



Professional
JavaScript
for Web Developers

Nicholas C. Zakas
www.it-ebooks.info

PROFESSIONAL JAVASCRIPT® FOR WEB DEVELOPERS

FOREWORD	xxxi
INTRODUCTION.....	xxxiii
CHAPTER 1 What Is JavaScript?	1
CHAPTER 2 JavaScript in HTML	13
CHAPTER 3 Language Basics	25
CHAPTER 4 Variables, Scope, and Memory	85
CHAPTER 5 Reference Types	103
CHAPTER 6 Object-Oriented Programming	173
CHAPTER 7 Function Expressions	217
CHAPTER 8 The Browser Object Model	239
CHAPTER 9 Client Detection	271
CHAPTER 10 The Document Object Model	309
CHAPTER 11 DOM Extensions.....	357
CHAPTER 12 DOM Levels 2 and 3	381
CHAPTER 13 Events.....	431
CHAPTER 14 Scripting Forms	511
CHAPTER 15 Graphics with Canvas	551
CHAPTER 16 HTML5 Scripting.....	591
CHAPTER 17 Error Handling and Debugging	607
CHAPTER 18 XML in JavaScript.....	641
CHAPTER 19 ECMAScript for XML	671
CHAPTER 20 JSON	691
CHAPTER 21 Ajax and Comet	701
CHAPTER 22 Advanced Techniques.....	731
CHAPTER 23 Offline Applications and Client-Side Storage.....	765
CHAPTER 24 Best Practices.....	801

Continues

CHAPTER 25	Emerging APIs	835
APPENDIX A	ECMAScript Harmony	857
APPENDIX B	Strict Mode	877
APPENDIX C	JavaScript Libraries	885
APPENDIX D	JavaScript Tools	891
INDEX		897

PROFESSIONAL

JavaScript® for Web Developers

PROFESSIONAL

JavaScript® for Web Developers

Third Edition

Nicholas C. Zakas



John Wiley & Sons, Inc.

Professional JavaScript® for Web Developers, Third Edition

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2012 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-118-02669-4
ISBN: 978-1-118-22219-5 (ebk)
ISBN: 978-1-118-23309-2 (ebk)
ISBN: 978-1-118-26080-7 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at <http://booksupport.wiley.com>. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2011943911

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Wrox Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. JavaScript is a registered trademark of Oracle America, Inc. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

To my parents, who never cease to support and inspire me.

ABOUT THE AUTHOR



NICHOLAS C. ZAKAS has been working with the web for over a decade. During that time, he has worked both on corporate intranet applications used by some of the largest companies in the world and on large-scale consumer websites such as My Yahoo! and the Yahoo! homepage. As a presentation architect at Yahoo!, Nicholas guided front-end development and standards for some of the most-visited websites in the world. Nicholas is an established speaker and regularly gives talks at companies, conferences, and meetups regarding front-end best practices and new technology.

He has authored several books, including *Professional Ajax* and *High Performance JavaScript*, and writes regularly on his blog at <http://www.nczonline.net/>. Nicholas's Twitter handle is @slicknet.

ABOUT THE TECHNICAL EDITOR

JOHN PELOQUIN is a front-end engineer with over ten years of JavaScript experience ranging across applications of all sizes. John earned his B.A. in mathematics from the University of California at Berkeley and is currently a lead developer for a health care startup where he makes use of the latest in front-end technologies. Prior to editing this volume, John edited *JavaScript 24-Hour Trainer* by Jeremy McPeak (Wiley, 2010). When he is not coding or collecting errata, John is often found engaged in mathematics, philosophy, or juggling.

CREDITS

EXECUTIVE EDITOR
Carol Long

SENIOR PROJECT EDITOR
Kevin Kent

TECHNICAL EDITOR
John Peloquin

PRODUCTION EDITOR
Kathleen Wisor

COPY EDITOR
Katherine Burt

EDITORIAL MANAGER
Mary Beth Wakefield

FREELANCER EDITORIAL MANAGER
Rosemarie Graham

ASSOCIATE DIRECTOR OF MARKETING
David Mayhew

MARKETING MANAGER
Ashley Zurcher

BUSINESS MANAGER
Amy Knies

PRODUCTION MANAGER
Tim Tate

**VICE PRESIDENT AND EXECUTIVE GROUP
PUBLISHER**
Richard Swadley

**VICE PRESIDENT AND EXECUTIVE
PUBLISHER**
Neil Edde

ASSOCIATE PUBLISHER
Jim Minatel

PROJECT COORDINATOR, COVER
Katie Crocker

PROOFREADER
Nicole Hirschman

INDEXER
Robert Swanson

COVER DESIGNER
LeAndra Young

COVER IMAGE
© iStock/Andrew Rich

ACKNOWLEDGMENTS

EVEN THOUGH THE AUTHOR'S NAME is the one that graces the cover of a book, no book is the result of one person's efforts, and I'd like to thank a few of the people involved in this one.

First and foremost, thanks to John Wiley & Sons for continuing to give me opportunities to write. They were the only people willing to take a risk on an unknown author for the first edition of *Professional JavaScript for Web Developers*, and for that I will be forever grateful.

Thanks to the staff of John Wiley & Sons, specifically Kevin Kent and John Peloquin, who both did an excellent job keeping me honest and dealing with my frequent changes to the book as I was writing.

I'd also like to thank everyone who provided feedback on draft chapters of the book: Rob Friesel, Sergey Ilinsky, Dan Kielp, Peter-Paul Koch, Jeremy McPeak, Alex Petrescu, Dmitry Soshnikov, and Juriy "Kangax" Zaytsev. Your feedback made this book something that I'm extremely proud of.

A special thanks to Brendan Eich for his corrections to the history of JavaScript included in Chapter 1.

Last, but certainly not least, thanks to Rey Bango for writing the foreword of this book. I had the pleasure of meeting Rey for the first time in 2010 after conversing online for several years. He's one of the truly nice guys in the industry, and I'm honored that he agreed to lend his time to this book.

CONTENTS

FOREWORD	<i>xxxii</i>
INTRODUCTION	<i>xxxiii</i>
<hr/>	
CHAPTER 1: WHAT IS JAVASCRIPT?	1
A Short History	2
JavaScript Implementations	3
ECMAScript	3
The Document Object Model (DOM)	6
The Browser Object Model (BOM)	9
JavaScript Versions	10
Summary	11
<hr/>	
CHAPTER 2: JAVASCRIPT IN HTML	13
The <script> Element	13
Tag Placement	16
Deferred Scripts	16
Asynchronous Scripts	17
Changes in XHTML	18
Deprecated Syntax	20
Inline Code versus External Files	20
Document Modes	20
The <noscript> Element	22
Summary	22
<hr/>	
CHAPTER 3: LANGUAGE BASICS	25
Syntax	25
Case-sensitivity	25
Identifiers	26
Comments	26
Strict Mode	27
Statements	27
Keywords and Reserved Words	28
Variables	29
Data Types	31
The typeof Operator	31

