{
    alert(message);
}

saySomething('Hello world!');
```

When you use this method, browsers that don't understand the script element will ignore it and render no contents (since the element is empty), but browsers that do understand it will load and process the script. This helps to keep scripting and content separate, and is far more easily maintained—you can use the same script on multiple pages without having to maintain copies of the code in multiple documents.

Discussion

You may question the recommendation of not using code directly inside the script element. "No problem," you might say. "I'll just put HTML comments around it." Well, I'd have to disagree with that: using HTML comments to "hide" code is a very bad habit that we should avoid falling into.

Putting HTML Comments Around Code

A validating parser is not required to read comments, much less to process them. The fact that commented JavaScript works at all is an anachronism—a throwback

to an old, outdated practice that makes an assumption about the document that might not be true: it assumes that the page is served to a non-validating parser.

All the examples in this book are provided in HTML (as opposed to XHTML), so this assumption is reasonable, but if you're working with XHTML (correctly served with a MIME type of application/xhtml+xml), the comments in your code may be discarded by a validating XML parser before the document is processed by the browser, in which case commented scripts will no longer work *at all*. For the sake of ensuring forwards compatibility (and the associated benefits to your own coding habits as much as to individual projects), I strongly recommend that you avoid putting comments around code in this way. Your JavaScript should *always* be housed in external JavaScript files.

The language Attribute

The language attribute is no longer necessary. In the days when Netscape 4 and its contemporaries were the dominant browsers, the <script> tag's language attribute had the role of sniffing for up-level support (for example, by specifying javascript1.3), and impacted on small aspects of the way the script interpreter worked.

But specifying a version of JavaScript is pretty meaningless now that JavaScript is ECMAScript, and the language attribute has been deprecated in favor of the type attribute. This attribute specifies the MIME type of included files, such as scripts and style sheets, and is the only one you need to use:

<script type="text/javascript">

Technically, the value should be text/ecmascript, but Internet Explorer doesn't understand that. Personally, I'd be happier if it did, simply because javascript is (ironically) a word I have great difficulty typing—I've lost count of the number of times a script failure occurred because I'd typed type="text/javsacript".

Getting Multiple Scripts to Work on the Same Page

When multiple scripts don't work together, it's almost always because the scripts want to assign event handlers for the same event on a given element. Since each element can have only one handler for each event, the scripts override one another's event handlers.

Solution

The usual suspect is the window object's load event handler, because only one script on a page can use this event; if two or more scripts are using it, the last one will override those that came before it.

We could call multiple functions from inside a single load handler, like this:

```
window.onload = function()
{
   firstFunction();
   secondFunction();
}
```

But, if we used this code, we'd be tied to a single piece of code from which we'd have to do everything we needed to at load time. A better solution would provide a means of adding load event handlers that *don't conflict with other handlers*.

When the following single function is called, it will allow us to assign *any number* of load event handlers, without any of them conflicting:

```
file: add-load-listener.js

function addLoadListener(fn)
{
   if (typeof window.addEventListener != 'undefined')
   {
      window.addEventListener('load', fn, false);
   else if (typeof document.addEventListener != 'undefined')
   {
      document.addEventListener('load', fn, false);
   else if (typeof window.attachEvent != 'undefined')
   {
      window.attachEvent('onload', fn);
   }
   else
   {
      var oldfn = window.onload;
      if (typeof window.onload != 'function')
      {
            window.onload = fn;
      }
      else
      {
            window.onload = function()
```

```
{
    oldfn();
    fn();
    };
}
```

Once this function is in place, we can use it any number of times:

```
addLoadListener(firstFunction);
addLoadListener(secondFunction);
addLoadListener(twentyThirdFunction);
```

You get the idea!

Discussion

JavaScript includes methods for adding (and removing) **event listeners**, which operate much like event handlers, but allow multiple listeners to subscribe to a single event on an element. Unfortunately, the syntax for event listeners is completely different in Internet Explorer than it is in other browsers: where IE uses a proprietary method, others implement the W3C Standard. We'll come across this dichotomy frequently, and we'll discuss it in detail in Chapter 13.

The W3C standard method is called addEventListener:

```
window.addEventListener('load', firstFunction, false);
```

The IE method is called attachEvent:

```
window.attachEvent('onload', firstFunction);
```

As you can see, the standard construct takes the name of the event (without the "on" prefix), followed by the function that's to be called when the event occurs, and an argument that controls event bubbling (see Chapter 13 for more details on this). The IE method takes the event *handler* name (*including* the "on" prefix), followed by the name of the function.

To put these together, we need to add some tests to check for the existence of each method before we try to use it. We can do this using the JavaScript operator typeof, which identifies different types of data (as "string", "number", "boolean", "object", "array", "function", or "undefined"). A method that doesn't exist will return "undefined".

```
if (typeof window.addEventListener != 'undefined')
{
    : window.addEventListener is supported
}
```

There's one additional complication: in Opera, the load event that can trigger multiple event listeners comes from the document object, not the window. But we can't just use document because that doesn't work in older Mozilla browsers (such as Netscape 6). To plot a route through these quirks we need to test for window.addEventListener, then document.addEventListener, then window.attachEvent, in that order.

Finally, for browsers that don't support any of those methods (Mac IE 5, in practice), the fallback solution is to **chain** multiple old-style event handlers together so they'll get called in turn when the event occurs. We do this by dynamically constructing a new event handler that calls any existing handler before it calls the newly-assigned handler when the event occurs.⁵

```
File: add-load-listener.js (excerpt)
var oldfn = window.onload;
if (typeof window.onload != 'function')
{
    window.onload = fn;
}
else
{
    window.onload = function()
    {
        oldfn();
        fn();
    };
}
```

Don't worry if you don't understand the specifics of how this works—we'll explore the techniques involved in much greater detail in Chapter 13. There, we'll learn that event listeners are useful not just for the load event, but for *any* kind of event-driven script.

⁵This technique was pioneered by Simon Willison [http://www.sitepoint.com/blogs/2004/05/26/closures-and-executing-javascript-on-page-load/].

Hiding JavaScript Source Code

If you've ever created something that you're proud of, you'll understand the desire to protect your intellectual property. But JavaScript on the Web is an open-source language by nature; it comes to the browser in its source form, so if the browser can run it, a person can read it.

There are a few applications on the Web that claim to offer source-code encryption, but in reality, there's nothing you can do to encrypt source-code that another coder couldn't decrypt in seconds. In fact, some of these programs actually cause problems: they often reformat code in such a way as to make it slower, less efficient, or just plain broken. My advice? Stay away from them like the plague.

But still, the desire to hide code remains. There is something that you can do to **obfuscate**, if not outright encrypt, the code that your users can see.

Solution

Code that has been stripped of all comments and unnecessary whitespace is very difficult to read, and as you might expect, extracting individual bits of functionality from such code is extremely difficult. The simple technique of compressing your scripts in this way can put-off all but the most determined hacker. For example, take this code:

```
File: obfuscate-code.js (excerpt)
var oldfn = window.onload;
if (typeof window.onload != 'function')
{
    window.onload = fn;
}
else
{
    window.onload = function()
    {
        oldfn();
        fn();
    };
}
```

We can compress that code into the following two lines simply by removing unnecessary whitespace:

```
File: obfuscate-code.js (excerpt)
```

```
var oldfn=window.onload;if(typeof window.onload!='function'){
window.onload=fn;}else{window.onload=function(){oldfn();fn();};}
```

However, remember that important word—unnecessary. Some whitespace is essential, such as the single spaces after var and typeof.

Discussion

This practice has advantages quite apart from the benefits of obfuscation. Scripts that are stripped of comments and unnecessary whitespace are smaller; therefore, they're faster loading, and may process more quickly.

But please do remember that the code must remain strictly formatted using semicolon line terminators and braces (as we discussed in "Using Braces and Semicolons (Consistent Coding Practice)"); otherwise, the removal of line breaks will make lines of code run together, and ultimately cause errors.

Before you start compression, remember to make a copy of the script. I know it seems obvious, but I've made this mistake plenty of times, and it's all the more galling for being so elementary! What I do these days is write and maintain scripts in their fully spaced and commented form, then run them through a bunch of search/replace expressions just before they're published. Usually, I keep two copies of a script, named myscript.js and myscript-commented.js, or something similar.

We'll come back to this subject in Chapter 20, where we'll discuss this among a range of techniques for improving the speed and efficiency of scripts, as well as reducing the amount of physical space they require.

Debugging a Script

Debugging is the process of finding and (hopefully) fixing bugs. Most browsers have some kind of bug reporting built in, and a couple of external debuggers are also worth investigating.

Understanding a Browser's Built-in Error Reporting

Opera, Mozilla browsers (such as Firefox), and Internet Explorer all have decent bug reporting functionality built in, but Opera and Mozilla's debugging tools are the most useful.

Opera

Open the JavaScript console from Tools > Advanced > JavaScript console. You can also set it to open automatically when an error occurs by going to Tools > Preferences > Advanced > Content, then clicking the JavaScript options button to open its dialog, and checking Open JavaScript console on error.

Firefox and other Mozilla browsers

Open the JavaScript console from Tools > JavaScript console.

Internet Explorer for Windows

Go to Tools > Internet Options > Advanced and uncheck the option Disable script debugging, then check the option Display a notification about every script error, to make a dialog pop up whenever an error occurs.

Internet Explorer for Mac

Go to Explorer > Preferences > Web Browser > Web Content and check the Show scripting error alerts option.

Safari doesn't include bug reporting by default, but recent versions have a "secret" Debug menu, including a JavaScript console, which you can enable by entering the following Terminal command:⁶

\$ defaults write com.apple.safari IncludeDebugMenu -bool true

You can also use an extension called Safari Enhancer, ⁷ which includes an option to dump JavaScript messages to the Mac OS Console; however, these messages are not very helpful.

Understanding the various browsers' console messages can take a little practice, because each browser gives such different information. Here's an example of an error—a mistyped function call:

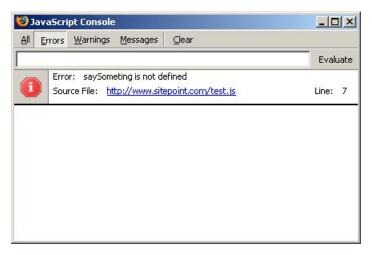
http://www.lordofthecows.com/safari_enhancer.php

⁶The \$ represents the command prompt, and is not to be typed.

```
function saySomething(message)
{
    :
    alert(message);
}
saySometing('Hello world');
```

Firefox gives a concise but very accurate report, which includes the line number at which the error occurred, and a description, as shown in Figure 1.1.





As Figure 1.2 illustrates, Opera gives an extremely verbose report, including a backtrace to the event from which the error originated, a notification of the line where it occurred, and a description.

A backtrace helps when an error occurs in code that was originally called by other code; for example, where an event-handler calls a function that goes on to call a second function, and it's at this point that the error occurs. Opera's console will trace this process back through each stage to its originating event or call.

Internet Explorer gives the fairly basic kind of report shown in Figure 1.3. It provides the number of the line at which the interpreter encountered the error (this may or may not be close to the true location of the actual problem),⁸ plus

⁸Internet Explorer is particularly bad at locating errors in external JavaScript files. Often, the line number it will report as the error location will actually be the number of the line at which the script is loaded in the HTML file.

a summary of the error type, though it doesn't explain the specifics of the error itself.

Figure 1.2. The JavaScript console in Opera

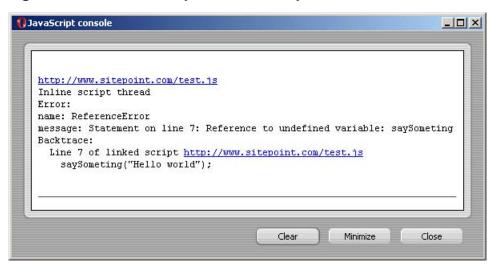


Figure 1.3. The JavaScript console in Windows IE



As you probably gathered, I'm not overly impressed by Internet Explorer's error reporting, but it is vastly better than nothing: at least you know that an error has occurred.

Using alert

The alert function is a very useful means of analyzing errors—you can use it at any point in a script to probe objects and variables to see if they contain the data you expect. For example, if you have a function that has several conditional branches, you can add an alert within each condition to find out which is being executed:

Maybe the value for *years* is not coming back as a number, like it should. You could add to the start of your script an alert that tests the variable to see what type it is:

```
function checkAge(years)
{
  alert(typeof years);
  :
```

In theory, you can put any amount of information in an alert dialog, although a very long string of data could create such a wide dialog that some of the information would be clipped or outside the window. You can avoid this by formatting the output with escape characters, such as \n for a line break.

Using try-catch

The try-catch construct is an incredibly useful way to get a script just to "try something," leaving you to handle any errors that may result. The basic construct looks like this:

If you're not sure where an error's coming from, you can wrap a try-catch around a very large block of code to trap the general failure, then tighten it around progressively smaller chunks of code within that block. For example, you could wrap a try brace around the first half of a function (at a convenient point in the code), then around the second half, to see where the error occurs; you could then divide the suspect half again, at a convenient point, and keep going until you've isolated the problematic line.

catch has a single argument (I've called it err in this case), which receives the error object; we can query properties of that object, such as name and message, to get details about the error.

Often, I use a for-in iterator to run through the entire object and find out what it says:

```
File: debugging-trycatch.js (excerpt)
for (var i in err)
{
    alert(i + ': ' + err[i]);
}
```

Writing to the Page or Window

If you're examining a great deal of data while debugging, or you're dealing with data that's formatted in a complicated way, it's often better to write that data directly to a page or popup window than to try to deal with lots of alert dialogs. If you're examining data in a loop, in particular, you could end up generating hundreds of dialogs, each of which you'll have to dismiss manually—a very tedious process.

In these kinds of situations, we can use an element's innerHTML property to write the data to the page. Here's an example in which we build a list using the contents of an array (data), then write it into a test div:

```
File: debugging-writing.js (excerpt)
var test = document.getElementById('testdiv');

test.innerHTML += '';
for (var i = 0; i < data.length; i++)
{
   test.innerHTML += '<li>' + i + '=' + data[i] + '';
}
test.innerHTML += '';
```

We can also write the data into a popup, which is useful if there's no convenient place to put it on the page:

```
File: debugging-writing.js (excerpt)
var win = window.open('', win, 'width=320,height=240');
win.document.open();
win.document.write('');
for (var i = 0; i < data.length; i++)
{
   win.document.write('<li>' + i + '=' + data[i] + '')
}
win.document.write('');
win.document.close();
```

You can format the output however you like, and use it to structure data in any way that makes it easier for you to find the error.

When you're working with smaller amounts of data, you can gain a similar advantage by writing the data to the main title element:

```
File: debugging-writing.js (excerpt)
document.title = '0 = ' + data[0];
```

This final approach is most useful when tracking data that changes continually or rapidly, such as a value being processed by a setInterval function (an asynchronous timer we'll meet properly in Chapter 14).

Using an External Debugger

I can recommend two debuggers:

- □ **Venkman**⁹ for Mozilla and Firefox
- ☐ Microsoft Script Debugger¹⁰ for Windows Internet Explorer

External debuggers are a far more detailed way to analyze your scripts, and have much greater capabilities than their in-browser counterparts. External debuggers can do things like stopping the execution of the script at specific points, or watching particular properties so that you're informed of any change to them, however it may be caused. They also include features that allow you "step through" code line by line, in order help find errors that may occur only briefly, or are otherwise difficult to isolate.

External debuggers are complex pieces of software, and it can take time for developers to learn how to use them properly. They can be very useful for highlighting logical errors, and valuable as learning tools in their own right, but they're limited in their ability to help with browser incompatibilities: they're only useful there if the bug you're looking for is in the browser that the debugger supports!

Strict Warnings

If you open the JavaScript console in Firefox you'll see that it includes options to show Errors and Warnings. Warnings notify you of code that, though it is not erroneous per se, does rely on automatic error handling, uses deprecated syntax, or is in some other way untrue to the ECMAScript specification. 11

For example, the variable fruit is defined twice in the code below:

⁹ http://www.mozilla.org/projects/venkman/
10 http://msdn.microsoft.com/scripting/

¹¹To see these warnings, it may be necessary to enable strict reporting by typing in the address about:config and setting javascript.options.strict to true.

```
File: strict-warnings.js (excerpt)
var fruit = 'mango';
if (basket.indexOf('apple') != -1)
{
   var fruit = 'apple';
}
```

We should have omitted the second var, because var is used to declare a variable for the *first time*, which we've already done. Figure 1.4 shows how the JavaScript console will highlight our error as a warning.

Figure 1.4. The JavaScript warnings console in Firefox



There are several coding missteps that can cause warnings like this. For example:

re-declaring a variable

This produces the warning, "redeclaration of var *name*," as we just saw.

failing to declare a variable in the first place

This oversight produces the warning, "assignment to undeclared variable name."

This might arise, for example, if the first line of our code read simply fruit = 'mango';

assuming the existence of an object

This assumption produces the warning "reference to undefined property name."

For example, a test condition like if (document.getElementById) assumes the existence of the getElementById method, and banks on the fact that JavaScript's automatic error-handling capabilities will convert a nonexistent method to false in browsers in which this method doesn't exist. To achieve the same end without seeing a warning, we would be more specific, using if(typeof document.getElementById != 'undefined').

There are also some function-related warnings, and a range of other miscellaneous warnings that includes my personal favorite, "useless expression," which is produced by a statement within a function that does nothing:

```
File: strict-warnings.js (excerpt)

function getBasket()
{
  var fruit = 'pomegranate';
  fruit;
}
```

For a thorough rundown on the topic, I recommend Alex Vincent's article *Tackling JavaScript strict warnings*. ¹²

Warnings don't *matter* in the sense that they don't prevent our scripts from working, but working to avoid warnings helps us to adopt better coding practice, which ultimately creates efficiency benefits. For instance, scripts run faster in Mozilla if there are no strict warnings, a subject we'll look at again in Chapter 20.



Type Conversion Testing

Although we shouldn't rely on type conversion to test a value that might be undefined, it's perfectly fine to do so for a value that might be null, because the ECMAScript specification requires that null evaluates to false. So, for example, having already established the existence of getElementById using the typeof operator as shown above, it's perfectly safe from then on to test for individual elements as shown below, because getElementById returns null for nonexistent elements in the DOM:

```
if (document.getElementById('something'))
{
```

¹² http://javascriptkit.com/javatutors/serror.shtml

```
: the element exists
}
```

Summary

In this chapter, we've talked about best-practice approaches to scripting that will make our code easier to read and manage, and will allow it to degrade gracefully in unsupported devices. We've also begun to introduce some of the techniques we'll need to build useful scripts, including the ubiquitous load event listener that we'll use for almost every solution in this book!

We've covered some pretty advanced stuff already, so don't worry if some of it was difficult to take in. We'll be coming back to all the concepts and techniques we've introduced here as we progress through the remaining chapters.

5

Navigating the Document Object Model

Browsers give JavaScript programs access to the elements on a web page via the Document Object Model (DOM)—an internal representation of the headings, paragraphs, lists, styles, IDs, classes, and all the other data to be found in the HTML on your page.

The DOM can be thought of as a tree consisting of interconnected **nodes**. Each tag in an HTML document is represented by a node; any tags that are nested inside that tag are nodes that are connected to it as children, or branches in the tree. Each of these nodes is called an **element node**. There are several other types of nodes; the most useful are the **document node**, **text node**, and **attribute node**. The document node represents the document itself, and is the root of the DOM tree. Text nodes represent the text contained between an element's tags. Attribute nodes represent the attributes specified inside an element's opening tag. Consider this basic HTML page structure:

¹Strictly speaking, each element node represents a *pair* of tags—the start and end tags of an element (e.g., $\langle p \rangle$ and $\langle p \rangle$)—or a single self-closing tag (e.g., $\langle br \rangle$, or $\langle br \rangle$ in XHTML).