The Undefined Type	32
The Null Type	33
The Boolean Type	34
The Number Type	35
The String Type	41
The Object Type	44
Operators	45
Unary Operators	46
Bitwise Operators	49
Boolean Operators	56
Multiplicative Operators	59
Additive Operators	61
Relational Operators	63
Equality Operators	65
Conditional Operator	67
Assignment Operators	67
Comma Operator	68
Statements	69
The if Statement	69
The do-while Statement	70
The while Statement	70
The for Statement	71
The for-in Statement	72
Labeled Statements	73
The break and continue Statements	73
The with Statement	75
The switch Statement	76
Functions	78
Understanding Arguments	80
No Overloading	83
Summary	83
CHAPTER 4: VARIABLES, SCOPE, AND MEMORY	85
Primitive and Reference Values	85
Dynamic Properties	86
Copying Values	86
Argument Passing	88
Determining Type	89
Execution Context and Scope	90
Scope Chain Augmentation	92
No Block-Level Scopes	93

Garbage Collection	96
Mark-and-Sweep	96
Reference Counting	97
Performance	98
Managing Memory	99
Summary	100
CHAPTER 5: REFERENCE TYPES	103
The Object Type	104
The Array Type	106
Detecting Arrays	110
Conversion Methods	110
Stack Methods	112
Queue Methods	113
Reordering Methods	114
Manipulation Methods	116
Location Methods	118
Iterative Methods	119
Reduction Methods	121
The Date Type	122
Inherited Methods	124
Date-Formatting Methods	125
Date/Time Component Methods	126
The RegExp Type	128
RegExp Instance Properties	131
RegExp Instance Methods	132
RegExp Constructor Properties	134
Pattern Limitations	136
The Function Type	136
No Overloading (Revisited)	138
Function Declarations versus Function Expressions	138
Functions as Values	139
Function Internals	141
Function Properties and Methods	143
Primitive Wrapper Types	146
The Boolean Type	148
The Number Type	149
The String Type	151
Singleton Built-in Objects	161
The Global Object	162
The Math Object	166
Summary	170

CHAPTER 6: OBJECT-ORIENTED PROGRAMMING	173
 Understanding Objects	173
Types of Properties	174
Defining Multiple Properties	178
Reading Property Attributes	179
 Object Creation	180
The Factory Pattern	180
The Constructor Pattern	181
The Prototype Pattern	184
Combination Constructor/Prototype Pattern	197
Dynamic Prototype Pattern	198
Parasitic Constructor Pattern	199
Durable Constructor Pattern	200
 Inheritance	201
Prototype Chaining	202
Constructor Stealing	207
Combination Inheritance	209
Prototypal Inheritance	210
Parasitic Inheritance	211
Parasitic Combination Inheritance	212
 Summary	215
CHAPTER 7: FUNCTION EXPRESSIONS	217
 Recursion	220
 Closures	221
Closures and Variables	224
The this Object	225
Memory Leaks	227
 Mimicking Block Scope	228
 Private Variables	231
Static Private Variables	232
The Module Pattern	234
The Module-Augmentation Pattern	236
 Summary	237
CHAPTER 8: THE BROWSER OBJECT MODEL	239
 The window Object	239
The Global Scope	240
Window Relationships and Frames	241
Window Position	244

Window Size	245
Navigating and Opening Windows	247
Intervals and Timeouts	251
System Dialogs	253
The location Object	255
Query String Arguments	256
Manipulating the Location	257
The Navigator Object	259
Detecting Plug-ins	262
Registering Handlers	264
The screen Object	265
The history Object	267
Summary	268
CHAPTER 9: CLIENT DETECTION	271
Capability Detection	271
Safer Capability Detection	273
Capability Detection Is Not Browser Detection	274
Quirks Detection	275
User-Agent Detection	276
History	277
Working with User-Agent Detection	286
The Complete Script	303
Usage	306
Summary	306
CHAPTER 10: THE DOCUMENT OBJECT MODEL	309
Hierarchy of Nodes	310
The Node Type	310
The Document Type	316
The Element Type	326
The Text Type	337
The Comment Type	341
The CDATASection Type	342
The DocumentType Type	343
The DocumentFragment Type	344
The Attr Type	345
Working with the DOM	346
Dynamic Scripts	346
Dynamic Styles	348

Manipulating Tables	350
Using NodeLists	353
Summary	354
CHAPTER 11: DOM EXTENSIONS	357
Selectors API	357
The querySelector() Method	358
The querySelectorAll() Method	358
The matchesSelector() Method	359
Element Traversal	360
HTML5	361
Class-Related Additions	361
Focus Management	364
Changes to HTMLDocument	364
Character Set Properties	366
Custom Data Attributes	366
Markup Insertion	367
The scrollIntoView() Method	372
Proprietary Extensions	372
Document Mode	373
The children Property	374
The contains() Method	374
Markup Insertion	376
Scrolling	379
Summary	379
CHAPTER 12: DOM LEVELS 2 AND 3	381
DOM Changes	382
XML Namespaces	382
Other Changes	386
Styles	390
Accessing Element Styles	391
Working with Style Sheets	396
Element Dimensions	401
Traversals	408
NodeIterator	410
TreeWalker	413
Ranges	415
Ranges in the DOM	415
Ranges in Internet Explorer 8 and Earlier	424
Summary	428

CHAPTER 13: EVENTS	431
Event Flow	432
Event Bubbling	432
Event Capturing	433
DOM Event Flow	433
Event Handlers	434
HTML Event Handlers	434
DOM Level 0 Event Handlers	437
DOM Level 2 Event Handlers	438
Internet Explorer Event Handlers	439
Cross-Browser Event Handlers	441
The Event Object	442
The DOM Event Object	442
The Internet Explorer Event Object	447
Cross-Browser Event Object	449
Event Types	451
UI Events	452
Focus Events	458
Mouse and Wheel Events	459
Keyboard and Text Events	471
Composition Events	478
Mutation Events	479
HTML5 Events	482
Device Events	490
Touch and Gesture Events	494
Memory and Performance	498
Event Delegation	498
Removing Event Handlers	500
Simulating Events	502
DOM Event Simulation	502
Internet Explorer Event Simulation	508
Summary	509
CHAPTER 14: SCRIPTING FORMS	511
Form Basics	511
Submitting Forms	512
Resetting Forms	513
Form Fields	514
Scripting Text Boxes	520
Text Selection	521

Input Filtering	524
Automatic Tab Forward	528
HTML5 Constraint Validation API	530
Scripting Select Boxes	534
Options Selection	536
Adding Options	537
Removing Options	538
Moving and Reordering Options	539
Form Serialization	540
Rich Text Editing	542
Using contenteditable	543
Interacting with Rich Text	543
Rich Text Selections	547
Rich Text in Forms	549
Summary	549
CHAPTER 15: GRAPHICS WITH CANVAS	551
Basic Usage	551
The 2D Context	553
Fills and Strokes	553
Drawing Rectangles	553
Drawing Paths	556
Drawing Text	557
Transformations	559
Drawing Images	563
Shadows	564
Gradients	565
Patterns	567
Working with Image Data	567
Compositing	569
WebGL	571
Typed Arrays	571
The WebGL Context	576
Support	588
Summary	588
CHAPTER 16: HTML5 SCRIPTING	591
Cross-Document Messaging	591
Native Drag and Drop	593
Drag-and-Drop Events	593

Custom Drop Targets	594
The dataTransfer Object	595
DropEffect and effectAllowed	596
Draggability	597
Additional Members	598
Media Elements	598
Properties	599
Events	601
Custom Media Players	602
Codec Support Detection	603
The Audio Type	604
History State Management	605
Summary	606
CHAPTER 17: ERROR HANDLING AND DEBUGGING	607
Browser Error Reporting	607
Internet Explorer	608
Firefox	609
Safari	610
Opera	612
Chrome	613
Error Handling	614
The try-catch Statement	615
Throwing Errors	619
The error Event	622
Error-handling Strategies	623
Identify Where Errors Might Occur	623
Distinguishing between Fatal and Nonfatal Errors	628
Log Errors to the Server	629
Debugging Techniques	630
Logging Messages to a Console	631
Logging Messages to the Page	633
Throwing Errors	634
Common Internet Explorer Errors	635
Operation Aborted	635
Invalid Character	637
Member Not Found	637
Unknown Runtime Error	638
Syntax Error	638
The System Cannot Locate the Resource Specified	639
Summary	639

CHAPTER 18: XML IN JAVASCRIPT	641
XML DOM Support in Browsers	641
DOM Level 2 Core	641
The DOMParser Type	642
The XMLSerializer Type	644
XML in Internet Explorer 8 and Earlier	644
Cross-Browser XML Processing	649
XPath Support in Browsers	651
DOM Level 3 XPath	651
XPath in Internet Explorer	656
Cross-Browser XPath	657
XSLT Support in Browsers	660
XSLT in Internet Explorer	660
The XSLTProcessor Type	665
Cross-Browser XSLT	667
Summary	668
CHAPTER 19: ECMASCIPT FOR XML	671
E4X Types	671
The XML Type	672
The XMLList Type	673
The Namespace Type	674
The QName Type	675
General Usage	676
Accessing Attributes	678
Other Node Types	679
Querying	681
XML Construction and Manipulation	682
Parsing and Serialization Options	685
Namespaces	686
Other Changes	688
Enabling Full E4X	689
Summary	689
CHAPTER 20: JSON	691
Syntax	691
Simple Values	692
Objects	692
Arrays	693
Parsing and Serialization	694

The JSON Object	695
Serialization Options	696
Parsing Options	699
Summary	700
CHAPTER 21: AJAX AND COMET	701
The XMLHttpRequest Object	702
XHR Usage	703
HTTP Headers	706
GET Requests	707
POST Requests	708
XMLHttpRequest Level 2	710
The FormData Type	710
Timeouts	711
The overrideMimeType() Method	711
Progress Events	712
The load Event	712
The progress Event	713
Cross-Origin Resource Sharing	714
CORS in Internet Explorer	714
CORS in Other Browsers	716
Preflighted Requests	717
Credentialled Requests	718
Cross-Browser CORS	718
Alternate Cross-Domain Techniques	719
Image Pings	719
Comet	721
Server-Sent Events	723
Web Sockets	725
SSE versus Web Sockets	727
Security	728
Summary	729
CHAPTER 22: ADVANCED TECHNIQUES	731
Advanced Functions	731
Safe Type Detection	731
Scope-Safe Constructors	733
Lazy Loading Functions	736
Function Binding	738
Function Currying	741

Tamper-Proof Objects	743
Nonextensible Objects	744
Sealed Objects	744
Frozen Objects	745
Advanced Timers	746
Repeating Timers	748
Yielding Processes	750
Function Throttling	752
Custom Events	755
Drag and Drop	758
Fixing Drag Functionality	760
Adding Custom Events	762
Summary	764
CHAPTER 23: OFFLINE APPLICATIONS AND CLIENT-SIDE STORAGE	765
Offline Detection	765
Application Cache	766
Data Storage	768
Cookies	768
Internet Explorer User Data	778
Web Storage	780
IndexedDB	786
Summary	799
CHAPTER 24: BEST PRACTICES	801
Maintainability	801
What Is Maintainable Code?	802
Code Conventions	802
Loose Coupling	805
Programming Practices	809
Performance	814
Be Scope-Aware	814
Choose the Right Approach	816
Minimize Statement Count	821
Optimize DOM Interactions	824
Deployment	827
Build Process	827
Validation	829
Compression	830
Summary	833

CHAPTER 25: EMERGING APIs	835
RequestAnimationFrame()	835
Early Animation Loops	836
Problems with Intervals	836
mozRequestAnimationFrame	837
webkitRequestAnimationFrame and msRequestAnimationFrame	838
Page Visibility API	839
Geolocation API	841
File API	843
The FileReader Type	844
Partial Reads	846
Object URLs	847
Drag-and-Drop File Reading	848
File Upload with XHR	849
Web Timing	851
Web Workers	852
Using a Worker	852
Worker Global Scope	853
Including Other Scripts	855
The Future of Web Workers	855
Summary	856
APPENDIX A: ECMASCIPT HARMONY	857
General Changes	857
Constants	858
Block-Level and Other Scopes	858
Functions	859
Rest and Spread Arguments	859
Default Argument Values	860
Generators	861
Arrays and Other Structures	861
Iterators	862
Array Comprehensions	863
Destructuring Assignments	864
New Object Types	865
Proxy Objects	865
Proxy Functions	868
Map and Set	868
WeakMap	869
StructType	869

ArrayType	870
Classes	871
Private Members	872
Getters/Setters	872
Inheritance	873
Modules	874
External Modules	875
APPENDIX B: STRICT MODE	877
Opting-in	877
Variables	878
Objects	878
Functions	879
eval()	880
eval and arguments	881
Coercion of this	882
Other Changes	882
APPENDIX C: JAVASCRIPT LIBRARIES	885
General Libraries	885
Yahoo! User Interface Library (YUI)	885
Prototype	886
The Dojo Toolkit	886
MooTools	886
jQuery	886
MochiKit	886
Underscore.js	887
Internet Applications	887
Backbone.js	887
Rico	887
qooxdoo	887
Animation and Effects	888
script.aculo.us	888
moo.fx	888
Lightbox	888
Cryptography	888
JavaScript MD5	889
JavaScrypt	889

APPENDIX D: JAVASCRIPT TOOLS	891
Validators	891
JSLint	891
JSHint	892
JavaScript Lint	892
Minifiers	892
JSMIn	892
Dojo ShrinkSafe	892
YUI Compressor	893
Unit Testing	893
JsUnit	893
YUI Test	893
Dojo Object Harness (DOH)	894
qUnit	894
Documentation Generators	894
JsDoc Toolkit	894
YUI Doc	894
AjaxDoc	895
Secure Execution Environments	895
ADsafe	895
Caja	895
INDEX	897

FOREWORD

I look back at my career (now 20+ years), and in between coming to the realization that my gray hairs have really sprouted out, I reflect on the technologies and people that have dramatically affected my professional life and decisions. If I had to choose one technology, though, that has had the single biggest positive influence on me, it would be JavaScript. Mind you, I wasn't always a JavaScript believer. Like many, I looked at it as a play language relegated to doing rotating banners and sprinkling some interesting effects on pages. I was a server-side developer, and we didn't play with toy languages, damn it! But then something happened: Ajax.