```
 For centuries, the stars have been
    more to humankind than just burning balls of gas ...
  </body>
</html>
```

The DOM for this page could be visualized as Figure 5.1.

Every page has a document node, but its descendents are derived from the content of the document itself. Through the use of element nodes, text nodes, and attribute nodes, every piece of information on a page is accessible via JavaScript.

The DOM isn't just restricted to HTML and JavaScript, though. Here's how the W3C DOM specification site² explains the matter:

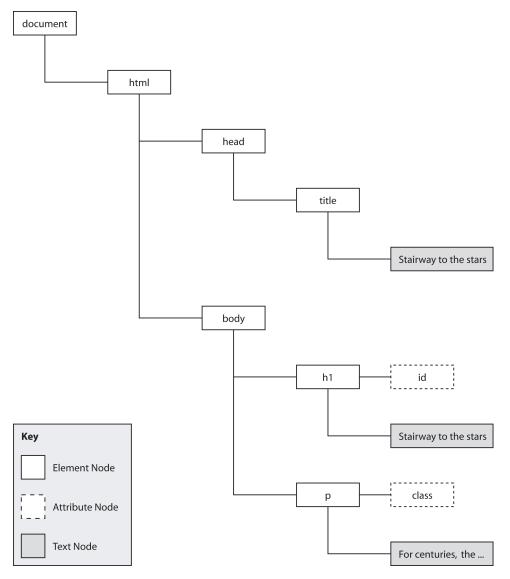
The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.

So, even though the mixture of JavaScript and HTML is the most common combination of technologies in which the DOM is utilized, the knowledge you gain from this chapter can be applied to a number of different programming languages and document types.

In order to make you a "master of your DOMain," this chapter will explain how to find any element you're looking for on a web page, then change it, rearrange it, or erase it completely.

² http://www.w3.org/DOM/

Figure 5.1. The DOM structure of a simple HTML page, visualized as a tree hierarchy



Accessing Elements

Access provides control, control is power, and you're a power programmer, right? So you need access to everything that's on a web page. Fortunately, JavaScript gives you access to any element on a page using just a few methods and properties.

Solution

Although it's possible to navigate an HTML document like a road map—starting from home and working your way towards your destination one node at a time—this is usually an inefficient way of finding an element because it requires a lot of code, and any changes in the structure of the document will usually mean that you have to rewrite your scripts. If you want to find something quickly and easily, the method that you should tattoo onto the back of your hand is document.getElementById.

Assuming that you have the correct markup in place, getElementById will allow you immediately to access any element by its unique id attribute value. For instance, imagine your web page contains this code:

```
File: access_element.html (excerpt)
 <a id="sirius" href="sirius.html">Journey to the stars</a>
```

You can use the a element's id attribute to get direct access to the element itself:

```
File: access_element.js (excerpt)
var elementRef = document.getElementById("sirius");
```

The value of the variable elementRef will now be referenced to the a element—any operations that you perform on elementRef will affect that exact hyperlink.

getElementById is good for working with a specific element; however, sometimes you'll want to work with a *group* of elements. In order to retrieve a group of elements on the basis of their tag names, you can use the method getElementsByTagName.

As can be seen from its name, getElementsByTagName takes a tag name and returns all elements of that type. Assume that we have this HTML code:

We can retrieve a collection that contains each of the hyperlinks like so:

```
File: access_element2.js (excerpt)
var anchors = document.getElementsByTagName("a");
```

The value of the variable anchors will now be a **collection** of a elements. Collections are similar to arrays in that each of the items in a collection is referenced using square bracket notation, and the items are indexed numerically starting at zero. The collection returned by <code>getElementsByTagName</code> sorts the elements by their source order, so we can reference each of the links thus:

```
anchorArray[0] the a element for "Sirius"
anchorArray[1] the a element for "Canopus"
anchorArray[2] the a element for "Arcturus"
anchorArray[3] the a element for "Vega"
```

Using this collection you can iterate through the elements and perform an operation on them, such as assigning a class using the element nodes' className property:

```
File: access_element2.js (excerpt)

var anchors = document.getElementsByTagName("a");

for (var i = 0; i < anchors.length; i++)
{
```

```
anchors[i].className = "starLink";
}
```

Unlike getElementById, which may be called on the document node only, the getElementsByTagName method is available from every single element node. You can limit the scope of the getElementsByTagName method by executing it on a particular element. getElementsByTagName will only return elements that are descendents of the element on which the method was called.

If we have two lists, but want to assign a new class to the links in one list only, we can target those a elements exclusively by calling getElementsByTagName on their parent list:

```
File: access_element3.html (excerpt)
ul id="planets">
 <1i>>
   <a href="mercury.html">Mercury</a>
 <
   <a href="venus.html">Venus</a>
 <1i>>
   <a href="earth.html">Earth</a>
 <
   <a href="mars.html">Mars</a>
 <
   <a href="sirius.html">Sirius</a>
 <
   <a href="canopus.html">Canopus</a>
 <a href="arcturus.html">Arcturus</a>
 <a href="vega.html">Vega</a>
```

To target the list of stars, we need to obtain a reference to the parent ul element, then call getElementsByTagName on it directly:

```
File: access_element3.js (excerpt)
```

```
var starsList = document.getElementById("stars");
var starsAnchors = starsList.getElementsByTagName("a");
```

The value of the variable starsAnchors will be a collection of the a elements inside the stars unordered list, instead of a collection of *all* a elements on the page.



DOM 0 Collections

Many "special" elements in an HTML document can be accessed by even more direct means. The body element of the document can be accessed as document.body. A collection of all the forms in a document may be found in document.forms. All of the images in a document may be found in document.images.

In fact, most of these collections have been around since before the DOM was standardized by the W3C, and are commonly referred to as **DOM 0** properties.

Because the initial implementations of these features were not standardized, these collections have occasionally proven unreliable in browsers that are moving towards standards compliance. Early versions of some Mozilla browsers (e.g., Firefox), for example, did not support these collections on XHTML documents.

Today's browsers generally do a good job of supporting these collections; however, if you *do* run into problems, it's worth trying the more verbose <code>getElementsByTagName</code> method of accessing the relevant elements. Instead of <code>document.body</code>, for example, you could use:

```
var body = document.getElementsByTagName("body")[0];
```

Discussion

If you really need to step through the DOM hierarchy element by element, each node has several properties that enable you to access related nodes:

node.childNodes a collection that contains source-order references to

each of the children of the specified node, including

both elements and text nodes

node.firstChild the first child node of the specified node

node.lastchild the last child node of the specific node

node.parentNode a reference to the parent element of the specified

node

node.nextSibling the next node in the document that has the same

parent as the specified node

node.previousSibling the previous element that's on the same level as the

specified node

If any of these properties do not exist for a specific node (e.g., the last node of a parent will not have a next sibling), they will have a value of null.

Take a look at this simple page:

The list item with ID star2 could be referenced using any of these expressions:

```
document.getElementById("star1").nextSibling;
document.getElementById("star3").previousSibling;
document.getElementById("starList").childNodes[1];
document.getElementById("star1").parentNode.childNodes[1];
```



Whitespace Nodes

Some browsers will create **whitespace nodes** between the element nodes in any DOM structure that was interpreted from a text string (e.g., an HTML file). Whitespace nodes are text nodes that contain only whitespace (tabs, spaces, new lines) to help format the code in the way it was written in the source file.

When you're traversing the DOM node by node using the above properties, you should always allow for these whitespace nodes. Usually, this means checking that the node you've retrieved is an element node, not just a whitespace node that's separating elements.

There are two easy ways to check whether a node is an element node or a text node. The nodeName property of a text node will always be "#text", whereas the nodeName of an element node will identify the element type. However, in distinguishing text nodes from element nodes, it's easier to check the nodeType property. Element nodes have a nodeType of 1, whereas text nodes have a nodeType of 3. You can use this knowledge as a test when retrieving elements:

```
File: access_element4.js (excerpt)

var star2 = document.getElementById("star1").nextSibling;

while (star2.nodeType == "3")
{
   star2 = star2.nextSibling;
}
```

Using these DOM properties, it's possible to start your journey at the root html element, and end up buried in the legend of some deeply-nested fieldset—it's all just a matter of following the nodes.

Creating Elements and Text Nodes

JavaScript doesn't just have the ability to modify existing elements in the DOM; it can also create new elements and place them anywhere within a page's structure.

Solution

createElement is the aptly named method that allows you to create new elements. It only takes one argument—the type (as a string) of the element you wish to create—and returns a reference to the newly-created element:

```
File: create_elements.js (excerpt) var newAnchor = document.createElement("a");
```

The variable newAnchor will be a new a element, ready to be inserted into the page.



Specifying Namespaces in Documents with an XML MIME Type

If you're coding JavaScript for use in documents with a MIME type of application/xhtml+xml (or some other XML MIME type), you should use the method createElementNS, instead of createElement, to specify the namespace for which you're creating the element:

```
var newAnchor = document.createElementNS(
    "http://www.w3.org/1999/xhtml", "a");
```

This distinction applies to a number of DOM methods, such as removeElement/removeElementNS and getAttribute/getAttributeNS; however, we won't use the namespace-enhanced versions of these methods in this book.

Simon Willison provides a brief explanation of working with JavaScript and different MIME types³ on his web site.

The text that goes inside an element is actually a child text node of the element, so it must be created separately. Text nodes are different from element nodes, so they have their own creation method, createTextNode:

```
File: create_elements.js (excerpt)
var anchorText = document.createTextNode("monoceros");
```

If you're modifying an existing text node, you can access the text it contains via the nodeValue property. This allows you to get and set the text inside a text node:

```
var textNode = document.createTextNode("monoceros");
var oldText = textNode.nodeValue;
textNode.nodeValue = "pyxis";
```

The value of the variable oldText is now "monoceros", and the text inside textNode is now "pyxis".

You can insert either an element node or a text node as the last child of an existing element using its appendChild method. This method will place the new node after all of the element's existing children.

Consider this fragment of HTML:

 $^{^3\} http://simon.incutio.com/archive/2003/06/15/javascriptWithXML$

We can use DOM methods to create and insert another link at the end of the paragraph:

```
File: create_elements.js (excerpt)
var anchorText = document.createTextNode("monoceros");

var newAnchor = document.createElement("a");
newAnchor.appendChild(anchorText);

var parent = document.getElementById("starLinks");
var newChild = parent.appendChild(newAnchor);
```

The value of the variable newChild will be a reference to the newly inserted element.

If we were to translate the state of the DOM after this code had executed into HTML code, it would look like this:

```
     <a href="sirius.htm">Sirius</a><a>monoceros</a>
```

We didn't specify any attributes for the new element, so it doesn't link anywhere at the moment. The process for specifying attributes is explained shortly in "Reading and Writing the Attributes of an Element".

Discussion

There are three basic ways by which a new element or text node can be inserted into a web page. The approach you use will depend upon the point at which you want the new node to be inserted: as the last child of an element, before another node, or as the replacement for a node. The process of appending an element as the last child was explained above. You can insert the node before an existing node using the insertBefore method of its parent element, and you can replace a node using the replaceChild method of its parent element.

In order to use insertBefore, you need to have references to the node you're going to insert, and to the node before which you wish to insert it. Consider this HTML code:

We can insert a new link before the existing one by calling insertBefore from its parent element (the paragraph):

```
File: create_elements2.js (excerpt)
var anchorText = document.createTextNode("monoceros");

var newAnchor = document.createElement("a");
newAnchor.appendChild(anchorText);

var existingAnchor = document.getElementById("sirius");
var parent = existingAnchor.parentNode;
var newChild = parent.insertBefore(newAnchor, existingAnchor);
```

The value of the variable newChild will be a reference to the newly inserted element.

If we were to translate into HTML the state of the DOM after this operation, it would look like this:

```
    <a>monoceros</a><a id="sirius" href="sirius.htm">Sirius</a>
```

Instead, we could replace the existing link entirely using replaceChild:

```
File: create_elements3.js (excerpt)
var anchorText = document.createTextNode("monoceros");
var newAnchor = document.createElement("a");
newAnchor.appendChild(anchorText);

var existingAnchor = document.getElementById("sirius");
var parent = existingAnchor.parentNode;
var newChild = parent.replaceChild(newAnchor, existingAnchor);
```

The DOM would then look like this:

```
    <a>monoceros</a>
```

Changing the Type of an Element

Are your ordered lists feeling a bit unordered? Do your headings have paragraph envy? Using a little JavaScript knowledge, it's possible to change the type of an element entirely, while preserving the structure of its children.

Solution

There's no straightforward, simple way to change the type of an element. In order to achieve this feat you'll have to perform a bit of a juggling act.

Let's assume that we want to change this paragraph into a div:

We need to create a new div, move each of the paragraph's children into it, then swap the new element for the old:

```
File: change_type_of_element.js(excerpt)
var div = document.createElement("div");
var paragraph = document.getElementById("starLinks");

for (var i = 0; i < paragraph.childNodes.length; i++)
{
  var clone = paragraph.childNodes[i].cloneNode(true);
  div.appendChild(clone);
}
paragraph.parentNode.replaceChild(div, paragraph);</pre>
```

The only unfamiliar line here should be the point at which a **clone** is created for each of the paragraph's children. The **cloneNode** method produces an identical copy of the node from which it's called. By passing this method the argument

true, we indicate that we want all of that element's children to be copied along with the element itself. Using cloneNode, we can mirror the original element's children under the new div, then remove the paragraph once we're finished copying.

While cloning nodes is useful in some circumstances, it turns out that there's a cleaner way to approach this specific problem. We can simply move the child nodes of the existing paragraph into the new div. DOM nodes can belong only to one parent element at a time, so adding the nodes to the div also removes them from the paragraph:

```
File: change_type_of_element2.js (excerpt)
var div = document.createElement("div");
var paragraph = document.getElementById("starLinks");
while (paragraphNode.childNodes.length > 0){
    div.appendChild(paragraphNode.firstChild);
}
paragraph.parentNode.replaceChild(div, paragraph);
```

note

Take Care Changing the Node Structure of the DOM

The elements in a collection are updated automatically whenever a change occurs in the DOM—even if you copy that collection into a variable before the change occurs. So, if you remove from the DOM an element that was contained in a collection with which you had been working, the element reference will also be removed from the collection. This will change the length of the collection as well as the indexes of any elements that appear after the removed element.

When performing operations that affect the node structure of the DOM—such as moving a node to a new parent element—you have to be careful about iterative processes. The code above uses a <code>While</code> loop that only accesses the first child of the paragraph, because each time a child is relocated, the length of the <code>childNodes</code> collection will decrease by one, and all the elements in the collection will shift along. A <code>for</code> loop with a counter variable would not handle all the children correctly because it would assume that the contents of the collection would remain the same throughout the loop.

Discussion

There's no easy way to copy the attributes of an element to its replacement.⁴ If you want the new element to have the same id, class, href, and so on, you'll have to copy the values over manually:

```
File: change_type_of_element.js (excerpt)
div.id = paragraph.getAttribute("id");
div.className = paragraph.className;
```

Removing an Element or Text Node

Once an element has outlived its usefulness, it's time to give it the chop. You can use JavaScript to remove any element cleanly from the DOM.

Solution

The removeChild method removes any child node from its parent, and returns a reference to the removed object.

Let's start off with this HTML:

```
File: remove_element.html (excerpt)
 <a id="sirius" href="sirius.html">Sirius</a>
```

We could use removeChild to remove the hyperlink from its parent paragraph like so:

```
File: remove_element.js (excerpt)

var anchor = document.getElementById("sirius");

var parent = anchor.parentNode;

var removedChild = parent.removeChild(anchor);
```

The variable removedChild will be a reference to the a element, but that element will not be located anywhere in the DOM: it will simply be available in memory, much as if we had just created it using createElement. This allows us to relocate it to another position on the page, it we wish, or we can simply let the variable

⁴If you look at the DOM specification, it *looks* like there is. Unfortunately, Internet Explorer's support for the relevant properties and methods is just not up to the task.

disappear at the end of the script, and the reference will be lost altogether—effectively deleting it. Following the above code, the DOM will end up like this:

Of course, you don't need to assign the return value from removeChild to a variable. You can just execute it and forget about the element altogether:

```
var anchor = document.getElementById("sirius");
var parent = anchor.parentNode;
parent.removeChild(anchor);
```

Discussion

If the element that you're deleting has children that you wish to preserve (i.e., you just want to "unwrap" them by removing their parent), you must rescue those children to make sure they stay in the document when their parent is removed. You can achieve this using the already-mentioned insertBefore method, which, when used on elements that are already contained in the DOM, first removes them, then inserts them at the appropriate point.

The paragraph in the following HTML contains multiple children:

We can loop through the paragraph's childNodes collection, and relocate each of its children individually before removing the element itself:

```
File: remove_element2.js (excerpt)
var parent = document.getElementById("starLinks");
var container = document.getElementById("starContainer");
while (parent.childNodes.length > 0)
{
   container.insertBefore(parent.childNodes[0], parent);
}
```

```
container.removeChild(parent);
```

The page's DOM will now look like this:

Reading and Writing the Attributes of an Element

The most frequently used parts of an HTML element are its attributes—its id, class, href, title, or any of a hundred other pieces of information that can be included in an HTML tag. JavaScript is able not only to read these values, but write them as well.

Solution

Two methods exist for reading and writing an element's attributes. getAttribute allows you to read the value of an attribute, while setAttribute allows you to write it.

Consider this HTML:

```
File: read_write_attributes.html (excerpt)
<a id="antares" href="antares.html" title="A far away place">
Antares</a>
```

We would be able to read the attributes of the element like so:

```
File: read_write_attributes.js (excerpt)
var anchor = document.getElementById("antares");
var anchorId = anchor.getAttribute("id");
var anchorTitle = anchor.getAttribute("title");
```

The value of the variable anchorId will be "antares", and the value of the variable anchorTitle will be "A far away place".

To change the attributes of the hyperlink, we use **setAttribute**, passing it the name of the attribute to be changed, and the value we want to change it to:

```
File: read_write_attributes2.js (excerpt)
var anchor = document.getElementById("antares");
anchor.setAttribute("title", "Not that far away");
var newTitle = anchor.getAttribute("title");
```

The value of the variable newTitle will now be "Not that far away".

Discussion

In its journey from the free-roaming Netscape wilderness to the more tightly defined, standards-based terrain of the modern age, the DOM standard has picked up a fair amount of extra syntax for dealing with HTML. One of the most pervasive of these extras is the mapping between DOM properties and HTML attributes.

When a document is parsed into its DOM form, special attribute nodes are created for an element's attributes. These nodes are not accessible as "children" of that element: they are accessible only via the two methods mentioned above. However, as a throwback to the original DOM implementations (called DOM 0, where the zero suggests these features came prior to standards), current DOM specs contain additional functionality that's specific to HTML. In particular, attributes are accessible directly as properties of an element. So, the href attribute of a hyperlink is accessible through <code>link.getAttribute("href")</code> as well as through <code>link.href</code>.

This shortcut syntax is not only cleaner and more readable: in some situations it is also *necessary*. Internet Explorer 6 and versions below will not propagate changes made via setAttribute to the visual display of an element. So any changes that are made to the class, id, or style of an element using setAttribute will not affect the way it's displayed. In order for those changes to take effect, they must be made via the element node's attribute-specific properties.

To further confuse matters, the values that are returned when an attribute-specific property is read vary between browsers, the most notable variations occurring in Konqueror. If an attribute doesn't exist, Konqueror will return null as the value of an attribute-specific property, while all other browsers will return an empty string. In a more specific case, some browsers will return <code>link.getAttrib-ute("href")</code> as an absolute URL (e.g., "http://www.example.com/ant-

ares.html"), while others return the actual attribute value (e.g., "antares.html"). In this case, it's safer to use the dot property, as it consistently returns the absolute URL across browsers.

So, what's the general solution to these problems?

The basic rule is this: if you are certain that an attribute has been assigned a value, it's safe to use the dot property method to access it. If you're unsure whether or not an attribute has been set, you should first use one of the DOM methods to ensure that it has a value, then use the dot property to obtain its value.

For *reading* an unverified attribute, use the following:

```
var anchor = document.getElementById("sirius");
if (anchor.getAttribute("title") &&
        anchor.title == "Not the satellite radio")
{
        :
}
```

This makes sure that the attribute exists, and is not null, before fetching its value.

For writing to an unverified attribute, use the following code:

```
var anchor = document.getElementById("sirius");
anchor.setAttribute("title", "");
anchor.title = "Yes, the satellite radio";
```

This code makes sure that the attribute is created correctly first, and is then set in such a way that Internet Explorer will not have problems if the attribute affects the visual display of the element.

This rule has a few exceptions for attributes whose existence you can guarantee. The most notable of these "must-have" attributes are style and class, which will always be valid for any given element; thus, you can immediately reference them as dot properties (element.style and element.className respectively).

class is one of two attributes that get a little tricky, because class is a reserved word in JavaScript. As a property, it is written element.className, but using

getAttribute/setAttribute, we write element.getAttribute("class"), except
in Internet Explorer, where we still use element.getAttribute("className").

The other attribute that we have to watch out for is the for attribute of a label. It follows the same rules as class, but its property form is htmlFor. Using getAttribute/setAttribute, we write element.getAttribute("for"), but in Internet Explorer it's element.getAttribute("htmlFor").

Getting all Elements with a Particular Attribute Value

The ability to find all the elements that have a particular attribute can be pretty handy when you need to modify all elements that have the same class or title, for example.

Solution

In order to find elements with a particular attribute value, we need to check every element on the page for that attribute. This is a very calculation-intensive operation, so it shouldn't be undertaken lightly. If you wanted to find all input elements with type="checkbox", you're better off limiting your search to input elements first:

```
var inputs = document.getElementsByTagName("input");

for (var i = 0; i < inputs.length; i++)
{
   if (inputs.getAttribute("type") == "checkbox")
   {
        :
     }
}</pre>
```

This will require less calculation than iterating through *every* element on the page and checking its type. However, the function presented in this solution—getElementsByAttribute—is ideal when you need to find a number of elements of *different* types that have the same attribute value.

The easiest way to check every element on a page is to loop through the collection returned by getElementsByTagName("*"). The only problem with this method is that Internet Explorer 5.0 and 5.5 do not support the asterisk wildcard for tag

selection. Luckily, these browsers support the document.all property, which is an array containing all the elements on the page. getElementsByAttribute handles this issue with a simple code branch, then proceeds to check the elements for a given attribute value, adding matches to an array to be returned:

```
File: get_elements_by_attribute.js (excerpt)
function getElementsByAttribute(attribute, attributeValue)
  var elementArray = new Array();
  var matchedArray = new Array();
  if (document.all)
    elementArray = document.all;
  else
    elementArray = document.getElementsByTagName("*");
  for (var i = 0; i < elementArray.length; i++)</pre>
    if (attribute == "class")
      var pattern = new RegExp("(^| )" +
          attributeValue + "( |$)");
      if (pattern.test(elementArray[i].className))
        matchedArray[matchedArray.length] = elementArray[i];
    else if (attribute == "for")
      if (elementArray[i].getAttribute("htmlFor") ||
          elementArray[i].getAttribute("for"))
        if (elementArray[i].htmlFor == attributeValue)
          matchedArray[matchedArray.length] = elementArray[i];
        }
      }
    else if (elementArray[i].getAttribute(attribute) ==
        attributeValue)
```

```
matchedArray[matchedArray.length] = elementArray[i];
}

return matchedArray;
}
```

A lot of the code in <code>getElementsByAttribute</code> deals with the browser differences in attribute handling that were mentioned earlier in this chapter, in "Reading and Writing the Attributes of an Element". The necessary techniques are used if the required attribute is <code>class</code> or <code>for</code>. As an added bonus when checking for a match on the <code>class</code> attribute, if an element has been assigned multiple classes, the function automatically checks each of these to see whether it matches the required value.

Adding and Removing Multiple Classes to/from an Element

Combining multiple classes is a very useful CSS technique. It provides a very primitive means of inheritance by allowing a number of different styles to be combined on the one element, allowing you to mix and match different effects throughout a site. They're particularly useful in situations like highlighting elements: a class can be added that highlights an element without disturbing any of the other visual properties that may have been applied to the element by other classes. However, if you are assigning classes in JavaScript you have to be careful that you don't inadvertently overwrite previously assigned classes.

Solution

The class for any element is accessible via its className property. This property allows you both to read and write the classes that are currently applied to that element. Because it's just one string, the most difficult part of working with className is that you need to deal with the syntax it uses to represent multiple classes.

The class names in an element's className property are separated by spaces. The first class name is not preceded by anything, and the last class name is not followed by anything. This makes it easy to add a class to the class list naively: just concatenate a space and the new class name to the end of className. However, you'll want to avoid adding a class name that already exists in the list, as

this will make removing the class harder. You'll also want to avoid using a space at the beginning of the className value, because this will cause errors in Opera 7:

```
file: add_remove_classes.js (excerpt)

function addClass(target, classValue)
{
   var pattern = new RegExp("(^| )" + classValue + "( |$)");

   if (!pattern.test(target.className))
   {
      if (target.className == "")
      {
        target.className = classValue;
      }
      else
      {
        target.className += " " + classValue;
      }
   }

   return true;
}
```

First, addClass creates a regular expression pattern containing the class to be added. It then uses this pattern to test the current className value. If the class name doesn't already exist, we check for an empty className value (in which case the class name is assigned to the property verbatim), or we append to the existing value a space and the new class name.



Separating Classes

Some regular expression examples for finding classes use the word boundary special character (\b) to separate classes. However, this will not work with all valid class names, such as those containing hyphens.

The process for removing a class uses a regular expression pattern that's identical to the one we use to add a class, but we don't need to perform as many checks:

```
File: add_remove_classes.js(excerpt)

function removeClass(target, classValue)

{
   var removedClass = target.className;
   var pattern = new RegExp("(^| )" + classValue + "( |$)");
```

```
removedClass = removedClass.replace(pattern, "$1");
removedClass = removedClass.replace(/ $/, "");

target.className = removedClass;
return true;
}
```

After removeClass has executed the replacement regular expression on a copy of the className property's value, it cleans up the resulting value by removing any trailing space (which is created when we remove the last class in a multiple class className), then assigns it back to the target's className.

Summary

This chapter introduced the basic but powerful tools that you'll need in order to manipulate the Document Object Model. It's important that you understand the DOM—the skeleton beneath everything you see in a browser—as you manipulate any web page. Knowing how to create, edit, and delete parts of the DOM is crucial to understanding the remainder of this book. Once you've mastered these techniques, you'll be well on your way to becoming a proficient JavaScript programmer.

7

Working with Windows and Frames

This chapter is about simple window and frame manipulation, including tasks like opening popups, communicating between frames, ¹ and finding out the page's scrolling position.

Plenty of people feel that window manipulation is akin to the Dark Side. They believe that a window is part of the user's GUI, not the document, and since JavaScript is a document scripting language, it has no business manipulating windows.

I'm generally inclined to agree, yet I know that opinion is sometimes a luxury. If your clients ask for something specific, you can't necessarily change their minds, or have the freedom to turn down work on the basis of such a principle. In this chapter, we'll cover a range of practical window and frame manipulation tasks while remaining sensitive to the usability and accessibility issues that can arise from their use.

Note, though, that there are limits, and some varieties of window scripting are particularly unfriendly. We won't be dealing with aggressive tactics like closing or modifying the user's *primary* window, moving windows around the screen, or opening full-screen or "chromeless" windows. These are exactly the kinds of abuses that have given JavaScript a bad name.

¹The techniques involved in reading data from an **iframe** will be covered in Chapter 18.

Through most of this chapter we'll be looking closely at the properties and methods of the window object. These are implemented by different browsers in a variety of ways, most of which have been in use since the days before JavaScript was standardized.

We'll have quite a few code branches to deal with, but we'll avoid the dreaded browser sniffing by careful use of **object detection**, the process of detecting an object or feature to test for compatibility, rather than detecting specific browsers.

Using Popup Windows

Should you use popup windows? The most considered answer I have is this: *not if you can help it*. Popup windows have gained a bad reputation from marketers' aggressive use of them, but even *requested* popups can be barriers to good usability.

I won't say that popups are *never* appropriate, but I will say that they're *seldom* so. Nevertheless, there are situations where popping open a new window is arguably the most appropriate solution: an online survey might be one example, as the format may make the content more approachable; DHTML games are another, as the viewport may need to be of a known size.

I'll qualify my opinion by discussing the problems that popups create, then providing a pragmatic method for using them that mitigates these problems as much as possible.

What's Wrong with Popups?

The main problem with most popup window scripts is that they don't consider the needs of the user—they address only the needs of the designer. The results? We've all seen them:

popups that are generated from links, though those links do nothing when scripting is not available
popup windows that don't have a status bar, so you can't necessarily tell whether the document has loaded or stalled, is still loading, etc.
popups that don't give users the ability to resize the window, and popups that fail to generate scrollbars for content that might scale outside the window
windows that are "chromeless," or open to the full size of the user's screen

These issues are not just questions of usability, but of accessibility as well. For example, screen-reader users may not be notified by their devices that a new window has opened. This could obviously cause confusion if they then attempted to go back in the browser history (they can't). The same thing might happen for a sighted user if a window opens at full-size: you and I may be familiar with using the taskbar to monitor open windows, but not all computer users are—they may not even realize that a new window has popped up.

If you're going to use popups, looking out for issues like these, and being generally sensitive to their impacts, will make your popups friendlier to users, and less of a strain on your conscience.

Also, bear in mind that, from a *developer's* perspective, popup windows are not guaranteed to work: most browsers now include options to suppress popup windows, and in some cases, suppression occurs even if the popup is generated in response to a user event.

You may be able to allow for this as you would for situations in which scripting was not supported: by ensuring that the underlying trigger for the popup still does something useful if the popup fails. Or you might have your code open a window and then check its own closed property, to see if it's actually displayed (we'll look at this technique in the next solution).

But neither of these approaches is guaranteed to work with every browser and popup blocker out there, so for this as much as the usability reasons, it's simpler and better to avoid using popups whenever you can.

How Do I Minimize the Problems?

W	hat we need to do is establish some golden rules for the ethical use of popups:		
	Make sure any triggering link degrades properly when scripting is not available		
	Always include the status bar.		
	Always include a mechanism to overflow the content: either allow window resizing, or allow scrollbars to appear, or both.		
	Don't open windows that are larger than 640x480 pixels.		

By limiting the size of popups, you ensure that they're smaller than users' primary windows on the vast majority of monitors. This increases the likelihood that the user will realize that the popup is a new window.

Solution

Here's a generic popup function that's based on the guidelines above:

```
File: make-popup.js (excerpt)
function makePopup(url, width, height, overflow)
  if (width > 640) { width = 640; }
  if (height > 480) { height = 480; }
  if (overflow == '' || !/^(scroll|resize|both)$/.test(overflow))
    overflow = 'both';
  }
  var win = window.open(url, '',
      'width=' + width + ',height=' + height
      + ',scrollbars=' + (/^(scroll|both)$/.test(overflow) ?
      'yes' : 'no')
      + ',resizable=' + (/^(resize|both)$/.test(overflow) ?
      'yes' : 'no')
      + ',status=yes,toolbar=no,menubar=no,location=no'
  );
  return win;
```

As well as limiting the window size, this script refuses to create a popup that doesn't have an overflow, so if you don't specify "scroll", "resize", or "both" for the overflow argument, the default setting of "both" will be used.



The Ternary Operator

This script uses a shortcut expression called a **ternary operator** to evaluate each of the overflow options. The ternary operator uses ? and : characters to divide the two possible outcomes of an evaluation, and is equivalent to a single pair of if..else conditions. Consider this code:

```
if (outlook == 'optimistic') { glass = 'half-full'; }
else { glass = 'half-empty'; }
```

That code is equivalent to the markup below:

```
glass = (outlook == 'optimistic' ? 'half-full' :
    'half-empty');
```

The parentheses are not required, but you may find they make the expression easier to read.

For more about this and other useful shortcuts, see Chapter 20.

Once you have the popup function in place, you can call it in a variety of ways. For example, you could use a regular link:

```
File: make-popup.html (excerpt) <a href="survey.html" id="survey_link">Online survey</a>
```

If scripting is not available, this will work just like any other link, but if scripting is available, the script can trigger a click event handler that passes its href to the makePopup function, along with the other settings. The return value of the handler depends on whether or not the window is actually opened; browsers that block the popup will follow the link as normal:

```
File: make-popup.js (excerpt)

document.getElementById('survey_link').onclick = function()

{
   var survey = makePopup(this.href, 640, 480, 'scroll');
   return survey.closed;
};
```

In general, if you have a script that requires that a window be generated, you can call the makePopup function directly with a URL:

```
var cpanel = makePopup('cpanel.html', 480, 240, 'resize');
```

If you need to close that window later in your script, you can do so by using the close method on the stored window reference:

```
cpanel.close();
```

Discussion

The window.open method can take a number of arguments—in addition to the URL and window name—which specify whether the window should have particular decorations, such as the menu bar, tool bar, or address (location) bar. These

arguments are passed as a comma-delimited string to the *third* argument of window.open:

```
var win = window.open('page.html', 'winName',
    'width=640,height=480,'
    + 'scrollbars=yes,resizable=yes,status=yes,'
    + 'toolbar=no,menubar=no,location=no');
```

In our makePopup function, the menubar, toolbar, and location arguments are all preset to no because these elements are rarely useful for popup windows—they're navigational tools, after all. Popups are mostly used for one-page interfaces, or those in which history navigation is discouraged, such as our survey example, or the logon procedure for a bank's web site.

You can change those arguments if you need to, but the status argument should *always* be set to yes, because turning it off undermines good usability. (I know—I've mentioned it already, but I'm saying it again because it's important!)

The resizable argument may not have any effect—in some browsers, either by design or as a result of user preferences, it's not possible to create non-resizable windows, even if you set this value to no. In fact, in Opera 8 for Mac OS X, it's not possible to create custom-sized windows at all—a created window will appear as a new tab in the current window. That specific exception might not be significant in itself, but it serves to illustrate the general point that control over the properties of a created window is not absolutely guaranteed.

Once a new window is open, you can bring it into focus using the object's focus method. This isn't usually necessary—generally, it happens by default—but the technique may be useful when you're scripting with multiple windows:

```
var cpanel = makePopup('cpanel.html', 480, 240, 'resize');
cpanel.focus();
```

Alternatively, you may want to open a popup but keep the focus in the *primary* window (thereby creating a so-called "popunder"). You can take the focus away from a window using its blur method:

```
var cpanel = makePopup('cpanel.html', 480, 240, 'resize');
cpanel.blur();
```

However, in that case you can't predict where the focus will go to next, so it's more reliable to refocus the primary window:

```
var cpanel = makePopup('cpanel.html', 480, 240, 'resize');
self.focus();
```

Opening Off-site Links in a New Window

In the strict versions of HTML 4 and XHTML 1, the target attribute for links no longer exists. One interpretation of this is that web pages simply shouldn't open links in new windows; another is that targeting doesn't have universal semantics and therefore shouldn't be defined in HTML.²

There are other interpretations, and the arguments are long (and sometimes tedious), but suffice it to say that you may find yourself needing a solution to this problem. Whatever your personal views may be, it's a common request of web development clients.

Solution

This script identifies links by the rel attribute value external. The rel attribute is a way of describing the relationship between a link and its target,³ so its use for identifying links that point to another site is semantically non-dubious:

```
File: offsite-links.html (excerpt)
<a href="http://www.google.com/" rel="external">Google
    (offsite)</a>
```

If each external link is identified like that, a single document.onclick event handler can process clicks on all such links:

```
File: offsite-links.js

document.onclick = function(e)
{
   var target = e ? e.target : window.event.srcElement;

   while (target && !/^(a|body)$/i.test(target.nodeName))
   {
      target = target.parentNode;
```

³ http://www.w3.org/TR/REC-html40/struct/links.html#h-12.1.2

The CSS 3 working draft includes a set of target properties for link presentation [http://www.w3.org/TR/2004/WD-css3-hyperlinks-20040224/], which could eventually see this mechanism handed to CSS instead. Personally, I hope this never gets past the draft stage, because it's nothing to do with CSS: interface control is no more appropriate in a design language than it is in a semantic markup language!

```
if (target && target.getAttribute('rel')
     && target.rel == 'external')
{
   var external = window.open(target.href);
   return external.closed;
}
```

Discussion

Using a single, document-wide event handler is the most efficient approach—it's much better than iterating through all the links and binding a handler to each one individually. We can find out which element was actually clicked by referencing the **event target** property. For more about events and event properties, see Chapter 13, but here's a brief summary of the situation.

Two completely different event models are employed by current browsers. The script establishes which one should be used by looking for e—the event argument that's used by Mozilla browsers, and has been adopted by most other browsers—as opposed to the window.event object used by Internet Explorer. It then saves the object property that's appropriate to the model in use: either target for Mozilla and like browsers, or srcElement for IE.

The target object (if it's not null) can be one of three things: a link element node, an element or text node inside a link, or some other node. We want the *first two* cases to be handled by our script, but clicks arising from the last situation may be safely ignored. What we do is follow the trail of parent nodes from the event target until we either find a link, or get to the body element.

Once we have a unified target link, we need simply to check for a rel attribute with the correct value; if it exists, we can open a window with the link's href, and if all of *that* is successful (as judged by the new window object's closed property), the handler will return false, preventing the original link from being followed.

Passing a link to window.open without defining arguments will create a window with default decorations—as will a link with target="blank".



The First Test

We use **getAttribute** as the first test for **rel** because attribute-specific properties are only reliable *if* you know for certain that the attribute in question has been assigned a value. We can't go straight to testing target.rel against a string, because it might be null or undefined. This was discussed in more detail in "Reading and Writing the Attributes of an Element" in Chapter 5.

Communicating Between Frames

If you're working in a framed environment, it may be necessary to have scripts communicate between frames, either reading or writing properties, or calling functions in different documents.

If you have a choice about whether or not to use frames, I'd strongly advise *against* doing so, because they have many serious usability and accessibility problems, quite apart from the fact that they're conceptually broken (they create within the browser states that cannot be addressed⁴). But as with your use of popups, in some cases you may not have a choice about your use of frames. So if you really must use them, here's what you'll need to do.

Solution

Let's begin with a simple frameset document:

135

⁴ http://www.456bereastreet.com/archive/200411/who_framed_the_web_frames_and_usability/

```
</noframes>
  </frameset>
</html>
We can use four references for cross-frame scripting:
window or self refers to the current framed page.
parent refers to the page that contains the frame that contains the current
   page.
□ top refers to the page at the very top of the hierarchy of frames, which will
   be the same as parent if there's only one frameset in the hierarchy.
☐ The frames collection is an associative array of all the frames in the current
   page.
Let's say we have a script in contentFrame that wants to communicate the page
in navigationFrame. Both pages are contained in a single frameset—the only
one in the hierarchy—so we could successfully make any of the following refer-
ences from within contentFrame:
parent.frames[0]
top.frames[0]
parent.frames['navigationFrame']
top.frames['navigationFrame']
```

The frames collection is an associative array (like the forms collection we saw in Chapter 6), so each element can be accessed by either index or name. It's generally best to use the name (unless you have a good reason not to) so that you won't have to edit your code later if the frame order changes. By the same token, parent references in a complex nested frameset can change if the hierarchy changes, so I generally recommend that developers always start referencing from top. Of the above options, the reference I prefer, then, is top.frames['navigationFrame'].

Now that we have a reference to the frame, we can call a function in the other framed page:

```
File: frames-navigation.js (excerpt)
var navframe = top.frames['navigationFrame'];
navframe.callMyFunction();
```

Alternatively, we can get a reference to the other framed document, and work with the DOM from there:

```
File: frames-navigation.js (excerpt)
var navdoc = navframe.document;
var menu = navdoc.getElementById('menulist');
```

Discussion

Communication between frames is only allowed for documents in the same domain—for security reasons, it's not possible to work with a document that was loaded from a different domain than the script. It wouldn't do, for example, for a malicious site owner to load a site that you visit regularly into a frame, and steal the personal data you enter there.

In fact, some browsers let users disallow *all* scripts from communicating between frames, just to eradicate any possibility of a cross-site scripting vulnerability, and there's no way to work around this preference if your script finds itself running in a browser so configured.

If you do have users who are complaining of problems (and they can't or won't change their settings to allow cross-frame scripting), the safest thing to do is simply to avoid cross-frame scripting altogether.

Alternative methods of passing data between pages are discussed in Chapter 6 and Chapter 8.

Getting the Scrolling Position

Page scrolling is one of the least-standardized properties in JavaScript: *three* variations are now in use by different versions of different browsers. But with a few careful object tests, we can reliably get a consistent value.

Solution

There are three ways of getting this information. We'll use object tests on each approach, to determine the level of support available:

```
File: get-scrolling-position.js (excerpt)
function getScrollingPosition()
  var position = [0, 0];
  if (typeof window.pageYOffset != 'undefined')
    position = [
        window.pageXOffset,
        window.pageYOffset
    ];
  }
  else if (typeof document.documentElement.scrollTop
      != 'undefined' && document.documentElement.scrollTop > 0)
    position = [
        document.documentElement.scrollLeft,
        document.documentElement.scrollTop
    ];
  }
  else if (typeof document.body.scrollTop != 'undefined')
    position = [
        document.body.scrollLeft,
        document.body.scrollTop
    ];
  }
  return position;
}
```

The function can now be called as required. Here's a simple demonstration, using a window.onscroll event handler, that gets the figures and writes them to the title bar:



The Problem with scroll

scroll is not the most reliable of events: it may not fire at all in Konqueror or Safari 1.0, or when the user navigates with a mouse wheel in Firefox. And if it does fire, it may do so continually and rapidly (as it does in Internet Explorer), which can be slow and inefficient if the scripting you set to respond to the event is very complex.

If you have difficulties of this kind, you may find it better to use the setInterval function instead of an onscroll event handler. setInterval will allow you to call the function at a predictable interval, rather than in response to an event. You can find out more about this kind of scripting in Chapter 14, but here's a comparable example:

Discussion

The only real complication here is that IE 5 actually *does* recognize the documentElement.scrollTop property, but its value is always zero, so we have to check the value as well as looking for the existence of the property.

Otherwise, it doesn't really matter to us which browser is using which property; all that matters is that our script gets through one of the compatibility tests and returns a useful value. However, the properties used by each browser are shown here for reference:

- window.pageY0ffset is used by Firefox and other Mozilla browsers, Safari, Konqueror, and Opera.
- □ document.documentElement.scrollTop is used by IE 6 in standards-compliant mode.
- $\hfill \square$ document.body.scrollTop is used by IE 5, and IE 6 in "Quirks" mode.

This list doesn't tell the complete story, but it's intended primarily to describe the ordering of the tests. More recent Mozilla browsers (such as Firefox) also support documentElement.scrollTop and body.scrollTop, by the same render-

ing mode rules as IE 6. Safari and Konqueror support body.scrollTop in either mode. Opera supports all three properties in any mode!

But none of this is important for you to know—browser vendors add these multiple properties to allow for scripts that are unaware of one property or another, not to provide arbitrary choices for the sake of it. From our perspective, the important point is to settle on a set of compatibility tests that ensures our script will work as widely as possible.



Rendering Modes

"Standards" mode and "Quirks" mode are the two main **rendering modes** in use by current browsers. These modes affect various aspects of the output document, including which element is the canvas (<body> or <html>), and how CSS box sizes are calculated. For more on rendering modes, see Chapter 11.

Making the Page Scroll to a Particular Position

All current browsers implement the same (nonstandard) methods for scrolling a page. At least something here is simple!

Solution

There are two methods that can be used to scroll the page (or rather, the window or frame), either by a particular amount (window.scrollBy), or to a particular point (window.scrollTo):

```
File: scroll-page.js (excerpt)

//scroll down 200 pixels
window.scrollBy(0, 200);

File: scroll-page.js (excerpt)

//scroll across 200 pixels
window.scrollBy(200, 0);

File: scroll-page.js (excerpt)

//scroll to 300 from the edge and 100 from the top
window.scrollTo(300, 100);
```

```
File: scroll-page.js (excerpt)

//scroll to the beginning
window.scrollTo(0, 0);
```

These examples say: scroll down by 200 pixels, then across by 200 pixels, then to a point that's 300 pixels from the left and 100 pixels from the top, then back to the top corner.

Getting the Viewport Size (the Available Space inside the Window)

The details of the viewport size are needed for many kinds of scripting, wherever available space is a factor in the script's logic. This solution provides a utility function for getting the viewport size We'll be seeing the function again quite a few times throughout this book!

Solution

The properties we need are implemented in three different ways, like the properties we saw for page scrolling in the previous section ("Making the Page Scroll to a Particular Position"). As was the case in that example, we can use object testing to determine which implementation is relevant, including the test for a zero-value that we need in IE 5 (this test is required for the same reason: because, though the property exists, it isn't what we want):

```
file: get-viewport-size.js (excerpt)

function getViewportSize()

{
  var size = [0, 0];

  if (typeof window.innerWidth != 'undefined')
  {
    size = [
        window.innerWidth,
        window.innerHeight
    ];
  }
  else if (typeof document.documentElement != 'undefined'
        && typeof document.documentElement.clientWidth !=
        'undefined' && document.documentElement.clientWidth != 0)
  {
```

```
size = [
          document.documentElement.clientWidth,
          document.documentElement.clientHeight
];
}
else
{
    size = [
          document.getElementsByTagName('body')[0].clientWidth,
          document.getElementsByTagName('body')[0].clientHeight
];
}
return size;
}
```

The function returns an array of the width and height, so we can call it whenever we need that data:

```
File: get-viewport-size.js (excerpt)
window.onresize = function()
{
  var size = getViewportSize();
  alert('Viewport size: [' + size[0] + ', ' + size[1] + ']');
};
```

Summary

We've covered the basics of window and frame manipulation from a pragmatist's point of view in this chapter. We've also talked about principles and techniques that we can use to ensure that scripts like this are as user-friendly and as accessible as we can make them. Doubtless, this kind of work will remain controversial, and clearly we do need *some kind* of targeting mechanism, because even though the use of frames is slowly dying out, the advent of ever more sophisticated interfaces keeps these issues alive.

I rather like the XLink standard's show attribute, which has values like new and replace.⁵ These *suggest* a target process (open a new window, and replace the contents of the current window, respectively) but they don't actually *define* specific behaviors. They leave it up to the user agent to control what actually happens, so, for example, new could be used to open tabs instead of windows.

⁵ http://www.w3.org/TR/xlink/#show-att

13

Basic Dynamic HTML

Dynamic HTML isn't a single piece of technology that you can point to and say, "This is DHTML." The term is a descriptor that encompasses all of the technologies that combine to make a web page dynamic: the technologies that let you create new elements without refreshing the page, change the color of those elements, and make them expand, contract, and zoom around the screen.

DHTML uses HTML, the DOM, and CSS in combination with a client-side scripting language—JavaScript—to bring life to what was traditionally a static medium. In previous chapters, we learned that we can use JavaScript to manipulate parts of a page to achieve some very handy results. DHTML provides solutions to much more complex problems by assembling these parts into a coherent whole—one that satisfies real-world needs, rather than programming puzzles.

This chapter explores a few of the tools we need in order to create effective user interfaces with DHTML. It then discusses a couple of simple widgets in preparation for the more complex modules we'll consider throughout the rest of this book.

Handling Events

Any interaction that users have with a web page—whether they're moving the mouse or tapping the keyboard—will cause the browser to generate an event.

Sometimes, we want our code to respond to this interaction, so we listen for these events, which let us know when we should execute our code.

Solution

There are two ways to handle events: the short way, and the W3C way. Each has its pros and cons, but both allow you to execute a specified function when an event occurs on a particular element.

The Short Way: Using Event Handlers

The shorter way of handling an event is to use the DOM 0 event handlers that are assigned as shortcut properties of every element. Much as we saw in Chapter 5 when we discussed DOM 0 attribute shortcuts, these event handlers are not future-proof. However, they do offer some advantages over standard W3C event listeners:

- ☐ Every browser that's currently in operation supports DOM 0 event handlers without the need for code branching.
- ☐ Each function executed by a DOM 0 event handler has access to the exact element to which the event handler was assigned. (As you'll see later, this is not always available in W3C event listeners.)

The main problem with utilizing DOM 0 event handlers is that they are not designed to work with multiple scripts. Every time you assign a DOM 0 event handler, you overwrite any previously assigned handler for that event. This can interfere with the operation of multiple scripts that require event handling on the same element. With W3C event listeners, you can apply any number of event listeners on the same element, and enjoy the ability to remove any of them at any time.

If you can be certain that your code will not interfere with someone else's event handling (e.g., you're placing events on elements that are created dynamically in your own script), it will be safe to use DOM 0 event handlers. But—all things being equal—it is safer to use the W3C event listeners wherever practical, as we do in this book.

A number of DOM 0 event handlers are available via the browser; Table 13.1 lists the most commonly used handlers.

Table 13.1. DOM 0 event handlers

DOM 0 Event Handler	W3C DOM Event	Indicated Action
onblur	blur	Remove focus from an element by clicking outside or tabbing away from it.
onfocus	focus	Focus the cursor on an element.
onchange	change	Remove focus from an element after changing its content.
onmouseover	mouseover	Move the mouse pointer over an element.
onmouseout	mouseout	Move the mouse pointer out of an element.
onmousemove	mousemove	Move the mouse pointer while it is over an element.
onmousedown	mousedown	Press a mouse button while the pointer is over an element.
onmouseup	mouseup	Release a mouse button while the pointer is over an element.
onclick	click	Press and release the main mouse button or keyboard equivalent (Enter key) while the pointer is over an element.
ondblclick	dblclick	Double-click the main mouse button while the pointer is over an element.
onkeydown	keydown	Press a keyboard key while an element has focus.
onkeyup	keyup	Release a keyboard key while an element has focus.
onkeypress	keypress	Press and release a keyboard key while an element has focus.
onsubmit	submit	Request that a form be submitted.
onload	load	Finish loading a page and all associated assets (e.g., images).
onunload	unload	Request a new page to replace the currently-displayed page, or close the window.

In using DOM 0 event handlers, once you have a reference to the element whose events you want to handle, it's a simple matter of assigning a handling function to the appropriate property:

```
File: handle_events.js (excerpt)
var mylink = document.getElementById("mylink");

mylink.onclick = engage;
:
function engage()
{
   alert("Engage!");
   return false;
}
```

You'll note that, in the function assignment (button.onclick = engage;), parentheses do not follow the function name. Their inclusion would execute the function immediately, and assign the *return value* as the event handler. By omitting the parentheses, you can assign the function *itself* to the handler. This also means that you cannot supply arguments directly to the handling function: the function must obtain its information through other means.



Anonymous Functions

Instead of supplying a reference to a named function, you can supply an anonymous function for an event handler:

```
var mylink = document.getElementById("mylink");

mylink.onclick = function()
{
   alert("Engage!");
   return false;
}
```

Depending on whether you need to reuse the handling function (and your own coding preferences), this can be an easier way of writing event handling code.

The return value of the handling function determines whether the default action for that event occurs. So, in the preceding code, if mybutton were a hyperlink, its default action when clicked would be to navigate to its href location. By returning false, the engage function does not allow the default action to occur, and the hyperlink navigation will not take place. If the return value were true, the default action *would* occur after the event handling function's code had executed.

When an event occurs, detailed information about the how, why, and where of that event is written to an **event object**. In Internet Explorer, this takes the form of a global window.event object, but in other browsers the object is passed as an argument to the event-handling function. This difference is fairly easy to address within the handling function:

```
File: handle_events2.js (excerpt)

function engage(event)

{
   if (typeof event == "undefined")
   {
      event = window.event;
   }

   alert("The screen co-ordinates of your click were: " +
      event.screenX + ", " + event.screenY);

   return false;
}
```

The event object allows you to find out a range of details, such as which element was clicked, whether any keys were pressed, the coordinates of the event (e.g., where the cursor was located when the mouse button was clicked), and the type of event that triggered the function. Quite a few of the event property names are consistent across browsers, but a few differ. The Mozilla event properties can be viewed at the Gecko DOM Reference, while the Internet Explorer event properties can be seen at MSDN. For properties whose names vary between browsers, the potential for associated problems can normally be rectified with a little object detection; we'll discuss this in detail later in this chapter.

The W3C Way (Event Listeners)

Although the DOM 0 event handlers are quick and easy, they do have limitations (aside from the fact that eventually they will become deprecated). The main advantage of the W3C event listeners is that they natively support the addition and removal of multiple handling functions for the same event on a single element. Event listeners also have the capability to respond to events in several phases (though most browsers don't yet support this capability).

¹ http://www.mozilla.org/docs/dom/domref/dom_event_ref.html

² http://msdn.microsoft.com/workshop/author/dhtml/reference/objects/obj_event.asp

In the W3C specification, an event can be added to an element using the element's addEventListener method, but Internet Explorer for Windows chooses to use a method called attachEvent, which has a slightly different syntax.³

To add an event listener in every browser except Internet Explorer, you would write code similar to this:

```
var mylink = document.getElementById("mylink");
mylink.addEventListener("click", engage, false);
```

To support Internet Explorer, you'd need this code:

```
var mylink = document.getElementById("mylink");
mylink.attachEvent("onclick", engage);
```

As well as the differing function names, it's important to note that Internet Explorer uses the DOM 0 handler name for the event—"onclick"—rather than the true event name: "click". The extra argument that's supplied to addEventListener specifies whether the listener is applied during the capture (true) or bubble (false) event propagation phase. Event propagation is explained in more detail in the discussion below, but bubble is really the most useful choice, and ensures the same behavior in standards-compliant browsers as in Internet Explorer.

The differences between these two approaches are fairly easy to work around using an abstracting function. We can also provide a fallback for browsers that don't support W3C event listeners at the same time:

³Internet Explorer for Mac doesn't support either of these event models, so we have to rely on the DOM 0 handlers to work with events in this browser.

```
else
{
  eventType = "on" + eventType;

  if (typeof target[eventType] == "function")
  {
    var oldListener = target[eventType];

    target[eventType] = function()
    {
       oldListener();
       return functionRef();
      };
    }
  else
  {
      target[eventType] = functionRef;
    }
}
```

The first two if statements deal with the standards-based and Internet Explorer methods respectively, but the catch-all else deals with older browsers that don't support either of these methods, particularly Internet Explorer 5 for Mac. In this last case, a DOM 0 event handler is used, but to ensure that multiple functions can be used to handle a single event for a particular element, a closure is used to execute any existing functions that are attached to the event.

Closures are an advanced feature of JavaScript that relates to scoping (which you can read about in Chapter 19). Closures allow an inner function to reference the variables of the containing function even after the containing function has finished running. Simon Willison has explained their usage in relation to event handlers in some detail.⁴ Suffice it to say that closures allow us to stack multiple event handlers in browsers that don't support W3C event listeners.

The cross-browser code for assigning an event listener is as follows:

```
File: handle_events3.js (excerpt)
var mylink = document.getElementById("mylink");
attachEventListener(mylink, "click", engage, false);
```

⁴ http://www.sitepoint.com/blogs/2004/05/26/closures-and-executing-javascript-on-page-load/



Not (quite) the Genuine Article

Although the DOM 0 event handler fallback mimics the ability to add multiple event listeners for one event type on an element, it does not provide exact replication of the W3C event model, because specific handlers cannot be *removed* from an element.

Whereas DOM 0 handlers allowed the cancellation of an element's default action by returning false, W3C event listeners achieve this goal slightly differently. To cancel a default action in this model, we need to modify the event object. Internet Explorer requires you to set its returnValue property to false; standards-based implementations offer the preventDefault method to do the same thing. We can create a small function that figures out the difference for us:

```
file: handle_events4.js (excerpt)

function stopDefaultAction(event)
{
   event.returnValue = false;

   if (typeof event.preventDefault != "undefined")
   {
      event.preventDefault();
   }
}
```

We can call this function whenever we want to cancel the default action:

```
file: handle_events4.js (excerpt)

function engage(event)
{
   if (typeof event == "undefined")
   {
      event = window.event;
   }

   alert("Engage!");

   stopDefaultAction(event);

   return false;
}
```

You still need to return false after executing stopDefaultAction in order to ensure that browsers that don't support the W3C event model will also prevent the default action.



Safari and W3C Event Listeners

Due to a bug in Safari, it's impossible to cancel the default action of clicking a hyperlink in that browser when using W3C event listeners. To achieve the cancellation, you'll have to use DOM 0 event handlers with a return value of false.



Checking for attachEvent

Internet Explorer for Windows actually passes an event object to the event-handling function when attachEvent is used to attach an event listener. However, we still need to check for the existence of this object for any browsers that use the old event model.

One of the advantages of using W3C event listeners is that you can remove an individual listener from an element without disturbing any other listeners on the same event. This is not possible using the DOM 0 handlers.

Internet Explorer uses the detachEvent method, while the standards-compliant browsers instead specify a method called removeEventListener. Each of these methods operates fairly similarly to its listener-adding counterpart: an event type must be supplied along with the function that was assigned to handle that event type. The standard method also demands to know whether the event handler was registered to respond during the capture or bubble phase.

Here's a function that supports this approach across browsers:



The W3C Event Model and Anonymous Functions

The W3C event model doesn't allow for the removal of anonymous functions, so if you need to remove an event listener, hang onto a reference to the function in question.

In browsers that don't support W3C event listeners, this function removes all event handlers on the given event: it's not possible to remove just one of them and leave the others.

Discussion

Referencing the Target Element

Quite often, you'll want to use the object that was the target of an event inside the event handler itself. With DOM 0 event handlers, the use of the special variable this inside a handling function will refer to the event target object. Consider this code:

```
File: handle_events6.js (excerpt)
var mylink = document.getElementById("mylink");

mylink.onclick = engage;
:
function engage()
{
  var href = this.getAttribute("href");
  alert("Engage: " + href);
  return false;
}
```

Here, this refers to the link with ID mylink. We can use it to get the link's href attribute.

However, if you use W3C event listeners, the target of the event is stored as part of the event object, under different properties in different browsers. Internet Explorer stores the target as srcElement, while the standards model stores it as target. But the element to which these properties point isn't necessarily the element to which the event listener was assigned. It is, in fact, the deepest element in the hierarchy affected by the event. Take a look at the following HTML.

```
File: handle_events6.html (excerpt)

These are the voyages of the <a id="mylink" href="enterprise.html">starship Enterprise</a>.
```

If a click event listener were placed on the paragraph and a user clicked on the link, the paragraph's click event handler would be executed, but the event target that was accessible through the above-mentioned properties would be the hyperlink. Some browsers (most notably, Safari) even go so far as to count the text node *inside* the link as the target node.

We can write a function that returns the event target irrespective of which property has been implemented, but this does not solve the problem of finding the element to which we originally applied the event listener. Often, the best resolution to this quandary is to iterate upwards from the event target provided by the browser until we find an element that's *likely* to be the element to which we attached an event listener. To do this, we can perform checks against the element's tag name, class, and other attributes.

The abstracting event target function would look like this:

```
file: handle_events7.js (excerpt)

function getEventTarget(event)
{
   var targetElement = null;

   if (typeof event.target != "undefined")
   {
      targetElement = event.target;
   }
   else
   {
      targetElement = event.srcElement;
   }

while (targetElement.nodeType == 3 &&
      targetElement.parentNode != null)
   {
      targetElement = targetElement.parentNode;
}
```

⁵The W3C Standard specifies another property called **currentTarget**, which lets you get the element to which the listener was assigned, but there is no Internet Explorer equivalent. Browsers that support **currentTarget** also set up the event handler-style **this** variable with the same value, but again, without Internet Explorer support, this isn't particularly useful.