I'll never forget hearing the buzzword *Ajax* all over the place and thinking that it was some very cool, new, and innovative technology. I had to check it out, and as I read about it, I was floored when I realized that the toy language I had so readily dismissed was now the technology that was on the lips of every professional web developer. And suddenly, my perception changed. As I continued to explore past what Ajax was, I realized that JavaScript was incredibly powerful, and I wanted in on all the goodness it had to offer. So I embraced it wholeheartedly, working to understand the language, joining the jQuery project team, and focusing on client-side development. Life was good.

The deeper I became involved in JavaScript, the more developers I met, some whom to this day I still see as rock stars and mentors. Nicholas Zakas is one of those developers. I remember reading the second edition of this very book and feeling like, despite all of my years of tinkering, I had learned so much from it. And the book felt genuine and thoughtful, as if Nicholas understood that his audience's experience level would vary and that he needed to manage the tone accordingly. That really stood out in terms of technical books. Most authors try to go into the deep-dive technobabble to impress. This was different, and it immediately became my go-to book and the one I recommended to any developer who wanted to get a solid understanding of JavaScript. I wanted everyone to feel the same way I felt and realize how valuable a resource it is.

And then, at a jQuery conference, I had the amazing fortune of actually meeting Nicholas in person. Here was one of top JavaScript developers in the world working on one of the most important web properties in the world (Yahoo!), and he was one of the nicest people I had ever met. I admit; I was a bit starstruck when I met him. And the great thing was that he was just this incredibly down-to-earth person who just wanted to help developers be great. So not only did his book change the way I thought about JavaScript, but Nicholas himself was someone that I wanted to continue to work with and get to know.

When Nicholas asked me to write this foreword, I can't explain how flattered I was. Here I am being the opening act for the guru. It's a testament to how cool of a person he is. Most important, though, it gives me an opportunity to share with you why I feel this book is so important. I've read many JavaScript books, and there are certainly awesome titles out there. This book, though, offers in my opinion the total package to make you an incredibly proficient and able JavaScript developer.

The smooth and thoughtful transition from introductory topics such as expressions and variable declarations to advanced topics such as closures and object-oriented development is what sets it apart from other books that either are too introductory or expect that you're already building missile guidance systems with JavaScript. It's the “everyman's” book that will help you write code that you'll be proud of and build web site that will excite and delight.

—REY BANGO

*Sr. Technical Evangelist, Microsoft Corporation
jQuery Project Team*

INTRODUCTION

SOME CLAIM THAT JAVASCRIPT is now the most popular programming language in the world, running any number of complex web applications that the world relies on to do business, make purchases, manage processes, and more.

JavaScript is very loosely based on Java, an object-oriented programming language popularized for use on the Web by way of embedded applets. Although JavaScript has a similar syntax and programming methodology, it is not a “light” version of Java. Instead, JavaScript is its own dynamic language, finding its home in web browsers around the world and enabling enhanced user interaction on web sites and web applications alike.

In this book, JavaScript is covered from its very beginning in the earliest Netscape browsers to the present-day incarnations flush with support for the DOM and Ajax. You learn how to extend the language to suit specific needs and how to create seamless client-server communication without intermediaries such as Java or hidden frames. In short, you learn how to apply JavaScript solutions to business problems faced by web developers everywhere.

WHO THIS BOOK IS FOR

This book is aimed at three groups of readers:

- Experienced developers familiar with object-oriented programming who are looking to learn JavaScript as it relates to traditional OO languages such as Java and C++.
- Web application developers attempting to enhance the usability of their web sites and web applications.
- Novice JavaScript developers aiming to better understand the language.

In addition, familiarity with the following related technologies is a strong indicator that this book is for you:

- Java
- PHP
- ASP.NET
- HTML
- CSS
- XML

This book is not aimed at beginners lacking a basic computer science background or those looking to add some simple user interactions to web sites. These readers should instead refer to Wrox’s *Beginning JavaScript, 4th Edition* (Wiley, 2009).

WHAT THIS BOOK COVERS

Professional JavaScript for Web Developers, 3rd Edition, provides a developer-level introduction, along with the more advanced and useful features of JavaScript.

Starting at the beginning, the book explores how JavaScript originated and evolved into what it is today. A detailed discussion of the components that make up a JavaScript implementation follows, with specific focus on standards such as ECMAScript and the Document Object Model (DOM). The differences in JavaScript implementations used in different popular web browsers are also discussed.

Building on that base, the book moves on to cover basic concepts of JavaScript including its version of object-oriented programming, inheritance, and its use in HTML. An in-depth examination of events and event handling is followed by an exploration of browser detection techniques. The book then explores new APIs such as HTML5, the Selectors API, and the File API.

The last part of the book is focused on advanced topics including performance/memory optimization, best practices, and a look at where JavaScript is going in the future.

HOW THIS BOOK IS STRUCTURED

This book comprises the following chapters:

- 1. What Is JavaScript?** — Explains the origins of JavaScript: where it came from, how it evolved, and what it is today. Concepts introduced include the relationship between JavaScript and ECMAScript, the Document Object Model (DOM), and the Browser Object Model (BOM). A discussion of the relevant standards from the European Computer Manufacturer's Association (ECMA) and the World Wide Web Consortium (W3C) is also included.
- 2. JavaScript in HTML** — Examines how JavaScript is used in conjunction with HTML to create dynamic web pages. Introduces the various ways of embedding JavaScript into a page including a discussion surrounding the JavaScript content-type and its relationship to the `<script>` element.
- 3. Language Basics** — Introduces basic language concepts including syntax and flow control statements. Explains the syntactic similarities of JavaScript and other C-based languages and points out the differences. Type coercion is introduced as it relates to built-in operators.
- 4. Variables, Scope, and Memory** — Explores how variables are handled in JavaScript given their loosely typed nature. A discussion about the differences between primitive and reference values is included, as is information about execution context as it relates to variables. Also, a discussion about garbage collection in JavaScript explains how memory is reclaimed when variables go out of scope.
- 5. Reference Types** — Covers all of the details regarding JavaScript's built-in reference types, such as `Object` and `Array`. Each reference type described in ECMA-262 is discussed both in theory and in how they relate to browser implementations.

6. **Object-Oriented Programming** — Explains how to use object-oriented programming in JavaScript. Since JavaScript has no concept of classes, several popular techniques are explored for object creation and inheritance. Also covered in this chapter is the concept of function prototypes and how that relates to an overall OO approach.
7. **Function Expressions** — Explores one of the most powerful aspects of JavaScript: function expressions. Topics include closures, how the `this` object works, the module pattern, and creating private object members.
8. **The Browser Object Model** — Introduces the Browser Object Model (BOM), which is responsible for objects allowing interaction with the browser itself. Each of the BOM objects is covered, including `window`, `document`, `location`, `navigator`, and `screen`.
9. **Client Detection** — Explains various approaches to detecting the client machine and its capabilities. Different techniques include capability detection and user-agent string detection. Each approach is discussed for pros and cons, as well as situational appropriateness.
10. **The Document Object Model** — Introduces the Document Object Model (DOM) objects available in JavaScript as defined in DOM Level 1. A brief introduction to XML and its relationship to the DOM gives way to an in-depth exploration of the entire DOM and how it allows developers to manipulate a page.
11. **DOM Extensions** — Explains how other APIs, as well as the browsers themselves, extend the DOM with more functionality. Topics include the Selectors API, the Element Traversal API, and HTML5 extensions.
12. **DOM Levels 2 and 3** — Builds on the previous two chapters, explaining how DOM Levels 2 and 3 augmented the DOM with additional properties, methods, and objects. Compatibility issues between Internet Explorer and other browsers are discussed.
13. **Events** — Explains the nature of events in JavaScript, where they originated, legacy support, and how the DOM redefined how events should work. A variety of devices are covered including the Wii and iPhone.
14. **Scripting Forms** — Looks at using JavaScript to enhance form interactions and work around browser limitations. Discussion focuses on individual form elements such as text boxes and select boxes and on data validation and manipulation.
15. **Graphics with Canvas** — Discusses the `<canvas>` tag and how to use it to create on-the-fly graphics. Both the 2D context and the WebGL (3D) context are covered, giving you a good starting point for creating animations and games.
16. **HTML5 Scripting** — Introduces JavaScript API changes as defined in HTML5. Topics include cross-document messaging, the Drag-and-Drop API scripting `<audio>` and `<video>` elements, as well as history state management.
17. **Error Handling and Debugging** — Discusses how browsers handle errors in JavaScript code and presents several ways to handle errors. Debugging tools and techniques are also discussed for each browser, including recommendations for simplifying the debugging process.