```
}
return targetElement;
}
```

The if-else retrieves the event target across browsers; the while loop then finds the first non-text-node parent if the target reported by the browser happens to be a text node.

If we want to retrieve the element that was clicked upon, we then make a call to getEventTarget:

```
File: handle_events7.js (excerpt)
var mylink = document.getElementById("mylink");
attachEventListener(mylink, "click", engage, false);
if unction engage(event)
{
   if (typeof event == "undefined")
   {
      event = window.event;
   }

   var target = getEventTarget(event);
   while(target.nodeName.toLowerCase() != "a")
   {
      target = target.parentNode;
   }

   var href = target.getAttribute("href");
   alert("Engage: " + href);
   return true;
}
```

Because we know, in this case, that the event-handling function will be attached only to links (<a> tags), we can iterate upwards from the event target, checking for a node name of "a". The first one we find will be the link to which the handler was assigned; this ensures that we aren't working with some element inside the link (such as a strong or a span).

Obviously, this method of target finding is not ideal, and cannot be 100% accurate unless you have knowledge of the exact HTML you'll be working with. Recently, much effort has gone into resolving this problem, and quite a few of the proposed solutions offer the same this variable as is available under DOM 0 event handlers, and in browsers that support the W3C Standard for event listeners (not Internet Explorer).

One such solution is to make the event listening function a method of the target object in Internet Explorer. Then, when the method is called, this will naturally point to the object for which the method was called. This requires both the attachEventListener and detachEventListener to be modified:

```
File: handle events8.js (excerpt)
function attachEventListener(target, eventType, functionRef,
    capture)
  if (typeof target.addEventListener != "undefined")
    target.addEventListener(eventType, functionRef, capture);
  else if (typeof target.attachEvent != "undefined")
    var functionString = eventType + functionRef;
    target["e" + functionString] = functionRef;
    target[functionString] = function(event)
      if (typeof event == "undefined")
        event = window.event;
      target["e" + functionString](event);
    };
    target.attachEvent("on" + eventType, target[functionString]);
  }
  else
    eventType = "on" + eventType;
    if (typeof target[eventType] == "function")
      var oldListener = target[eventType];
      target[eventType] = function()
```

```
oldListener();
        return functionRef();
      }
    }
    else
      target[eventType] = functionRef;
}
function detachEventListener(target, eventType, functionRef,
    capture)
  if (typeof target.removeEventListener != "undefined")
    target.removeEventListener(eventType, functionRef, capture)
  else if (typeof target.detachEvent != "undefined")
    var functionString = eventType + functionRef;
    target.detachEvent("on" + eventType, target[functionString]);
    target["e" + functionString] = null;
    target[functionString] = null;
  }
  else
    target["on" + eventType] = null;
```

This line of thinking was well represented in entries to Peter Paul Koch's improved addEvent competition. ⁶

Another solution by Dean Edwards totally eschews the W3C event model in favor of implementing DOM 0 event handlers with independent add and remove abilities.⁷

http://dean.edwards.name/weblog/2005/10/add-event/

⁶ http://www.quirksmode.org/blog/archives/2005/10/_and_the_winner_1.html

Although both of these solutions may prove to be well written and robust, they're largely untested as of this writing, so we'll stick with the approach whose flaws we know and can handle: the one presented in the main solution. Besides, in practice, the process of iterating to find an event's target isn't as unreliable as it may appear to be.

What is Event Bubbling, and How do I Control it?

You may have noticed that we needed to supply a third argument to the W3C Standard addEventListener method, and that a capture argument was included in our attachEventListener function to cater for this. This argument determines the phase of the event cycle in which the listener operates.

Suppose you have two elements, one nested inside the other:

```
<a href="untimely_death.html">Nameless Ensign</a>
```

When a user clicks on the link, click events will be registered on both the paragraph and the hyperlink. The question is, which one receives the event first?

The event cycle contains two phases, and each answers this question in a different way. In the **capture** phase, events work from the outside in, so the paragraph would receive the click first, then the hyperlink. In the **bubble** phase, events work from the inside out, so the anchor would receive the click before the paragraph.

Internet Explorer and Opera only support bubbling, which is why attachEvent doesn't require a third argument. For browsers that support addEventListener, if the third argument is true, the event will be caught during the capture phase; if it is false, the event will be caught during the bubble phase.

In browsers that support both phases, the capture phase occurs first and is always followed by the bubble phase. It's possible for an event to be handled on the same element in both the capture and bubbling phases, provided you set up listeners for each phase.

These phases also highlight the fact that nested elements are affected by the same event. If you no longer want an event to continue propagating up or down the hierarchy (depending upon the phase) after an event listener has been triggered, you can stop it. In Internet Explorer, this involves setting the cancelBubble

property of the event object to true; in the W3C model, you must instead call its stopPropagation method:

```
file: handle_events9.js (excerpt)

function stopEvent(event)
{
   if (typeof event.stopPropagation != "undefined")
   {
      event.stopPropagation();
   }
   else
   {
      event.cancelBubble = true;
   }
}
```

If we didn't want an event to propagate further than our event handler, we'd use this code:

```
File: handle_events9.js (excerpt)
var mylink = document.getElementById("mylink");
attachEventListener(mylink, "click", engage, false);
var paragraph = document.getElementsByTagName("p")[0];
attachEventListener(paragraph, "click", engage, false);
function engage(event)
{
   if (typeof event == "undefined")
   {
      event = window.event;
   }
   alert("She canna take no more cap'n!");
   stopEvent(event);
   return true;
}
```

Although we have assigned the engage function to listen for the click event on both the link and the paragraph that contains it, the function will only be called

once per click, as the event's propagation is stopped by the listener the first time it is called.

Finding the Size of an Element

There are so many variables that affect the size of an element—content length, CSS rules, font family, font size, line height, text zooming ... the list goes on. Add to this the fact that browsers interpret CSS dimensions and font sizes inconsistently, and you can never predict the dimensions at which an element will be rendered. The only consistent way to determine an element's size is to measure it once it's been rendered by the browser.

Solution

You can tell straight away that it's going to be useful to know exactly how big an element is. Well, the W3C can't help: there's no standardized way to determine the size of an element. Thankfully, the browser-makers have more or less settled on some DOM properties that let us figure it out.

Although box model differences mean that Internet Explorer includes padding and borders inconsistently as part of an element's CSS dimensions, the offsetWidth and offsetHeight properties will consistently return an element's width—including padding and borders—across all browsers.

Let's imagine that an element's dimensions were specified in CSS like this:

```
#enterprise
{
    width: 350px;
    height: 150px;
    margin: 25px;
    border: 25px solid #000000;
    padding: 25px;
}
```

We can determine that element's exact pixel width in JavaScript by checking the corresponding offsetWidth and offsetHeight properties:

```
File: find_size_element.js (excerpt)

var starShip = document.getElementById("enterprise");

var pixelWidth = starShip.offsetWidth;

var pixelHeight = starShip.offsetHeight;
```

In Internet Explorer 6, Opera, Mozilla, and Safari, the variable pixelWidth will now be set to 450, and the variable pixelHeight will be set to 250. In Internet Explorer 5/5.5, pixelWidth will be 350 and pixelHeight 150, because those are the dimensions at which the broken box model approach used in those browsers will render the element. The values are different across browsers, but only because the actual rendered size differs as well. The offset dimensions consistently calculate the exact pixel dimensions of the element.

If we did not specify the dimensions of the element, and instead left its display up to the default block rendering (thus avoiding the box model bugs), the values would be comparable between browsers (allowing for scrollbar width differences, fonts, etc.).



Attaining the Correct Dimensions

In order to correctly determine the dimensions of an element you must wait until the browser has finished rendering that element, otherwise the dimensions may be different from those the user ends up seeing. There's no guaranteed way to ensure that a browser has finished rendering an element, but it's normally safe to assume that once a window's load event has fired, all elements have been rendered.

Discussion

It is possible to retrieve the dimensions of an element minus its borders, but including its padding. These values are accessed using the clientWidth and clientHeight properties, and for the example element used above their values would be 300 and 100 in Internet Explorer 5/5.5, and 400 and 200 in all other browsers.

There is no property that will allow you to retrieve an element's width without borders *or* padding.

Finding the Position of an Element

Knowing the exact position of an element is very helpful when you wish to position other elements relative to it. However, because of different browser sizes,

font sizes, and content lengths, it's often impossible to hard-code the position of an element *before* you load a page. JavaScript offers a method to ascertain any element's position *after* the page has been rendered, so you can know exactly where your elements are located.

Solution

The offsetTop and offsetLeft properties tell you the distance between the top of an element and the top of its offsetParent. But what is offsetParent? Well, it varies widely for different elements and different browsers. Sometimes it's the immediate containing element; other times it's the html element; at other times it's nonexistent.

Thankfully, the solution is to follow the trail of offsetParents and add up their offset positions—a method that will give you the element's accurate absolute position on the page in every browser.

If the element in question has no offsetParent, then the offset position of the element itself is enough; otherwise, we add the offsets of the element to those of its offsetParent, then repeat the process for *its* offsetParent (if any):

```
file: find_position_of_element.js(excerpt)

function getPosition(theElement)
{
   var positionX = 0;
   var positionY = 0;

   while (theElement != null)
   {
      positionX += theElement.offsetLeft;
      positionY += theElement.offsetTop;
      theElement = theElement.offsetParent;
   }

   return [positionX, positionY];
}
```



IE 5 for Mac Bug

Internet Explorer 5 for Mac doesn't take the body's margin or padding into account when calculating the offset dimensions, so if you desire accurate measurements in this browser, you should have zero margins and padding on the body.

Discussion

The method above works for simple and complex layouts; however, you may run into problems when one or more of an element's ancestors has its CSS position property set to something other than static (the default).

There are so many possible combinations of nested positioning and browser differences that it's almost impossible to write a script that takes them all into account. If you are working with an interface that uses a lot of relative or absolute positioning, it's probably easiest to experiment with specific cases and write special functions to deal with them. Here are just a few of the differences that you might encounter:

- ☐ In Internet Explorer for Windows and Mozilla/Firefox, any element whose parent is relatively positioned will not include the parent's border in its own offset; however, the parent's offset will only measure to the edge of its border. Therefore, the sum of these values will not include the border distance.
- ☐ In Opera and Safari, any absolutely or relatively positioned element whose offsetParent is the body will include the body's margin in its own offset. The body's offset will include its own margin as well.
- ☐ In Internet Explorer for Windows, any absolutely positioned element inside a relatively positioned element will include the relatively positioned element's margin in its offset. The relatively positioned element will include its margin as well.

Detecting the Position of the Mouse Cursor

When working with mouse events, such as mouseover or mousemove, you will often want to use the coordinates of the mouse cursor as part of your operation (e.g., to position an element near the mouse). The solution explained below is actually a more reliable method of location detection than the element position detection method we discussed in "Finding the Position of an Element", so if it's possible to use the following solution instead of the previous one, go for it!

Solution

The event object contains everything you need to know to work with the position of the cursor, although a little bit of object detection is required to ensure you get equivalent values across all browsers.

The standard method of obtaining the cursor's position relative to the entire page is via the pageX and pageY properties of the event object. Internet Explorer doesn't support these properties, but it *does* include some properties that are *almost* the ones we want. clientX and clientY are available in Internet Explorer, though they measure the distance from the mouse cursor to the edges of the browser window. In order to find the position of the cursor relative to the entire page, we need to add the current scroll position to these dimensions. This technique was covered in Chapter 7; let's use the getScrollingPosition function from that solution to retrieve the required dimensions:

```
File: detect mouse cursor.js (excerpt)
function displayCursorPosition(event)
  if (typeof event == "undefined")
    event = window.event;
  var scrollingPosition = getScrollingPosition();
  var cursorPosition = [0, 0];
  if (typeof event.pageX != "undefined" &&
      typeof event.x != "undefined")
    cursorPosition[0] = event.pageX;
    cursorPosition[1] = event.pageY;
  }
  else
    cursorPosition[0] = event.clientX + scrollingPosition[0];
    cursorPosition[1] = event.clientY + scrollingPosition[1];
  }
  var paragraph = document.getElementsByTagName("p")[0];
  paragraph.replaceChild(document.createTextNode(
      "Your mouse is currently located at: " + cursorPosition[0] +
      "," + cursorPosition[1]), paragraph.firstChild);
```

```
return true;
}
```

clientX/clientY are valid W3C DOM event properties that exist in most browsers, so we can't rely on their existence as an indication that we need to use them. Instead, within our event handler, we test for the existence of pageX. Internet Explorer for Mac does have pageX, but it's an incorrect value, so we must also check for x. x is actually a nonstandard property, but most browsers support it (the exceptions being Opera 8+ and Internet Explorer). It's okay that Opera 8+ doesn't support x, because the else statement is actually a cross-browser method for calculating the mouse cursor position *except* in Safari, which incorrectly gives clientX the same value as pageX. That's why we still need to use both methods of calculating the cursor position.

Displaying a Tooltip when you Mouse Over an Element

Tooltips are a helpful feature in most browsers, but they can be a bit restrictive if you plan to use them as parts of your interface. If you'd like to use layers that appear when you want them to, aren't truncated, and can contain more than plain text, why not make your own enhanced tooltips?

Solution

For this example, we'll apply a class, hastooltip, on all the elements for which we'd like tooltips to appear. We'll get the information that's going to appear in the tooltip from each element's title attribute:

```
File: tooltips.html (excerpt)

These are the voyages of the <a class="hastooltip"
    href="enterprise.html" title="USS Enterprise (NCC-1701) ...">
    starship Enterprise</a>.
```

From our exploration of browser events earlier in this chapter, you'll probably already have realized that we need to set up some event listeners to let us know when the layer should appear and disappear.

Tooltips classically appear in a fixed location when you mouse over an element, and disappear when you mouse out. Some implementations of JavaScript tooltips also move the tooltip as the mouse moves over the element, but I personally find this annoying. In this solution, we'll focus on the mouseover and mouseout events:

```
File: tooltips.js(excerpt)
addLoadListener(initTooltips);

function initTooltips()
{
  var tips = getElementsByAttribute("class", "hastooltip");

  for (var i = 0; i < tips.length; i++)
  {
    attachEventListener(tips[i], "mouseover", showTip, false);
    attachEventListener(tips[i], "mouseout", hideTip, false);
  }
  return true;
}</pre>
```

We've already coded quite a few of the functions in this script, including addLoadListener from Chapter 1, getElementsByAttribute from Chapter 5, and the attachEventListener function that we created earlier in this chapter, so the bulk of the code is in the event listener functions:

```
file: tooltips.js(excerpt)

function showTip(event)
{
   if (typeof event == "undefined")
   {
      event = window.event;
   }

   var target = getEventTarget(event);

   while (target.className == null ||
        !/(^| )hastooltip( |$)/.test(target.className))
   {
      target = target.parentNode;
   }

   var tip = document.createElement("div");
   var content = target.getAttribute("title");
```

```
target.tooltip = tip;
target.setAttribute("title", "");
if (target.getAttribute("id") != "")
  tip.setAttribute("id", target.getAttribute("id") + "tooltip");
}
tip.className = "tooltip";
tip.appendChild(document.createTextNode(content));
var scrollingPosition = getScrollingPosition();
var cursorPosition = [0, 0];
if (typeof event.pageX != "undefined" &&
    typeof event.x != "undefined")
{
  cursorPosition[0] = event.pageX;
  cursorPosition[1] = event.pageY;
else
{
  cursorPosition[0] = event.clientX + scrollingPosition[0];
  cursorPosition[1] = event.clientY + scrollingPosition[1];
}
tip.style.position = "absolute";
tip.style.left = cursorPosition[0] + 10 + "px";
tip.style.top = cursorPosition[1] + 10 + "px";
document.getElementsByTagName("body")[0].appendChild(tip);
return true;
```

After getting a cross-browser event object, and iterating from the base event target element to one with a class of hastooltip, showtip goes about creating the tooltip (a div). The content for the tooltip is taken from the title attribute of the target element, and placed into a text node inside the tooltip.

To ensure that the browser doesn't display a tooltip of its own on top of our enhanced tooltip, the title of the target element is then cleared—now, there's nothing for the browser to display as a tooltip, so it can't interfere with the one we've just created. Don't worry about the potential accessibility issues caused by removing the title: we'll put it back later.



Controlling Tooltip Display in Opera

Opera still displays the original title even after we set it to an empty string. If you wish to avoid tooltips appearing in this browser, you'll have to stop the default action of the mouseover using the stopDefaultAction function from "Handling Events", the first section of this chapter. Be aware that this will also affect other mouseover behavior, such as the status bar address display for hyperlinks.

To provide hooks for the styling of our tooltip, we assign the tooltip element an ID that's based on the target element's ID (targetIDtooltip), and set a class of tooltip. Although this approach allows for styles to be applied through CSS, we are unable to calculate the tooltip's position ahead of time, so we must use the coordinates of the mouse cursor, as calculated when the event is triggered, to position the tooltip (with a few extra pixels to give it some space).

All that remains is to append the tooltip element to the body, so it will magically appear when we mouse over the link! With a little bit of CSS, it could look like Figure 13.1.

Figure 13.1. A dynamically generated layer that appears on mouseover



When the mouse is moved off the element, we delete the tooltip from the document, and it will disappear:

```
File: tooltips.js (excerpt)
function hideTip(event)
{
  if (typeof event == "undefined")
  {
```

Earlier, in showTip, we created a reference to the tooltip element as a property of the target element. Having done that, we can remove it here without needing to search through the entire DOM. Before we remove the tooltip, we retrieve its content and insert it into the title of the target element, so we can use it again later.



Do those Objects Exist?

You should check that objects created in other event listeners actually exist before attempting to manipulate them, because events can often misfire, and you can't guarantee that they will occur in a set order.

Discussion

One problem with the code above is that if the target element is close to the right or bottom edge of the browser window, the tooltip will be cut off. To avoid this, we need to make sure there's enough space for the tooltip, and position it accordingly.

By checking, in each dimension, whether the mouse position is less than the browser window size minus the tooltip size, we can tell how far to move the layer in order to get it onto the screen:

```
File: tooltips2.js (excerpt)
function showTip(event)
 if (typeof event == "undefined")
   event = window.event;
 }
 var target = getEventTarget(event);
 while (target.className == null ||
      !/(^| )hastooltip( |$)/.test(target.className))
   target = target.parentNode;
 }
 var tip = document.createElement("div");
 var content = target.getAttribute("title");
 target.tooltip = tip;
 target.setAttribute("title", "");
 if (target.getAttribute("id") != "")
   tip.setAttribute("id", target.getAttribute("id") + "tooltip");
 tip.className = "tooltip";
 tip.appendChild(document.createTextNode(content));
 var scrollingPosition = getScrollingPosition();
 var cursorPosition = [0, 0];
  if (typeof event.pageX != "undefined" &&
      typeof event.x != "undefined")
   cursorPosition[0] = event.pageX;
   cursorPosition[1] = event.pageY;
  }
 else
   cursorPosition[0] = event.clientX + scrollingPosition[0];
   cursorPosition[1] = event.clientY + scrollingPosition[1];
 tip.style.position = "absolute";
```

```
tip.style.left = cursorPosition[0] + 10 + "px";
tip.style.top = cursorPosition[1] + 10 + "px";
tip.style.visibility = "hidden";
document.getElementsByTagName("body")[0].appendChild(tip);
var viewportSize = getViewportSize();
if (cursorPosition[0] - scrollingPosition[0] + 10 +
    tip.offsetWidth > viewportSize[0] - 25)
  tip.style.left = scrollingPosition[0] + viewportSize[0] - 25 -
      tip.offsetWidth + "px";
else
{
  tip.style.left = cursorPosition[0] + 10 + "px";
}
if (cursorPosition[1] - scrollingPosition[1] + 10 +
    tip.offsetHeight > viewportSize[1] - 25)
  if (event.clientX > (viewportSize[0] - 25 - tip.offsetWidth))
    tip.style.top = cursorPosition[1] - tip.offsetHeight - 10 +
        "px";
  else
    tip.style.top = scrollingPosition[1] + viewportSize[1] -
        25 - tip.offsetHeight + "px";
  }
else
  tip.style.top = cursorPosition[1] + 10 + "px";
}
tip.style.visibility = "visible";
return true;
```

This function is identical to the previous version until we get to the insertion of the tooltip element. Just prior to inserting the element, we set its visibility to "hidden". This means that when it's placed on the page, the layer will occupy

the same space it would take up if it were visible, but the user won't see it on the page. This allows us to measure the tooltip's dimensions, then reposition it without the user seeing it flash up in its original position.

In order to detect whether the layer displays outside of the viewport, we use the position of the cursor relative to the viewport. This could theoretically be obtained by using clientX/clientY, but remember: Safari gives an incorrect value for this property. Instead, we use our cross-browser values inside cursorPosition and subtract the scrolling position (which is the equivalent of clientX/clientY). The size of the viewport is obtained using the getViewportSize function we created in Chapter 7, then, for each dimension, we check whether the cursor position plus the size of the layer is greater than the viewport size (minus an allowance for scrollbars).

If part of the layer is going to appear outside the viewport, we position it by subtracting its dimensions from the viewport size; otherwise, it's positioned normally, using the cursor position.

The only other exception to note is that if the layer would normally appear outside the viewport in both dimensions, when we are positioning it vertically, it is automatically positioned above the cursor. This prevents the layer from appearing directly on top of the cursor and triggering a mouseout event. It also prevents the target element from being totally obscured by the tooltip, which would prevent the user from clicking on it.



Measuring Visible Tooltip Dimensions

In order for the dimensions of the tooltip to be measured it must first be appended to the document. This will automatically make it appear on the page, so to prevent the user seeing it display in the wrong position, we need to hide it. We do so by setting its visibility to "hidden" until we have finalized the tooltip's position.

We can't use the more familiar display property here, because objects with display set to "none" are not rendered at all, so they have no dimensions to measure.

Sorting Tables by Column

Tables can be a mine of information, but only if you can understand them properly. Having the ability to sort a table by its different columns allows users

to view the data in a way that makes sense to them, and ultimately provides the opportunity for greater understanding.

Solution

To start off, we'll use a semantically meaningful HTML table. This will provide us with the structure we need to insert event listeners, inject extra elements, and sort our data:

File: sort_tables_by_columns.html (excerpt) <table class="sortableTable" cellspacing="0" summary="Statistics on Star Ships"> <thead> Star Ship Class Power Output (Terawatts) Maximum Warp Speed Captain's Seat Comfort Factor </thead> USS Enterprise NCC-1701-A 5000 6.0 4/10

First, we need to set up event listeners on each of our table heading cells. These will listen for clicks to our columns, and trigger a sort on the column that was clicked:

```
File: sort tables by columns.js (excerpt)
function initSortableTables()
  if (identifyBrowser() != "ie5mac")
    var tables = getElementsByAttribute("class", "sortableTable");
    for (var i = 0; i < tables.length; i++)
      var ths = tables[i].getElementsByTagName("th");
      for (var k = 0; k < ths.length; k++)
        var newA = document.createElement("a");
        newA.setAttribute("href", "#");
        newA.setAttribute("title",
             "Sort by this column in descending order");
        for (var m = 0; m < ths[k].childNodes.length; m++)</pre>
          newA.appendChild(ths[k].childNodes[m]);
        ths[k].appendChild(newA);
        attachEventListener(newA, "click", sortColumn, false);
  }
  return true;
```

Internet Explorer 5 for Mac has trouble dealing with dynamically generated table content, so we have to specifically exclude it from making any of the tables sortable.

Only tables with the class sortableTable will be turned into sortable tables, so initSortableTable navigates the DOM to find the table heading cells in these tables. Once they're found, the contents of each heading cell are wrapped in a hyperlink—this allows keyboard users to select a column to sort the table

by—and an event listener is set on these links to monitor click events, and execute sortColumn in response. The title attribute of each link is also set, providing the user with information on what will happen when the link is clicked.

The sortColumn function is fairly lengthy, owing to the fact that it must navigate and rearrange the entire table structure each time a heading cell is clicked:

```
File: sort_tables_by_columns.js (excerpt)
function sortColumn(event)
  if (typeof event == "undefined")
    event = window.event;
  var targetA = getEventTarget(event);
  while (targetA.nodeName.toLowerCase() != "a")
  targetA = targetA.parentNode;
  var targetTh = targetA.parentNode;
  var targetTr = targetTh.parentNode;
  var targetTrChildren = targetTr.getElementsBvTagName("th");
  var targetTable = targetTr.parentNode.parentNode;
  var targetTbody = targetTable.getElementsByTagName("tbody")[0];
  var targetTrs = targetTbody.getElementsByTagName("tr");
  var targetColumn = 0;
  for (var i = 0; i < targetTrChildren.length; i++)</pre>
    targetTrChildren[i].className = targetTrChildren[i].className.
        replace(/(^| )sortedDescending( |$)/, "$1");
    targetTrChildren[i].className = targetTrChildren[i].className.
        replace(/(^| )sortedAscending( |$)/, "$1");
    if (targetTrChildren[i] == targetTh)
      targetColumn = i;
      if (targetTrChildren[i].sortOrder == "descending" &&
          targetTrChildren[i].clicked)
        targetTrChildren[i].sortOrder = "ascending";
        targetTrChildren[i].className += " sortedAscending";
```

```
targetA.setAttribute("title",
          "Sort by this column in descending order");
    else
      if (targetTrChildren[i].sortOrder == "ascending" &&
          !targetTrChildren[i].clicked)
        targetTrChildren[i].className += " sortedAscending";
      }
      else
        targetTrChildren[i].sortOrder = "descending";
        targetTrChildren[i].className += " sortedDescending";
        targetA.setAttribute("title",
            "Sort by this column in ascending order");
      }
    }
   targetTrChildren[i].clicked = true;
 else
    targetTrChildren[i].clicked = false;
    if (targetTrChildren[i].sortOrder == "ascending")
      targetTrChildren[i].firstChild.setAttribute("title",
          "Sort by this column in ascending order");
    else
      targetTrChildren[i].firstChild.setAttribute("title",
          "Sort by this column in descending order");
    }
 }
}
var newTbody = targetTbody.cloneNode(false);
for (var i = 0; i < targetTrs.length; i++)</pre>
 var newTrs = newTbody.childNodes;
 var targetValue = getInternalText(
      targetTrs[i].getElementsByTagName("td")[targetColumn]);
```

```
for (var j = 0; j < newTrs.length; j++)</pre>
  var newValue = getInternalText(
      newTrs[j].getElementsByTagName("td")[targetColumn]);
  if (targetValue == parseInt(targetValue, 10) &&
      newValue == parseInt(newValue, 10))
  {
    targetValue = parseInt(targetValue, 10);
    newValue = parseInt(newValue, 10);
  else if (targetValue == parseFloat(targetValue) &&
      newValue == parseFloat(newValue))
    targetValue = parseFloat(targetValue, 10);
    newValue = parseFloat(newValue, 10);
  }
  if (targetTrChildren[targetColumn].sortOrder ==
      "descending")
  {
    if (targetValue >= newValue)
      break;
    }
  else
    if (targetValue <= newValue)</pre>
      break;
  }
if (j >= newTrs.length)
  newTbody.appendChild(targetTrs[i].cloneNode(true));
else
  newTbody.insertBefore(targetTrs[i].cloneNode(true),
      newTrs[j]);
}
```

```
targetTable.replaceChild(newTbody, targetTbody);
stopDefaultAction(event);
return false;
}
```

The first for loop that occurs after all the structural variables have been defined sets the respective states for each of the table heading cells when one of them is clicked. Not only are classes maintained to identify the heading cell on which the table is currently sorted, but a special sortOrder property is maintained on each cell to determine the order in which that column is sorted. Initially, a column will be sorted in descending order, but if a heading cell is clicked twice consecutively, the sort order will be changed to reflect an ascending sequence. Each heading cell remembers the sort order state it exhibited most recently, and the column is returned to that state when its heading cell is re-selected. The title of the hyperlink for a clicked heading cell is also rewritten depending upon the current sort order, and what the sort order would be if the user clicked on it again.

The second for loop sorts each of the rows that's contained in the body of the table. A copy of the original tbody is created to store the reordered table rows, and initially this copy is empty. As each row in the original tbody is scanned, the contents of the table cell in the column on which we're sorting is compared with the rows already in the copy.

In order to find the contents of the table cell, we use the function getInternalText:

```
file: sort_tables_by_columns.js(excerpt)

function getInternalText(target)
{
  var elementChildren = target.childNodes;
  var internalText = "";

  for (var i = 0; i < elementChildren.length; i++)
  {
    if (elementChildren[i].nodeType == 3)
    {
       if (!/^\s*$/.test(elementChildren[i].nodeValue))
        {
          internalText += elementChildren[i].nodeValue;
        }
    }
}</pre>
```