- 18.** **XML in JavaScript** — Presents the features of JavaScript used to read and manipulate eXtensible Markup Language (XML) data. Explains the differences in support and objects in various web browsers and offers suggestions for easier cross-browser coding. This chapter also covers the use of eXtensible Stylesheet Language Transformations (XSLT) to transform XML data on the client.
- 19.** **ECMAScript for XML** — Discusses the ECMAScript for XML (E4X) extension to JavaScript, which is designed to simplify working with XML. Explains the advantages of E4X over using the DOM for XML manipulation.
- 20.** **JSON** — Introduces the JSON data format as an alternative to XML. Browser-native JSON parsing and serialization are discussed as are security considerations when using JSON.
- 21.** **Ajax and Comet** — Looks at common Ajax techniques including the use of the XMLHttpRequest object and Cross-Origin Resource Sharing (CORS) for cross-domain Ajax. Explains the differences in browser implementations and support and provides recommendations for usage.
- 22.** **Advanced Techniques** — Dives into some of the more complex JavaScript patterns, including function currying, partial function application, and dynamic functions. Also covers creating a custom event framework to enable simple event support for custom objects and creating tamper-proof objects using ECMAScript 5.
- 23.** **Offline Applications and Client-Side Storage** — Discusses how to detect when an application is offline and provides various techniques for storing data on the client machine. Begins with a discussion of the most commonly supported feature, cookies, and then discusses newer functionality such as Web Storage and IndexedDB.
- 24.** **Best Practices** — Explores approaches to working with JavaScript in an enterprise environment. Techniques for better maintainability are discussed, including coding techniques, formatting, and general programming practices. Execution performance is discussed, and several techniques for speed optimization are introduced. Last, deployment issues are discussed, including how to create a build process.
- 25.** **Emerging APIs** — Introduces APIs being created to augment JavaScript in the browser. Even though these APIs aren't yet complete or fully implemented, they are on the horizon, and browsers have already begun partially implementing their features. Includes discussion of Web Timing, geolocation, and the File API.

WHAT YOU NEED TO USE THIS BOOK

To run the samples in the book, you need the following:

- Windows XP, Windows 7, or Mac OS X
- Internet Explorer 6 or higher, Firefox 2 or higher, Opera 9 or higher, Chrome, or Safari 2 or higher

The complete source code for the samples is available for download from the web site at www.wrox.com.

CONVENTIONS

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.



Boxes with a warning icon like this one hold important, not-to-be-forgotten information that is directly relevant to the surrounding text.



The pencil icon indicates notes, tips, hints, tricks, and asides to the current discussion.

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show keyboard strokes like this: Ctrl+A.
- We show file names, URLs, and code within the text like so: `persistence.properties`.
- We present code in two different ways:

We use a monofont type with no highlighting for most code examples.

We use bold to emphasize code that's particularly important in the present context.

SOURCE CODE

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All the source code used in this book is available for download at www.wrox.com. When at the site, simply locate the book's title (use the Search box or one of the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book. Code that is included on the web site is highlighted by the following icon:



Available for
download on
Wrox.com

Listings include the file name in the title. If it is just a code snippet, you'll find the file name in a code note such as this:

Code snippet file name



Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-1-118-02669-4.

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at www.wrox.com/dynamic/books/download.aspx to see the code available for this book and all other Wrox books.

ERRATA

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and at the same time you will be helping us provide even higher quality information.

To find the errata page for this book, go to www.wrox.com and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that have been submitted for this book and posted by Wrox editors. A complete book list including links to each book's errata is also available at www.wrox.com/misc-pages/booklist.shtml.

If you don't spot "your" error on the Book Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

P2P.WROX.COM

For author and peer discussion, join the P2P forums at p2p.wrox.com. The forums are a Web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At p2p.wrox.com you will find a number of different forums that will help you not only as you read this book but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to p2p.wrox.com and click the Register link.
2. Read the terms of use and click Agree.

- 3.** Complete the required information to join, as well as any optional information you wish to provide, and click Submit.
- 4.** You will receive an e-mail with information describing how to verify your account and complete the joining process.



You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works, as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

1

What Is JavaScript?

WHAT'S IN THIS CHAPTER?

- Review of JavaScript history
- What JavaScript is
- How JavaScript and ECMAScript are related
- The different versions of JavaScript

When JavaScript first appeared in 1995, its main purpose was to handle some of the input validation that had previously been left to server-side languages such as Perl. Prior to that time, a round-trip to the server was needed to determine if a required field had been left blank or an entered value was invalid. Netscape Navigator sought to change that with the introduction of JavaScript. The capability to handle some basic validation on the client was an exciting new feature at a time when use of telephone modems was widespread. The associated slow speeds turned every trip to the server into an exercise in patience.

Since that time, JavaScript has grown into an important feature of every major web browser on the market. No longer bound to simple data validation, JavaScript now interacts with nearly all aspects of the browser window and its contents. JavaScript is recognized as a full programming language, capable of complex calculations and interactions, including closures, anonymous (lambda) functions, and even metaprogramming. JavaScript has become such an important part of the Web that even alternative browsers, including those on mobile phones and those designed for users with disabilities, support it. Even Microsoft, with its own client-side scripting language called VBScript, ended up including its own JavaScript implementation in Internet Explorer from its earliest version.

The rise of JavaScript from a simple input validator to a powerful programming language could not have been predicted. JavaScript is at once a very simple and very complicated language that takes minutes to learn but years to master. To begin down the path to using JavaScript's full potential, it is important to understand its nature, history, and limitations.

A SHORT HISTORY

As the Web gained popularity, a gradual demand for client-side scripting languages developed. At the time, most Internet users were connecting over a 28.8 kbps modem even though web pages were growing in size and complexity. Adding to users' pain was the large number of round-trips to the server required for simple form validation. Imagine filling out a form, clicking the Submit button, waiting 30 seconds for processing, and then being met with a message indicating that you forgot to complete a required field. Netscape, at that time on the cutting edge of technological innovation, began seriously considering the development of a client-side scripting language to handle simple processing.