```
else
{
   internalText += getInternalText(elementChildren[i]);
}
}
return internalText;
}
```

getInternalText extracts all of the text inside an element—including all of its descendant elements—by recursively calling itself for each child element and concatenating the resultant values together. This allows us to access the text inside a table cell, irrespective of whether it's wrapped in elements such as spans, strongs, or ems. Any text nodes that are purely whitespace (spaces, tabs, or new lines) are ignored via a regular expression check.

When sortColumn finds a row in the copy whose sorted table cell value is "less" than the one we're scanning, we insert a copy of the scanned row into the copied tbody. For a column in ascending order, we simply reverse this comparison: the value of the row in the copy must be "greater" than that of the scanned row.

However, before a comparison is made, we check whether the contents of the sorted table cell can be interpreted as an integer or a float; if so, the comparison values are converted. This makes sure that columns that contain numbers are sorted properly; string comparisons will produce different results than number comparisons.

Once all of our original rows have been copied into the new tbody, that element is used to replace the old one, and we have our sorted table!

Using the sortableDescending and sortableAscending classes, which are assigned to the currently sorted table heading cells, we can use CSS to inform the user which column the table is sorted on, and how it is sorted, as shown in Figure 13.2 and Figure 13.3.

Figure 13.2. A sortable table sorted in descending order on the fourth column

Star Ship Class	Power Output (Terra Watts)	Maximum Warp Speed	Captain's Seat Comfort Factor
Ferengi Trading Vessel	500	4.0	8/10
USS Enterprise NCC-1701-D	6500	8.5	5/10
USS Enterprise NCC-1701-A	5000	6.0	4/10
Klingon Bird of Prey	3000	6.5	1/10
Class E Geo-stationary Satellite	2	0.1	0/10

Figure 13.3. A sortable table sorted in ascending order on the second column

Star Ship Class	Power Output (Terra Watts)	Maximum Warp Speed	Captain's Seat Comfort Factor
Class E Geo-stationary Satellite	2	0.1	0/10
Ferengi Trading Vessel	500	4.0	8/10
Klingon Bird of Prey	3000	6.5	1/10
USS Enterprise NCC-1701-A	5000	6.0	4/10
USS Enterprise NCC-1701-D	6500	8.5	5/10

Summary

The two main pillars of DHTML are the capturing of events, and the reorganization and creation of page elements via the DOM. Using these principles, it's possible to capture many of the different ways that users interact with a page and make the interface respond accordingly.

As can be seen by the number and quality of JavaScript-enhanced web applications that are now available, the features DHTML can bring to new interfaces represents one of the biggest growth areas for innovative JavaScript. The foundations and basic examples shown in this chapter give you a sense of the power that it can deliver inside a user's browser. We'll expand upon this further in the following chapters as we build some really interesting interfaces.

What's Next?

If you've enjoyed these chapters from *The JavaScript Anthology:* 101 Tips, Tricks & Hacks, why not order yourself a copy?

The JavaScript Anthology: 101 Tips, Tricks & Hacks is the ultimate toolkit for web developers using JavaScript. It's a collection of over 100 thoroughly-tested, customizable, and elegant solutions that will enable you to easily add usable and accessible interactivity to your site.

As JavaScript guru Bobby van der Sluis says, "The JavaScript Anthology is **the** cookbook of modern JavaScript, discussing only best practice solutions—a useful, timesaving, and practical reference for your desk."

The JavaScript Anthology: 101 Essential Tips, Tricks & Hacks also includes download access to all of the best practice code samples used throughout the book—plug them right into your own projects without any retyping!

In the rest of the book, you'll find solutions that will:

- Search and replace text using regular expressions
- Validate email addresses on your web forms
- Make a slideshow of images
- Make a style sheet switcher
- Build an accessible drop-down menu system
- Construct drag 'n' drop interfaces using AJAX
- Use JavaScript and Flash together
- Make your JavaScript accessible
- Use XMLHttpRequest to build AJAX applications
- Optimize your JavaScript code so that it runs faster
- And much more!

On top of that, order direct from sitepoint.com and you'll receive a free 17" x 24" poster of your choice!

Order now and get it delivered to your doorstep!

Index

	screen readers and, 436-456
Symbols	slider controls, 428-436
!= inequality operator, 49	tooltip display and, 402–411
!== non-identity operator, 50	ActionScript, 461
. wildcard character, 56	activate event, IE, 394, 397
== equality operator, 48	:active pseudo-class, 325
· · ·	ActiveX objects, 3, 468
=== identity operator, 50	(see also Flash; XMLHttpRequest
٨	object)
Α	Flash detection and, 458
abs method, Math class, 278	Flash version detection, 460
absolute positioning	FSCommand support and, 461, 463
browser differences, 248	memory leaks and, 556
CSS clip property and, 305	actuate event, 393
drop-down lists, 509	addDomFunction function, 562
iframe elements, 357	addEventListener method, 16, 234,
menus, IE, 332	243, 560
news ticker example, 299	addLoadListener function, 15
abstraction	accessible tooltip example, 405
direct referencing and, 520	adding a new style sheet, 226
object orientation feature, 516, 522,	auto-complete text fields, 507
549	clip–based transitions, 306
of tasks as functions, 548	custom dialog example, 483
Access Matters web site, 438	drag-and-drop effects, 282
accessibility	image swapping, 169
(see also keyboard accessibility; screen	soccer ball animation, 272
readers)	tooltip example, 251
attempted definition of, 386	WYSIWYG editor, 492
automatically initiated scripts, 441	addRule method, IE, 221, 226
current sub-branch display, 383	AJAX (Asynchronous JavaScript and
device-independent event handlers,	XML), 468
393–394	frameworks, 476
frames and, 135	keyboard accessibility, 401
hiding menu elements, 326	screen readers and, 446
keyboard and mouse, 395–402	Ajile module, 532
keyboard navigation and, 368	alert dialog
limitations of menus, 326	error analysis and, 23
non-programming aspects, 387	error messages, 119, 441
popups and, 129	page alternative, 25
	page arternative, 23

screen readers and, 449	example, 67
all property, document object	forms collection as, 104
accessible tooltip example, 405	image preloading, 168
browser detection and, 196	inverse sorting, 380
cleaning functions using, 558	multi-dimensional arrays, 66, 76
elements by attribute value, 98	radio button access, 110
alternate style sheets, 207, 211–212	select box access, 113
animated GIFs, 189	slideshow automation, 175
animation	sorting, 75, 77
achieving smoothness, 278-281	strings from, 71
applicable Flash techniques, 278	writing debugging data to, 25
automated slideshows, 173	arrow keys
drawing times, 280	accessible drag-and-drop functional-
frame rate changes, 279	ity, 400
optimization excluding, 536	accessible slider control, 433
realism in, 274	drop-down menus and, 508
scrolling news ticker, 298-305	key codes for, 424
soccer ball example, 272–278	keyboard accessible menus and,
straight line movement, 270	391, 411, 421, 424, 426, 428
transition effects, 305-311	arrow submenu indicators, 334, 337
anonymous functions	assistive technologies, 5
creating, 12	(see also screen readers)
DOM method loading, 562	associative arrays
event handlers and, 232	Flash version detection, 460
inline declaration, 269	forms collections as, 104
setInterval alternative, 273	frames collections as, 136
W3C event model and, 238	asterisks
antialiasing, 279	implying all elements, 405
appendChild method, 88, 92	in regular expressions, 56
arguments collection, 547	tag name wildcards, 98
arithmetic operators, 31–33	asynchronous processing
(see also Math class)	(see also AJAX)
array-literals, 66	load requests, 168
arrays, 65–78	open method requests, 471
adding or removing members, 72	updates and accessibility, 442, 453
alternate style sheets, 214	attachEvent method, IE
clock display, 183	addEventListener and, 16, 234, 243
code efficiency and, 550	attachEventListener and, 330
collections similar to, 83	checking for, 237
date and time comparisons, 164	circular references and, 559
Date object methods and, 153	event object and, 334
drop-down menus and, 503	

load event and, 560 attachEventListener function, 234, 241, 243 accessible rollover example, 396 accessible tooltip example, 404 circular reference cleaning, 559 click events and, 367 drag-and-drop effects, 282 drop-down menu example, 327, 330 screen reader identification, 450 tooltip example, 251	blur event listeners, 507 accessible rollover example, 396 accessible slider control, 432 blur events, 417, 511 blur method accessibility problems, 399 window object, 132 body element loading check, 560, 562 bold and italic text creation, 493 Boolean results, switch statements, 542 box model bugs, 246
attributes	box model calculations, 199
accessibility under DOM 0, 96	braces, 11
copying, 93	object literals use of, 71
reading and writing values, 95–98	typeof operator, problems with, 193
reading unverified, 97 retrieving elements with given, 98–	break statements, 116, 540, 542 browser detection, 194–198
100	(see also feature detection)
auto-complete text fields, 502	continuing need for, 191
automatic radix detection, 41	drag-and-drop effects, 282
automatically initiated scripts, 441	drop-down menu examples, 329,
	359, 510
В	feature detection alternative, 128,
back button problems, 479	192
background color slider, 317	identifyBrowser function, 197, 222, 226, 510
background images, 168, 334	screen readers, 369–370, 449
background masking, 486	when to use, 194
background-color property, 488	browser support
back-references, 62	addEventListener method, 17
backslash escaping, 45–46 backtraces, 21	advantages of feature detection, 192
backwards navigation and accessibility,	callback functions, 63
420	child selectors, 336
base of numbers, 41	currentTarget property, 239
baseOffset and extentOffset properties,	designMode property, 489
501	event listeners, 234
behavior layer, 514	Flash, 457–460, 464
behavioral pairing and accessibility, 395	JavaScript, 4
benchmarking tests, 545, 547	opacity property, 176–177, 180– 181, 488
best practices, 5, 453	ranges, 498, 502
block elements, 299	1411,800, 170, 302

scripting support by screen readers, 388, 437–449 scrolling, 139 style sheet manipulation, 217 XMLHttpRequest object, 192, 468– 469, 476 browser window (see viewport size) browser-based screen readers (see screen readers)	references to stored lengths, 537 rendering modes, 140 repeat rates, 435 scrolling behavior, 137, 428 sorting behaviors, 77 speaking browsers, 370 style sheet switchers, built-in, 211 substring detection benchmarking, 546
browsers (see also browser support; cross- browser scripting; Firefox; Inter- net Explorer; Konqueror; Nets- cape; Opera; Safari) absolute within relative positioning, 300 animation speed and, 281 argument fetching benchmarking, 547 attribute handling by, 96, 100 box model bugs, 246 computed style retrieval, 205 cookie restrictions, 148 CSS 2 property interfaces, 202 CSS property value separators, 308 Date object display, 152 DHTML Accessibility project and,	tabindex attribute and, 391 title attribute and, 403 viewport size calculation, 349 voice capabilities, 452 browser-specific optimizations, 538, 545–548 bubble phase, 243 button element accessible slider control, 430 keyboard accessibility and, 390 buttons custom dialog example, 484 disabling and accessibility, 400 WYSIWYG editor interface, 494– 495, 499 C caching
againes, 495 element positioning differences, 248 element size determination, 246 elements, hiding optional, 123 error reporting, built-in, 20 event models, 134, 233–234 external debuggers and, 26 focus event bubbling, 397 getSelection implementation, 497 grouped selector treatment, 220 keyboard accessible menus, 421 keyboard navigation modes, 411 popup resizing, 132	(see also preloading images) icons, 376 staggered loading alternative, 173 XMLHttpRequest and, 475 calculation, minimizing, 537 call method, Flash/JavaScript Integration Kit, 464 callback functions, 62 camel casing, 202 cancelBubble property, 243 capture phase, 243 caret, in regular expressions, 56 caret, in text selections, 501 carriage return character, 46

Cascading Style Sheets (see CSS)	code
case changes, 47, 219	(see also readability)
camel casing, 202	avoiding repetition, 550
case-insensitive flag, 54	compressing in production scripts,
ceil method, Math object, 32, 34, 188	552–556
chaining event handlers, 17	hiding, 13, 18
charAt method, 184	inserting custom, 499
checkboxes, 106	obfuscation, 18, 553
child selectors, CSS, 336	shortening for efficiency, 548–552
childNodes property, 85, 94	code efficiency (see optimization)
"chromeless" windows, 127-128	collapse method, 501
circular references, 556–560	collections
class attribute	checking loading, 562
access methods, 98, 100	DOM 0, 85
showing and hiding fields, 121, 123	from getElementsByTagName, 83
storing validation types, 118	color slider control, 317
class inheritance, 517–518, 526–528	color value normalization, 206
classes, multiple CSS, 100	comments
className property, 83, 100	code efficiency and, 548, 552
cleaning functions, 558	hiding code with HTML, 13
clearInterval function, 269, 274	removing URL protocols with, 554
clearMenus function, 343, 357	source code obfuscation and, 18
click event	communication interfaces (see data
device-independent event handling	transmission)
and, 394	compare function, 76, 380
transition effects, 305	compatMode property, document ob-
client-side language limitations, 2	ject, 198
clientWidth and clientHeight proper-	compressing script code, 18
ties, 246	computed styles, 204
clientX and clientY properties, IE, 249	conditions, compacting, 551
clip property, CSS, 305, 308	Connect Outloud screen reader, 445,
clip-based transitions, 305–311	448, 451
clocks, image-based, 181	consistent coding practice, 5
clone class, 295	constants, Math object, 32
cloneNode method, 91	constructors, 520, 525, 527
cloning objects by prototyping, 519,	contains methods
526	custom, for accessible drop-downs,
close method, window object, 131	418
closed property, checking, 129, 134	event target checking, 428
closures, 181, 235, 341, 531	proprietary IE, 332, 360

content (see dynamic content; separa-	current branch opening, 378
tion of content)	currentStyle property, IE, 205–206
Content-Type headers, 472	currentTarget property, 239
continue statements, 543	cursors (see mouse cursor)
control characters, 46	curtain transitions, 309
cookie property, document object, 144	custom code insertion, 499
cookies, 143–150	custom dialogs, 481–489
maintaining alternate style sheet	
states, 212	D
restricting access, 147	data transmission
setting expiry values, 146	requesting data from servers, 470
uses, 150	without XMLHttpRequest, 476–481
Coordinated Universal Time (UTC),	XMLHttpRequest and, 468–476
152, 154	data types, 16
createDialog function, 483, 485	arrays, 66
createElement method, 87	comparing unequal, 49
createElementNS method, 88	object literal properties, 71
createRange method, 497, 501	date format, cookie expiry, 146
createTextNode method, 88	Date object, 151–154
cross-browser scripting	calculating the day of the week, 162
accessibility, 436	compatible date formats, 161
computed style retrieval, 205–206	date and time comparisons, 152,
drag-and-drop functionality, 281	159–166
event listeners and, 235, 282	formatting by browsers, 152
mouse cursor position, 250, 257	formatting difference results, 164
style sheet modification, 220	formatting into sentences, 154–157,
cross-frame scripting, 135–137	165
CSS, 201–227	formatting methods, 153
(see also style sheets)	ISO date formats, 156
controlling element display, 121, 123	limits on values, 163
disabling optional elements, 124	meridian calculation, 158
opacity property in CSS 3, 180	Number function and, 40
opacity setting, 176–177, 180–181	string conversion, 37
pseudo-classes, 169, 325, 396, 404	time formatting, 157–159
System Colors, CSS 2, 403, 410	day of the week calculations, 162
tag uppercasing by IE, 219	debugging scripts, 19–29
target property, CSS 3, 133	deceleration in animation, 275
using multiple classes, 100	decorations, popup windows, 131
CSS1Compat value, 198	default case, switch statements, 540
cssFloat property, 202	deleteContents method, 501
currency values, 38	deleteRule method, 223
<i>,</i>	defendate filetion, 223

delimiters, 53–54	Document Object Model (see DOM)
designMode property, 489, 492	Dojo JavaScript framework, 476
detachEvent method, 237, 559	dollar sign, regular expressions, 56, 62
detachEventListener function, 241,	Dolphin Hal screen reader, 444, 448–
289, 450	451
DHTML, 229–266	DOM (Document Object Model), 9,
DHTML Accessibility project, 390,	79–102
392–393	cross-frame scripting, 137
DHTML controls	DHTML use, 229
accessible slider control, 428-436	element sizing properties, 245
scrolling news ticker, 298-305	methods, document loading and, 560
slider controls, 311–318	nodes and memory leaks, 556
DHTML menus, 321–383	W3C definition, 80
drop-down menu example, 323-361	DOM 0 functionality
expanding menus, 361–378	attributes as properties, 96
keyboard accessibility, 390, 392,	cleaning functions, 558
411–420	collections, 85
tabindex attribute and, 391	event handlers, 230, 558
usability, 421–428	DOMActivate event, 394
dialogs, custom in-page, 481–489	DOMFocusIn event, 397
digits (see numeric data)	dot property method, 97
directory paths, cookie, 148	double slash notation, 554
disability and accessibility, 386, 388	download times, 548
disabled property, style sheets, 208,	drag-and-drop effects, 281-290
215–216	accessible slider control, 401
disabling optional elements, 124	example interface, 289
display property	hot zone, 289
IE 5 and 6, 325	keyboard accessibility, 400
iframes, 478	reordering a list, 290–298, 400
screen reader identification, 369	drop sheets, 488
visibility and, 257, 368	drop-down menu example, 323–361
displayReset function, 369, 381	adding timers, 338–345
displayTime function, 184	constraining within windows, 345–
div elements	353
accessible tooltip example, 408	keyboard accessible version, 412-
changing a paragraph into, 91	423
nested divs, 313	select elements, 354-361
DOCTYPE declarations, 199	submenu arrows, 334–338
document object	drop-down menus
accessing forms from, 105	auto-complete text fields, 502
Opera load event listeners, 17	horizontal navigation and accessibil-
	ity, 426

keyboard accessibility, 411 positioning, 509	encryption, source code, 18 equality operator, 48, 50
dynamic content and screen readers, 390, 442, 444, 453	equivalence and accessibility, 389 error objects, 24
Dynamic HTML (see DHTML)	error reporting
dynamic variables, 537	built-in, 20–23
.,	external debuggers, 26
E	inline error messages, 119
ECMA–262 standard, 2, 461	page or window reporting, 25–26
editors	screen reader form validation, 441
browser engines for, 495	using alerts, 23–24
code optimization in, 552	using try/catch blocks, 24
example WYSIWYG, 489–496	escape characters
fully-functional, 496	formatting alerts, 24
efficient scripts (see optimization)	regular expressions, 54
element nodes, checking for, 87	special characters in strings, 46
elements	whitespace removal and, 555
accessing via the DOM, 82	escape function, 47
adding and removing multiple	cookies, 144
classes, 100	sub-cookie separators, 149
changing types of, 91–93	eval function, 543–544
creating editable, 489–496	event bubbling, 243
creating, using the DOM, 87–91	addEventListener method and, 16
default action cancellation, 236	drop-down menu example, 330, 344
dimensions, when rendered, 245-	expanding menu example, 367
246	focus events, 397
focus acceptance, 389	menu repositioning and, 349
insertion options, 89	event handlers
position of, when rendered, 246-	attribute, code in, 8
248, 348	behavioral pairing, 395
prototype-based method creation,	device-independence and accessibil-
523	ity, 393–394 multiple scripts and, 14
removing or relocating, 93–95	nonexistent elements, 10
repositioning, 348	XMLHttpRequest object, 472
retrieving by attribute value, 98–100	event handling approaches, 229–245
selecting all, 405	DOM 0 event handlers, 230
elements collection, 105, 118	W3C event listeners, 233
em elements, 441	event listeners
email address validation, 60, 115 emoticons, 499	checking object creation, 254
encapsulation, 516, 520, 522	cross-browser, 282
Circupsulation, 310, 320, 322	event handlers and, 16

location choice, 284	file extensions, 168, 175
removing, 237	findHere function, 379
event model, W3C, 238	Firefox browser
event models, browser, 134, 233-234	(see also Mozilla browsers)
event propagation, 234	CSS 2 System Colors and, 410
event target checking, 428	errors console, 21
event target property, 134	opacity support, 181
eventPhase property, 349	warnings console, 27
events	firstChild property, 85
keyboard accessibility and, 389	Flash, Macromedia, 457-465
speaking browsers, 371	detecting browser support, 457-460
execCommand method, 493	JavaScript animation and, 278
executeIframeRPC function, 478	JavaScript communication with, 461
execution order, operators, 32	screen reader alternative, 455
execution, stopping, 269	version detection, 458-460
expanding menus, 361–371	Flash/JavaScript Integration Kit, 464–
folder tree menus and, 361	465
indicating expanded branches, 371–	flickering, 213, 301
376	float property, CSS, 202, 325
restricting open branches, 377–378	floor method, Math object, 32, 34, 36
Expires header, XMLHttpRequest, 475	fly-out menus (see drop-down menus)
expiry dates and times, cookies, 146	focus
expressions	accessible tooltip display on, 402-
applying CSS rules in IE, 336	411
direct evaluation of, 551	keyboard accessibility and, 389
external debuggers, 26	tabindex attribute and, 391
external dependencies, loading, 560	focus event listeners
_	accessible drop-down menu, 412-
F	413, 418–419
fading effects, 176–181	accessible rollover example, 396
cross and straight fades, 181	accessible slider control, 432
feature detection, 128, 192–194	accessible tooltip example, 404
(see also browser detection)	source of focus events, 394, 434
ActiveX objects, 469	focus method
browser detection alternative, 194	accessible form validation, 398, 440
cursor position detection, 249	misuse, 399
omission of typeof operator, 193	opening new windows, 132
opacity property support, 180	remote scripting accessibility, 447
scroll position example, 137, 139	validation errors and, 399
style sheet creation, 227	:focus pseudo-class, 325
viewport size example, 141	

folder tree menus	functions
accessibility, 411	abstraction and, 548
example script, 374	assignment to event handlers, 232
expanding menus and, 361	creating with prototype objects, 523
indicating expanded branches, 371–	derivation of objects from, 519
376	execution order, 524
restricting open branches, 377–378	introduced, 8
font size, custom tooltips, 410	variable access in nested, 530
for attribute, accessing, 98, 100	variable scope and, 528
for loops	
avoiding repetition using, 550	G
caching images, 168	g (global flag), 54, 62
nested, 67	garbage collection, 556
node structure and, 92	gecko browsers, 196
radio button access, 111	(see also Mozilla)
validating radio buttons, 116	get* methods, Date object, 153, 156
for-in iterators, 24	getAttribute method, 95, 98, 135
form element, 430	getAttributeNS method, 88
form validation, 113–121	getComputedStyle method, 205–206
example script, 117	getDate method, 153
form submission and, 116	getDateOrdinal method, 156
inline error messages, 119	getDateString method, Date object,
keyboard accessibility, 398	155–156, 159
mandatory text fields, 113	getElementById method, 9
screen reader accessibility, 440	accessing elements with, 28, 82
validating several fields, 117	browser detection and, 196
forms collection, 104, 106	getElementsByTagName and, 84
forms processing, 103–125	warnings from testing for, 28
displaying and hiding fields, 121	getElementsByAttribute function, 98
validating before submission, 116	tooltip example, 251
forward slash delimiter, 54	transition effect, 306
frame rates, animation, 279	WYSIWYG editor, 492
frames collection, 136	getElementsByTagName method, 82
frames, communicating between, 135–	DOM 0 properties and, 85
137	iterating through elements, 98, 204,
FSCommand feature, Flash, 461, 464	473
fscommand function, ActionScript, 462	getEventTarget function, 239
function literals, 12	auto-complete text example, 511
function pointers, 269	drag-and-drop effects, 284
function references, 306	transition effect, 306
functional loops, 268	getHours method, Date object, 158
•	Sections metrod, Date Object, 150

tooltip positioning, 256–257
hiding code, 13, 18
highlighting selections, 330, 371, 496
history of JavaScript, 2
Home Page Reader, 441, 444, 448-
449, 451
horizontal navigation bars, 426
horizontal overflow, 350
horizontally collapsing transitions, 309
hot zone, drag-and-drop effects, 289
href attribute, 382, 390
HTML
equivalent DOM hierarchy, 81
Flash and, 457
forms collection and, 106
menu examples, 322
hyphens in style attributes, 202
I
i (case-insensitive flag), 54
IBM Corporation, 392
icons
accessible drag-and-drop functional-
ity, 401
caching, 376
folder tree menus, 362, 371, 373