Brendan Eich, who worked for Netscape at the time, began developing a scripting language called Mocha, and later LiveScript, for the release of Netscape Navigator 2 in 1995, with the intention of using it both in the browser and on the server (where it was to be called LiveWire). Netscape entered into a development alliance with Sun Microsystems to complete the implementation of LiveScript in time for release. Just before Netscape Navigator 2 was officially released, Netscape changed LiveScript's name to JavaScript to capitalize on the buzz that Java was receiving from the press.

Because JavaScript 1.0 was such a hit, Netscape released version 1.1 in Netscape Navigator 3. The popularity of the fledgling Web was reaching new heights, and Netscape had positioned itself to be the leading company in the market. At this time, Microsoft decided to put more resources into a competing browser named Internet Explorer. Shortly after Netscape Navigator 3 was released, Microsoft introduced Internet Explorer 3 with a JavaScript implementation called JScript (so called to avoid any possible licensing issues with Netscape). This major step for Microsoft into the realm of web browsers in August 1996 is now a date that lives in infamy for Netscape, but it also represented a major step forward in the development of JavaScript as a language.

Microsoft's implementation of JavaScript meant that there were two different JavaScript versions floating around: JavaScript in Netscape Navigator and JScript in Internet Explorer. Unlike C and many other programming languages, JavaScript had no standards governing its syntax or features, and the three different versions only highlighted this problem. With industry fears mounting, it was decided that the language must be standardized.

In 1997, JavaScript 1.1 was submitted to the European Computer Manufacturers Association (Ecma) as a proposal. Technical Committee #39 (TC39) was assigned to "standardize the syntax and semantics of a general purpose, cross-platform, vendor-neutral scripting language" (www.ecma-international.org/memento/TC39.htm). Made up of programmers from Netscape, Sun, Microsoft, Borland, NOMBAS, and other companies with interest in the future of scripting, TC39 met for months to hammer out ECMA-262, a standard defining a new scripting language named ECMAScript (often pronounced as "ek-ma-script").

The following year, the International Organization for Standardization and International Electrotechnical Commission (ISO/IEC) also adopted ECMAScript as a standard (ISO/IEC-16262). Since that time, browsers have tried, with varying degrees of success, to use ECMAScript as a basis for their JavaScript implementations.

JAVASCRIPT IMPLEMENTATIONS

Though JavaScript and ECMAScript are often used synonymously, JavaScript is much more than just what is defined in ECMA-262. Indeed, a complete JavaScript implementation is made up of the following three distinct parts (see Figure 1-1):

- The Core (ECMAScript)
- The Document Object Model (DOM)
- The Browser Object Model (BOM)

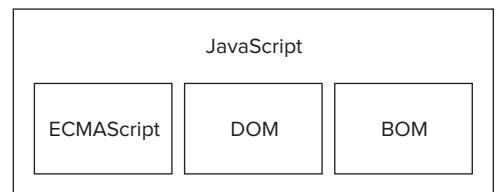


FIGURE 1-1

ECMAScript

ECMAScript, the language defined in ECMA-262, isn't tied to web browsers. In fact, the language has no methods for input or output whatsoever. ECMA-262 defines this language as a base upon which more-robust scripting languages may be built. Web browsers are just one *host environment* in which an ECMAScript implementation may exist. A host environment provides the base implementation of ECMAScript and implementation extensions designed to interface with the environment itself. Extensions, such as the Document Object Model (DOM), use ECMAScript's core types and syntax to provide additional functionality that's more specific to the environment. Other host environments include NodeJS, a server-side JavaScript platform, and Adobe Flash.

What exactly does ECMA-262 specify if it doesn't reference web browsers? On a very basic level, it describes the following parts of the language:

- Syntax
- Types
- Statements
- Keywords
- Reserved words
- Operators
- Objects

ECMAScript is simply a description of a language implementing all of the facets described in the specification. JavaScript implements ECMAScript, but so does Adobe ActionScript.

ECMAScript Editions

The different versions of ECMAScript are defined as *editions* (referring to the edition of ECMA-262 in which that particular implementation is described). The most recent edition of ECMA-262 is edition 5, released in 2009. The first edition of ECMA-262 was essentially the

same as Netscape’s JavaScript 1.1 but with all references to browser-specific code removed and a few minor changes: ECMA-262 required support for the Unicode standard (to support multiple languages) and that objects be platform-independent (Netscape JavaScript 1.1 actually had different implementations of objects, such as the `Date` object, depending on the platform). This was a major reason why JavaScript 1.1 and 1.2 did not conform to the first edition of ECMA-262.

The second edition of ECMA-262 was largely editorial. The standard was updated to get into strict agreement with ISO/IEC-16262 and didn’t feature any additions, changes, or omissions. ECMAScript implementations typically don’t use the second edition as a measure of conformance.

The third edition of ECMA-262 was the first real update to the standard. It provided updates to string handling, the definition of errors, and numeric outputs. It also added support for regular expressions, new control statements, `try-catch` exception handling, and small changes to better prepare the standard for internationalization. To many, this marked the arrival of ECMAScript as a true programming language.

The fourth edition of ECMA-262 was a complete overhaul of the language. In response to the popularity of JavaScript on the Web, developers began revising ECMAScript to meet the growing demands of web development around the world. In response, Ecma TC39 reconvened to decide the future of the language. The resulting specification defined an almost completely new language based on the third edition. The fourth edition includes strongly typed variables, new statements and data structures, true classes and classical inheritance, and new ways to interact with data.

As an alternate proposal, a specification called “ECMAScript 3.1,” was developed as a smaller evolution of the language by a subcommittee of TC39, who believed that the fourth edition was too big of a jump for the language. The result was a smaller proposal with incremental changes to ECMAScript that could be implemented on top of existing JavaScript engines. Ultimately, the ES3.1 subcommittee won over support from TC39, and the fourth edition of ECMA-262 was abandoned before officially being published.

ECMAScript 3.1 became ECMA-262, fifth edition, and was officially published on December 3, 2009. The fifth edition sought to clarify perceived ambiguities of the third edition and introduce additional functionality. The new functionality includes a native `JSON` object for parsing and serializing JSON data, methods for inheritance and advanced property definition, and the inclusion of a new strict mode that slightly augments how ECMAScript engines interpret and execute code.

What Does ECMAScript Conformance Mean?

ECMA-262 lays out the definition of ECMAScript conformance. To be considered an implementation of ECMAScript, an implementation must do the following:

- Support all “types, values, objects, properties, functions, and program syntax and semantics” (ECMA-262, p. 1) as they are described in ECMA-262.
- Support the Unicode character standard.

Additionally, a conforming implementation may do the following:

- Add “additional types, values, objects, properties, and functions” that are not specified in ECMA-262. ECMA-262 describes these additions as primarily new objects or new properties of objects not given in the specification.
- Support “program and regular expression syntax” that is not defined in ECMA-262 (meaning that the built-in regular-expression support is allowed to be altered and extended).

These criteria give implementation developers a great amount of power and flexibility for developing new languages based on ECMAScript, which partly accounts for its popularity.

ECMAScript Support in Web Browsers

Netscape Navigator 3 shipped with JavaScript 1.1 in 1996. That same JavaScript 1.1 specification was then submitted to Ecma as a proposal for the new standard, ECMA-262. With JavaScript’s explosive popularity, Netscape was very happy to start developing version 1.2. There was, however, one problem: Ecma hadn’t yet accepted Netscape’s proposal.