identifyBrowser function, 197, 222,
226, 510
identity operator, 50
If-Modified-Since header, XMLHttpRe-
quest, 475
iframes
data transmission using, 476-481
menu display and, 354
WYSIWYG editor and, 490
image collection, 167
image swapping, 169
image-based clock, 184
random display, 171
image-based clock, 181–186
images, 167–189

fading in and out, 176–181	deleting style sheet rules, 223
inserting, with the WYSIWYG edit-	eventPhase property support, 349
or, 493	float property, ÎE 5, 325
preloading, 167	FSCommand and, 463
slideshow automation, 173	garbage collection problems, 556
staggered loading, 173	getSelection alternative, 497
in command, 69	insecure page warnings, 358
index pages, default, 382	missing DOCTYPE declarations and
indexes	IE 6, 199
arrays, 65, 69	mouse cursor position, 249
multi-dimensional arrays, 67	opacity support, IE 5, 180, 488
radio button access, 110	references to stored lengths, 537
select box access, 113	relatedTarget support, 345
string index numbering, 51-52	relative positioning quirk, 331
style sheets, 217	screen readers and, 436
using form id tags as, 105	scrollTop property and IE 5, 139
indexOf method, 51-52, 545-546	setAttribute method and, 96
inequality operator, 49	tag name uppercasing, 219
inheritance, 517–518, 526–528	XMLHttpRequest support, 468–469
initAutoComplete function, 507	Internet Explorer for the Macintosh
initDialog function, 483	chaining event handlers, IE 5, 17
inline error messages, 119	distinguishing from IE for Windows,
inline style sheets, 224	196
inner scopes, 330	drag-and-drop effects and IE 5, 282
innerHTML property, 25, 477–478,	dynamically generated content, IE
493	5, 259, 484, 510
input element, 430	element sizing bug, IE 5, 247
insecure page warnings, 358	event listener support, 234
insertBefore method, 89, 94, 297	memory leaks in IE 5.0, 54
insertNode method, 501	setTimeout function and IE 5, 306,
insertRule method, 221–222, 226	344
interactive scripting, 267	slider control and IE 5, 313
interfaces (see user interfaces)	style switching and IE 5, 211
Internet Explorer	timing functions and IE 5, 269, 273
(see also attachEvent method)	Internet Explorer for Windows
:active pseudo-class, IE 5 and 6, 325	activate event, 394
asterisk wildcard support, 98	alternate style sheet bug, 215
attribute copying, 93	array function support in IE 5.0, 72–
browser detection, 197, 329	73
computed styles and, 205	asterisk notation and IE 5, 405
contains method, 332, 361	child selector support, 336
	distinguishing from IE for Mac, 196

drag-and-drop bug, 295	screen readers and, 437
errors console, 22	keyCode property
expression syntax, 337	repeat rates and, 435
Flash support, 458 IE 5.0, positioning, 301	testing for arrow key events, 424, 427–428
iframe support, IE 5, 359, 477, 492 navigator.plugins and, 459	testing for the Tab key, 369, 371, 450
positioning in IE 5.0, 299	keydown event, 369, 428
rules property, 217	keydown event listeners, 433, 507
select elements, 354, 358	keypress event, 428
interpreter, efficient use, 11, 537, 544,	keypress event listeners, 507
548	keyup event, 369
intuitive values, 175	keyup event listeners, 433, 450
inverted color scheme style sheet, 212 isNaN function, 41	Konqueror browser, 96, 139, 196–197, 345
iteration and code efficiency, 550	
,,	L
I	
Java LiveConnect module, 461–462	label element, 120, 441 lang pseudo-class, 404
JAWS screen reader, 441, 444, 448,	language attribute, script tag, 14
451, 455	lastChild property, 86
join method, 71	lastIndexOf method, 52
JSON (JavaScript Object Notation)	leap years, 162
format, 481	left property, style object, 420
Tormat, 101	length property, 53
K	iterating through arrays, 68
keyboard accessibility, 389–393	iterating through collections, 537
drag-and-drop functionality, 400	limitations, 69
form validation, 398	push function workaround, 73
menu usability, 421–428	limitations of JavaScript, 2
menus, 390, 392, 411–420	line breaks, 24, 553, 555
mouse accessibility combined with,	line feed character, 46
395–402	link element, 207
multiple navigation modes, 411	links
scripted rollovers, 396	creation, with the WYSIWYG editor,
simulating the experience, 389	493
slider controls, 428–436	insertion, DOM methods, 89
starting from scratch, 395	keyboard accessibility and, 390
user needs, 385, 388	navigation and screen reader identi-
keyboard navigation, 368	fication, 450
menu repositioning and, 425	opening in new windows, 133–135

screen reader identification of, 439 styled links in slider controls, 430	menus, 321 accessibility, 326
list item mouseout function, 332, 360	adding timers, 338
list item mouseover function, 330, 347,	closing, 417
349	drop-down menu example, 323–361
lists (see ordered lists; unordered lists)	expanding menus, 361–371
LiveConnect module, Java, 461–462	functional types, 321
load event	keyboard accessibility, 411–420
rendering completion and, 246	keyboard usability, 421–428
running scripts before, 560–563	nested submenus, 412
load event handler	repositioning, 345, 350
multiple script problems, 15	stacking, 351
script location, 10	method creation, 521–526
loading scripts, 12	methods, overriding, 74
local time defined, 152	MIME type, 88
LocalConnection function, Action-	modal interaction, 454
Script, 464	modifiers, regular expression, 54
location property, document object,	modulus operator, 31, 43
447	motion effects, 270–281
looping efficiently, 537, 542	(see also animation)
(see also for loops)	slider controls, 311-318
•	user control over, 302
M	mouse cursor
m (multi-line flag), 55	appearance change, 283
Macintosh versions of IE (see Internet	position detection, 248–250, 513
Explorer for the Macintosh)	mouse events, screen readers, 370
Macromedia Corporation (see Flash)	mouse movements
mandatory text fields, 113	adding timers to menus, 338
Math class	threshold values, 285–286, 293
abs method, 278	mousedown event listeners, 511
built-in operators, 32	mousedown events, 450
ceil method, 188	mouseout event listeners
floor method, 36	accessible slider control, 434
properties, 32	menu timers, 339
random method, 32, 35	removing iframe elements, 356
round method, 35	mouseover effects
mathematics (see numeric data)	accessible tooltip display, 402
matrixes, 66–67, 76	image swapping, 169
media attribute, 215	style sheet rule for, 222
media types, styling, 226	tooltip display, 250–257
memory leaks, 54, 556-560	mouseover event listeners
	accessible drop-down menu, 419

accessible slider control, 434	Netscape, 2, 462
creating iframe elements, 355	news ticker example (see scrolling news
menu timers, 339	ticker)
mouseover event sources, 394, 408, 450	nextSibling property, 86
movement (see animation; motion ef-	nodeName property, 87
fects; mouse movements)	nodes, DOM
moveObject function, 272–275	cloning, 91
Mozilla browsers	iterative change warning, 92
(see also Firefox)	node types, 79
browser detection, 197	relational properties, 85
distinguishing Safari from, 196	whitespace nodes, 86
	nodeType property, 87
focus event bubbling, 397	• • • •
script timeouts, 546	nodeValue property, 88
strict warnings and, 545	non-identity operator, 50
MSXML parser, 469	noscript element, 6
multi-dimensional arrays, 66–67, 76	Number function, 40
multi-line flag, regular expressions, 55	numeric data, 31–44
multiple inheritance, 527	adding ordinal suffixes, 42
multiple scripts	base detection, 41
event handlers and, 14, 230	converting dates to strings, 37
event listeners and, 233	converting numbers to strings, 36–
	38
N	converting strings to, 39–42
named arguments, 547	currency values, 38
namespaces, 88, 531–532	random numbers, 35
naming conflicts, 531, 555	rounding numbers, 33
NaN (Not a Number) value, 41	sorting and compare function, 75
navigation using lists, 322	sorting arrays, 76
	sorting in tables, 264
(see also keyboard navigation; menus)	string concatenation risks, 37
navigator object properties, 196, 459	testing for, 41, 58
browser detection and, 194, 196,	text field validation, 114
554	tere field validation, 111
nesting	0
event bubbling and, 243	
nested closures, 341	obfuscation, source code, 18, 553
nested divs, 313	object based scripting, 71, 518
nested for loops, 67	object detection (see feature detection)
nested functions, variable access, 530	object orientation, 515–533
nested lists, 323	code efficiency and, 549
nested submenus, 412	example script, 519–520
ternary operators, 539	method creation, 521-526

modelling inneritance, 326–328	Opera browser
object based code and, 518–519	absolutely positioned elements, 300-
principles and benefits, 515–518	301
object reference creation, 543	attribute leading spaces, 331
object-literals, 70	Content-Type headers, 472
objects	detection, 196–197
checking the existence of, 532	spatial navigation features, 403,
created in other event listeners, 254	407, 421
replication by cloning, 519	tooltip display, 253
storing references to, 536	window sizing, 132
warnings connected with, 28	operating systems
offleft positioning, 326	browser detection and, 197
accessible slider control, 429	distinguishing between IE versions,
hiding menus, 327, 368	196
optional questions, 123	GUI behaviors, 391
overriding, 381	operator precedence, 32
screen readers and, 440	operators, mathematical, 31–33
offset dimensions bug, IE 5 for Mac,	optimization, 535–564
247	anticipating load events, 560–563
offsetHeight property, 245	avoiding memory leaks, 556–560
offsetLeft property, 247, 352	browser-specific optimizations, 545
offsetParent property, 247	compressing production scripts, 552-
offsetTop property, 247	556
offsetWidth property, 245, 301, 332,	concise coding, 548-552
352	faster scripts, 536–548
on* event handlers, 230	looping efficiently, 537, 542
(see also * events)	ordered lists, 290–298
onclick event handler, 124	ordinal numbers, 42, 156
online application design, 467–514	overflow property, 301
frameworks, 476	overline text decoration, 218, 222
onload event handlers, 328	overriding
hiding optional elements, 123	classnames, 531
preloading images, 170	methods, 74, 517–518
progress indicator, 188	multiple scripts and, 10, 14
onmousedown event handlers, 495	styles, 226
onscroll event handler, 138	variables, 530, 555
onsubmit event handler, 116	
opacity property, CSS, 176–177, 180–	P
181, 295, 488	page dimensions, 486
open method	(see also viewport size)
window object, 131, 134	page load event, 213
XMLHttpRequest object, 471, 473	, , , , , , , , , , , , , , , , , , ,

page requests, individual, 454	menu repositioning, 345, 425
pageX and pageY properties, 249	offleft positioning, 123, 326–327,
pageYOffset property, window object,	368, 381, 429, 440
139	position detection, 348
paragraphs, changing to divs, 91	tooltips, 254, 408
parentheses, effects, 33, 38	pow method, Math object, 32
parentNode property, 86	preferred style sheets, 211
parseFloat function, 40, 114	preloading images, 167
parseInt function, 40–41, 114	image swapping, 170
pasteHTML method, 502	image-based clock, 182
path setting, cookies, 148	progress indicator, 186
per cent sign	presentation (see separation of con-
modulus operator, 31	tent)
URL coding, 47	preventDefault method, 236
performance of scripts (see optimiza-	previousSibling property, 86
tion)	private members, 516, 518
persistent style sheets, 207–208, 211	probability distributions, 173
phases, event cycle, 243	processing power and animation, 279-
phone numbers, 59	280
photographic slideshows, 173	processor latency, 184
pipe character, 56	progress indicators, 186
pixels, normalization to, 206	progressive enhancement, 5–7, 439,
placeholders, 170	455
plugins (see Flash, Macromedia)	properties, direct referencing, 520
plugins property, navigator object, 459	property creation, object oriented, 520
plus sign, in regular expressions, 56	Prototype JavaScript framework, 476
polymorphism, 518	prototyping, 74, 527
popups, 128–133, 481–489	cloning objects by, 526
error reporting to, 25	method creation using, 522
ethical use, 129	methods for built-in objects, 525
usability and accessibility, 128	mimicking inheritance, 518–519
position detection	object prototyping, 154
animation and, 273	prototype object, 523
elements, 246–248	prototype object functions, 523
mouse cursor, 248–250	prototype property, 523
position inversion, 350	pseudo-classes, CSS, 169, 325, 396,
position property, CSS, 248	404
position rounding, 350–351	pseudorandom numbers, 35
positioning	push method, 72
(see also absolute positioning)	
list items with CSS, 291	

Q	matching text in strings, 57
qualified values, href attributes, 382	searching for and replacing text, 61
question mark, regular expressions, 56	special characters, 57
Quirks mode, 140, 198–199	substring location test, 545
quotes, 45–46, 71	testing for email addresses, 60, 115
	testing for leading spaces, 331
R	testing for numeric data, 58
radio buttons, 108–109, 115	testing for phone numbers, 59
random image display, 171–172	testing for whitespace, 114, 264
random method, Math object, 32, 35	rel attribute, 133–134, 215
random numbers, 35	related property, 557
random sorting, 77	relatedTarget property, 345, 417
	relative positioning, 248, 299, 331
ranges auto-complete text fields, 502	remote procedure calls (see data trans-
browser support, 498	mission)
cursor position and, 513	remote scripting
getSelection alternative, 497	individual page requests and, 454
specifying limits of, 501	keyboard accessibility, 401
readability of code	screen readers and, 446
braces and semicolons, 11	removeChild method, 93
compacting conditions and, 552	removeEventListener method, 237
nested operators, 540	removeRule method, IE, 223
string concatenation and, 37	rendering modes, 139–140, 198–199
readyState property, XMLHttpRequest,	repeat rates, key events, 435
472	replace method, 62, 478
recursive functions, 264, 333, 383	replaceChild method, 89–90
redirects, accessibility and, 442	repositioning (see positioning)
referencing	reset functions, 377
circular references, 556	resizing swapped images, 170
direct referencing, 520	responseText property, XMLHttpRe-
eval function, 543	quest, 474
frequently used objects, 536	responseXML property, XMLHttpRe-
function definition and, 522	quest, 473
function references, 306	retrieveComputedStyle function, 273
RegExp class, 54–55	return Value property, 226
regular expressions, 53–63	returnValue property, 236
className property retrieval, 101	rollover effects, 396, 439
comment and whitespace removal,	rollover styles, 330, 336
553	round brackets, 56, 62
Flash version detection, 460	round method, Math object, 32, 34–35 rounding numbers, 33, 38
indexOf and, 53, 546	Touriding numbers, 33, 36

rules property, IE, 217	tricks and hacks, 449
_	user needs, 385, 388, 454–455
S	script element, 12, 14
Safari browser	scripts
cancelling link defaults, 237, 483, 495	anticipating load events, 560–563 concise coding, 548–552
CSS 2 System Colors and, 403, 410	faster running, 536–548
detection, 197	inside iframes, 480
distinguishing from Mozilla, 196	multiple, and DOM 0 event hand-
DOM support limitations, 221–223,	lers, 230
226	timing out, 546
events from text nodes, 344	Web version optimization, 552–556
href values, 382	scrollBy method, window object, 140
input element problem, 430	scrolling
lang pseudo-class, 404	menu repositioning and, 353
scroll event problems, 139	prevention, accessible menu example,
setTimeout support, 344	428
stopDefaultAction function and, 368	scroll position, 137–141, 249
stylesheet collection, 217	scrolling news ticker, 298–305
Safari Enhancer, 20	screen readers and, 442, 445
Sajax JavaScript framework, 476	user control, 302, 305
scope (see variable scopes)	scrollTo method, window object, 140
screen readers	scrollTop property, 139
accessible scripts for, 436–456	security
current sub-branch display, 383	cross-frame scripting, 137
detection through events, 369-370	iframes and, 480
Flash alternative, 455	restrictions on JavaScript, 3
form validation, 440	XMLHttpRequest and, 471
hiding menu elements, 326	select boxes, 111
identification, 449	select elements, 354, 358, 442
link identification by, 439	selectedIndex property, 113
menu accessibility, 392	selectionStart property, 513
modal interaction and, 454	semicolon terminator, 11, 553
problems with dynamic content,	send method, XMLHttpRequest object,
390, 442, 444, 453	471
products listed, 436	separation of content, style, and behavior, 8–11, 321, 323
reading label text, 441 remote scripting and, 446	status of navigation arrows, 337
scripting support, 388, 437–449	separators
simulating the user experience, 436	className property, 100, 102
suggested best practice, 453	CSS property values, 308
suggested best practice, 133	sub-cookies, 149

serif text style sheet, 209, 212	soccer ball animation, 272-278
server XMLHttpRequests, 470	sort method, 75-77
server-side scripting, 3, 182	sorting
set* methods, Date object, 154	drag-and-drop reordering, 291
setAttribute method, 95, 98	list items, real-time effect, 297
setInterval function, 183	random sorting, 77
alternative to onscroll, 139	stable sorts, 77
alternatives, 273	table sorting , 257–265
assessing document loading, 562	source code visibility, 1, 18
debugging and, 26	source order execution, 213
setTimeout compared to, 267	source order indexing, 217
soccer ball animation, 272	spaces
stopping execution, 269	className property, 100, 102
setSelectionRange method, 513	global removal, dangers, 554
setTimeout function, 175	underscores conversion to, 525
accessible slider control, 435	span element, 186
accessible tooltip example, 408	speaking browsers, 370
animation example, 180	special characters
auto-complete text example, 507,	avoiding in cookies, 144
513	escaping in strings, 45
clip–based transitions, 306	regular expressions, 57, 101
iframes and, 479	URLs, 47
menu timers, 339, 341	splice method, 72
setInterval compared to, 267	split method, 53, 145
style sheet maintenance script, 213	spoofing, 194
WYSIWYG editor, 492	sqrt method, Math object, 32
shopping cart applications, 34, 281	square brackets, 57, 65, 544
shortcuts	src property, 358
DOM 0 attributes, 230	stable sorting, 77
forms collection, 104	staggered loading, 173
ternary operator, 131	standardization, 2, 156
show attribute, XLink, 142	Standards mode, 139–140, 198–199
shrinking transitions, 310	static elements, 287
sidebar property, window object, 196	static HTML, 6
single-letter variable names, 556	status argument, window.open, 132
slider controls, 311–318	status bars, 186
accessible drag-and-drop functional-	stop button, news ticker, 302, 305
ity, 401	stop method, ActionScript, 462
example appearance, 316, 318	stopDefaultAction function, 236
fixed values, 315	drag-and-drop effects, 287
keyboard accessibility, 428–436	Opera tooltip display, 253
slideshows, 173	Safari bug, 368

slider control example, 316	iframes and, 355
stopPropagation method, 244	maintaining alternate style sheet
stopwatch example, 184	states, 212
strict warnings, 26–29, 544	manipulating, 217
string concatenation, 37, 100, 156	media types and, 226
string data type	types of, 211
array indexing using, 69	styleFloat property, 202
converting arrays to, 71	styles
converting numbers to, 36–38	changing for a group of elements,
converting to numbers, 39–42	203
cookies as, 144	changing for a single element, 201
existence of data, 51	expanding and folder tree menus,
substrings, 51	376
String function, 36	rendering modes and, 199
string manipulation, 45–63	retrieving computed styles, 204
case changes, 47	styleSheets collection, 224
comparing strings, 48, 264	sub-cookies, 149
date formatting, 156, 165	subdomains and cookies, 147
matching with regular expressions,	submenus
57	arrow indicators, 334
searching for and replacing text, 61	constraining within windows, 345
substrings, 51–52, 545	expanded, 374
testing for email addresses, 60	submission and form validation, 116,
testing for numeric data, 58	398, 400
testing for phone numbers, 59	substring method, 52
style object	substrings, 51–52, 545
computed styles and, 205	switch menus (see expanding menus)
left property, 420	switch statements, 116, 540–542
style attribute and, 201, 203	synchronous requests, 471
style property references, 538	System Colors, CSS 2, 403, 410
style sheet switching, 207	
built-in, 211	T
loading delays and, 563	tab order and accessibility, 391, 402
media types and, 215	tab space character, 46, 553, 555
style sheets	tabIndex attribute, 391, 478
(see also CSS)	tabIndex property, 358
adding new rules, 220	table sorting by column, 257–265
creating, 224	target attribute, 133
deleting rules, 223	target elements, 238, 252
drop-down menu example, 323,	ternary operators, 539–540
326, 330, 334	compacting scripts, 551
expanding menu example, 362	1 0 1,

popup overflow example, 130	toLocaleString method, 152
time comparison example, 162, 166	toLowerCase method, 47, 219
test method, 58, 545	tooltips
text boxes and slider controls, 317	accessibility and, 402-411
text fields	displaying on mouseover, 250-257
accessing, 105	Safari browser, 410
auto-completing, 502	toString method, 37, 39, 152
label location, 121	toUpperCase method, 47
locating, 103	transition effects
slider controls and, 311	clip-based transitions, 305
validating mandatory, 113	curtain effect, 309
text manipulation, 61	shrinking effect, 310
(see also string data type)	squashing an object, 306
text nodes	treeMenu function, 364-382
checking for, 87	try/catch structures, 24, 468, 484
creating, using the DOM, 88	24-hour clock, 157
insertion options, 89	type attribute, script tag, 14
removing or relocating, 93	type conversion testing, 28
Safari browser events from, 344	typeof command, 16, 28
text selections, 295, 496-502	alert functions and, 23
text sizes, tooltips, 410	feature detection using, 192
textarea element, 490	isNaN function and, 41
text-only browsers, 436	string-indexed arrays, 70
this variable, 238, 520–521	
Safari bug, 170	U
threshold movement values, 285–286,	undefined data type, 16
293	underscores conversion to spaces, 525
time based data (see Date object; image-	unescape function, 48, 144, 149
based clock)	uniqueID property, document object,
timed effects, 267–270	196
timers	unit normalization, 206
open and close timers, 338	Unobtrusive Flash Objects, 458, 465
timer IDs, 269	unobtrusive scripting, 5, 8–11
timing functions (see setInterval func-	unordered lists
tion; setTimeout function)	auto-complete text example, 508
timing out scripts, 546	drag-and-drop repositioning, 282
title attribute, 207–208, 216	expanding menus from, 362
tooltips and, 250, 252, 402–411	menus using, 322
title element error reports, 25	nesting and wellformedness, 323
toElement property, IE, 417	URL removal with comments, 554
toGMTString method, 152	URL-safe characters, 47

usability	direct evaluation avoiding, 551
accessible DHTML menus, 421–428	dynamic and non-dynamic, 537–538
drag-and-drop effects, 283	nested functions and, 530
frames and, 135	warnings about, 27
inline error messages, 119	VBScript, 463
menu repositioning and, 353	vendor property, navigator object, 196
menu structure and, 345	vertical navigation bars, 325, 412
menu timers and, 338	vertical overflow, 352
online applications, 467	vertically collapsing transitions, 305
open and close timers, 338	viewport size, 141–142
opening current menu branch, 378	(see also page dimensions)
popups and, 128	constraining menus within, 345
progress indicators, 186	drop sheet positioning, 488
screen reader hidden content, 440	tooltip positioning and, 257, 409
user agent strings, 192, 194, 197	visibility property, 368
user interfaces	custom dialog example, 485
creating with DHTML, 229–266	drag-and-drop reordering, 295
differing GUI behaviors, 391	tooltips, 256–257
drag-and-drop functionality, 281–	
290	W
screen readers, 454–455	W3C (World Wide Web Consortium)
UTC (Coordinated Universal Time),	addEventListener method, 16
152, 154	data transmission specifications and,
UTC epoch, 153, 161	468
	device-independent event handlers,
V	393
validating parsers, 13	DOM definition, 80
validation	event listeners, 233
(see also form validation)	event model, 238
email addresses, 60, 115	warnings, 26–29, 544
numeric fields, 114	WCAG (Web Content Accessibility
radio buttons, 115	Guidelines), 393, 395
value property, 105, 111	weighted random selections, 172–173
var keyword, 529–530	whitespace
variable scopes, 528–531	(see also spaces)
closures, 181, 235, 341, 531	code efficiency and, 552
inner scopes, 330	detection, 114
naming conflicts and, 555	necessary whitespace, 554
variables	regular expression check for, 264
(see also global variables)	removal from node trees, 421
compacting names, 555–556	source code obfuscation and, 18

properties, 473 XMLHttpRequest object and, 474 whitespace nodes, DOM, 86 Safari browser support, 197 white-space property, 300 Z wildcard characters, 56 window area (see viewport size) zeroes, 38, 43 window object properties, 128 z-order, 331, 354 windowed controls, 354, 358 windows (see also popups) aggressive scripting, 127 constraining menus within, 345 opening links in new, 133–135 primary, and popup size, 130 Windows Eyes screen reader, 444, 448– 449, 451, 455 browser compatibility, 436 DHTML Accessibility project, 392– 393 Windows IE (see Internet Explorer for Windows) word boundary character, 101 WYSIWYG editor, 489–496, 499 X **XHTML** comments and, 14 forms collection and, 106 navigation list element, 322 XLink, 142 **XML** (see also AJAX) form element access, 106 MIME types, 88 XMLHttpRequest object, 468–476 application development frameworks, 476 feature detection example, 192 headers, 475 iframe alternative, 476-481 methods, 470 notifying users of updates, 446