A little after Netscape Navigator 3 was released, Microsoft introduced Internet Explorer 3. This version of IE shipped with JScript 1.0, which was supposed to be equivalent to JavaScript 1.1. However, because of undocumented and improperly replicated features, JScript 1.0 fell far short of JavaScript 1.1.

Netscape Navigator 4 was shipped in 1997 with JavaScript 1.2 before the first edition of ECMA-262 was accepted and standardized later that year. As a result, JavaScript 1.2 is not compliant with the first edition of ECMAScript even though ECMAScript was supposed to be based on JavaScript 1.1.

The next update to JScript occurred in Internet Explorer 4 with JScript version 3.0 (version 2.0 was released in Microsoft Internet Information Server version 3.0 but was never included in a browser). Microsoft put out a press release touting JScript 3.0 as the first truly Ecma-compliant scripting language in the world. At that time, ECMA-262 hadn’t yet been finalized, so JScript 3.0 suffered the same fate as JavaScript 1.2: it did not comply with the final ECMAScript standard.

Netscape opted to update its JavaScript implementation in Netscape Navigator 4.06 to JavaScript 1.3, which brought Netscape into full compliance with the first edition of ECMA-262. Netscape added support for the Unicode standard and made all objects platform-independent while keeping the features that were introduced in JavaScript 1.2.

When Netscape released its source code to the public as the Mozilla project, it was anticipated that JavaScript 1.4 would be shipped with Netscape Navigator 5. However, a radical decision to completely redesign the Netscape code from the bottom up derailed that effort. JavaScript 1.4 was released only as a server-side language for Netscape Enterprise Server and never made it into a web browser.

By 2008, the five major web browsers (Internet Explorer, Firefox, Safari, Chrome, and Opera) all complied with the third edition of ECMA-262. Internet Explorer 8 was the first to start implementing the fifth edition of ECMA-262 specification and delivered complete support in Internet Explorer 9. Firefox 4 soon followed suit. The following table lists ECMAScript support in the most popular web browsers.

BROWSER	ECMASCRIPT COMPLIANCE
Netscape Navigator 2	—
Netscape Navigator 3	—
Netscape Navigator 4–4.05	—
Netscape Navigator 4.06–4.79	Edition 1
Netscape 6+ (Mozilla 0.6.0+)	Edition 3
Internet Explorer 3	—
Internet Explorer 4	—
Internet Explorer 5	Edition 1
Internet Explorer 5.5–7	Edition 3
Internet Explorer 8	Edition 5*
Internet Explorer 9+	Edition 5
Opera 6–7.1	Edition 2
Opera 7.2+	Edition 3
Safari 1–2.0.x	Edition 3*
Safari 3.x	Edition 3
Safari 4.x–5.x	Edition 5*
Chrome 1+	Edition 3
Firefox 1–2	Edition 3
Firefox 3.0.x	Edition 3
Firefox 3.5–3.6	Edition 5*
Firefox 4+	Edition 5

*Incomplete implementations

The Document Object Model (DOM)

The *Document Object Model* (DOM) is an application programming interface (API) for XML that was extended for use in HTML. The DOM maps out an entire page as a hierarchy of nodes. Each part of an HTML or XML page is a type of a node containing different kinds of data. Consider the following HTML page:

```
<html>
  <head>
    <title>Sample Page</title>
  </head>
  <body>
    <p>Hello World!</p>
  </body>
</html>
```

This code can be diagrammed into a hierarchy of nodes using the DOM (see Figure 1-2).

By creating a tree to represent a document, the DOM allows developers an unprecedented level of control over its content and structure. Nodes can be removed, added, replaced, and modified easily by using the DOM API.

Why the DOM Is Necessary

With Internet Explorer 4 and Netscape Navigator 4 each supporting different forms of Dynamic HTML (DHTML), developers for the first time could alter the appearance and content of a web page without reloading it. This represented a tremendous step forward in web technology but also a huge problem. Netscape and Microsoft went separate ways in developing DHTML, thus ending the period when developers could write a single HTML page that could be accessed by any web browser.

It was decided that something had to be done to preserve the cross-platform nature of the Web. The fear was that if someone didn't rein in Netscape and Microsoft, the Web would develop into two distinct factions that were exclusive to targeted browsers. It was then that the World Wide Web Consortium (W3C), the body charged with creating standards for web communication, began working on the DOM.

DOM Levels

DOM Level 1 became a W3C recommendation in October 1998. It consisted of two modules: the DOM Core, which provided a way to map the structure of an XML-based document to allow for easy access to and manipulation of any part of a document, and the DOM HTML, which extended the DOM Core by adding HTML-specific objects and methods.



Note that the DOM is not JavaScript-specific and indeed has been implemented in numerous other languages. For web browsers, however, the DOM has been implemented using ECMAScript and now makes up a large part of the JavaScript language.

Whereas the goal of DOM Level 1 was to map out the structure of a document, the aims of DOM Level 2 were much broader. This extension of the original DOM added support for mouse and user-interface events (long supported by DHTML), ranges, traversals (methods to iterate over a DOM document), and support for Cascading Style Sheets (CSS) through object interfaces. The original DOM Core introduced in Level 1 was also extended to include support for XML namespaces.

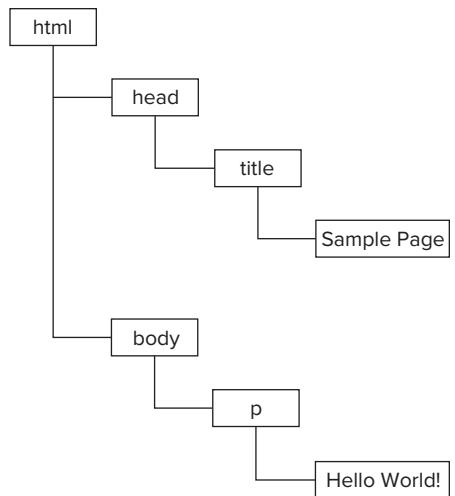


FIGURE 1-2

DOM Level 2 introduced the following new modules of the DOM to deal with new types of interfaces:

- **DOM Views** — Describes interfaces to keep track of the various views of a document (the document before and after CSS styling, for example)
- **DOM Events** — Describes interfaces for events and event handling
- **DOM Style** — Describes interfaces to deal with CSS-based styling of elements
- **DOM Traversal and Range** — Describes interfaces to traverse and manipulate a document tree

DOM Level 3 further extends the DOM with the introduction of methods to load and save documents in a uniform way (contained in a new module called DOM Load and Save) and methods to validate a document (DOM Validation). In Level 3, the DOM Core is extended to support all of XML 1.0, including XML Infoset, XPath, and XML Base.



When reading about the DOM, you may come across references to DOM Level 0. Note that there is no standard called DOM Level 0; it is simply a reference point in the history of the DOM. DOM Level 0 is considered to be the original DHTML supported in Internet Explorer 4.0 and Netscape Navigator 4.0.

Other DOMs

Aside from the DOM Core and DOM HTML interfaces, several other languages have had their own DOM standards published. The languages in the following list are XML-based, and each DOM adds methods and interfaces unique to a particular language:

- Scalable Vector Graphics (SVG) 1.0
- Mathematical Markup Language (MathML) 1.0
- Synchronized Multimedia Integration Language (SMIL)

Additionally, other languages have developed their own DOM implementations, such as Mozilla's XML User Interface Language (XUL). However, only the languages in the preceding list are standard recommendations from W3C.

DOM Support in Web Browsers

The DOM had been a standard for some time before web browsers started implementing it. Internet Explorer made its first attempt with version 5, but it didn't have any realistic DOM support until version 5.5, when it implemented most of DOM Level 1. Internet Explorer didn't introduce new DOM functionality in versions 6 and 7, though version 8 introduced some bug fixes.

For Netscape, no DOM support existed until Netscape 6 (Mozilla 0.6.0) was introduced. After Netscape 7, Mozilla switched its development efforts to the Firefox browser. Firefox 3+ supports all of Level 1, nearly all of Level 2, and some parts of Level 3. (The goal of the Mozilla development team was to build a 100 percent standards-compliant browser, and their work paid off.)

DOM support became a huge priority for most browser vendors, and efforts have been ongoing to improve support with each release. The following table shows DOM support for popular browsers.

BROWSER	DOM COMPLIANCE
Netscape Navigator 1.–4.x	—
Netscape 6+ (Mozilla 0.6.0+)	Level 1, Level 2 (almost all), Level 3 (partial)
Internet Explorer 2–4.x	—
Internet Explorer 5	Level 1 (minimal)
Internet Explorer 5.5–8	Level 1 (almost all)
Internet Explorer 9+	Level 1, Level 2, Level 3
Opera 1–6	—
Opera 7–8.x	Level 1 (almost all), Level 2 (partial)
Opera 9–9.9	Level 1, Level 2 (almost all), Level 3 (partial)
Opera 10+	Level 1, Level 2, Level 3 (partial)
Safari 1.0.x	Level 1
Safari 2+	Level 1, Level 2 (partial)
Chrome 1+	Level 1, Level 2 (partial)
Firefox 1+	Level 1, Level 2 (almost all), Level 3 (partial)

The Browser Object Model (BOM)

The Internet Explorer 3 and Netscape Navigator 3 browsers featured a *Browser Object Model* (BOM) that allowed access and manipulation of the browser window. Using the BOM, developers can interact with the browser outside of the context of its displayed page. What made the BOM truly unique, and often problematic, was that it was the only part of a JavaScript implementation that had no related standard. This changed with the introduction of HTML5, which sought to codify much of the BOM as part of a formal specification. Thanks to HTML5, a lot of the confusion surrounding the BOM has dissipated.

Primarily, the BOM deals with the browser window and frames, but generally any browser-specific extension to JavaScript is considered to be a part of the BOM. The following are some such extensions:

- The capability to pop up new browser windows
- The capability to move, resize, and close browser windows
- The `navigator` object, which provides detailed information about the browser

- The `location` object, which gives detailed information about the page loaded in the browser
- The `screen` object, which gives detailed information about the user's screen resolution
- Support for cookies
- Custom objects such as `XMLHttpRequest` and Internet Explorer's `ActiveXObject`

Because no standards existed for the BOM for a long time, each browser has its own implementation. There are some de facto standards, such as having a `window` object and a `navigator` object, but each browser defines its own properties and methods for these and other objects. With HTML5 now available, the implementation details of the BOM are expected to grow in a much more compatible way. A detailed discussion of the BOM is included in Chapter 8.

JAVASCRIPT VERSIONS

Mozilla, as a descendant from the original Netscape, is the only browser vendor that has continued the original JavaScript version-numbering sequence. When the Netscape source code was spun off into an open-source project (named the Mozilla Project), the last browser version of JavaScript was 1.3. (As mentioned previously, version 1.4 was implemented on the server exclusively.) As the Mozilla Foundation continued work on JavaScript, adding new features, keywords, and syntaxes, the JavaScript version number was incremented. The following table shows the JavaScript version progression in Netscape/Mozilla browsers.

BROWSER	JAVASCRIPT VERSION
Netscape Navigator 2	1.0
Netscape Navigator 3	1.1
Netscape Navigator 4	1.2
Netscape Navigator 4.06	1.3
Netscape 6+ (Mozilla 0.6.0+)	1.5
Firefox 1	1.5
Firefox 1.5	1.6
Firefox 2	1.7
Firefox 3	1.8
Firefox 3.5	1.8.1
Firefox 3.6	1.8.2

The numbering scheme was based on the idea that Firefox 4 would feature JavaScript 2.0, and each increment in the version number prior to that point indicates how close the JavaScript implementation is to the 2.0 proposal. Though this was the original plan, the evolution of JavaScript happened in such a way that this was no longer possible. There is currently no target implementation for JavaScript 2.0.



It's important to note that only the Netscape/Mozilla browsers follow this versioning scheme. Internet Explorer, for example, has different version numbers for JScript. These JScript versions don't correspond whatsoever to the JavaScript versions mentioned in the preceding table. Furthermore, most browsers talk about JavaScript support in relation to their level of ECMAScript compliance and DOM support.

SUMMARY

JavaScript is a scripting language designed to interact with web pages and is made up of the following three distinct parts:

- ECMAScript, which is defined in ECMA-262 and provides the core functionality
- The Document Object Model (DOM), which provides methods and interfaces for working with the content of a web page
- The Browser Object Model (BOM), which provides methods and interfaces for interacting with the browser

There are varying levels of support for the three parts of JavaScript across the five major web browsers (Internet Explorer, Firefox, Chrome, Safari, and Opera). Support for ECMAScript 3 is generally good across all browsers, and support for ECMAScript 5 is growing, whereas support for the DOM varies widely. The BOM, recently codified in HTML5, can vary from browser to browser, though there are some commonalities that are assumed to be available.

2

JavaScript in HTML

WHAT'S IN THIS CHAPTER?

- Using the `<script>` element
- Comparing inline and external scripts
- Examining how document modes affect JavaScript
- Preparing for JavaScript-disabled experiences

The introduction of JavaScript into web pages immediately ran into the Web's predominant language, HTML. As part of its original work on JavaScript, Netscape tried to figure out how to make JavaScript coexist in HTML pages without breaking those pages' rendering in other browsers. Through trial, error, and controversy, several decisions were finally made and agreed upon to bring universal scripting support to the Web. Much of the work done in these early days of the Web has survived and become formalized in the HTML specification.

THE `<SCRIPT>` ELEMENT

The primary method of inserting JavaScript into an HTML page is via the `<script>` element. This element was created by Netscape and first implemented in Netscape Navigator 2. It was later added to the formal HTML specification. There are six attributes for the `<script>` element:

- `async` — Optional. Indicates that the script should begin downloading immediately but should not prevent other actions on the page such as downloading resources or waiting for other scripts to load. Valid only for external script files.
- `charset` — Optional. The character set of the code specified using the `src` attribute. This attribute is rarely used, because most browsers don't honor its value.
- `defer` — Optional. Indicates that the execution of the script can safely be deferred until after the document's content has been completely parsed and displayed. Valid only for external scripts. Internet Explorer 7 and earlier also allow for inline scripts.

- `language` — Deprecated. Originally indicated the scripting language being used by the code block (such as "JavaScript", "JavaScript1.2", or "VBScript"). Most browsers ignore this attribute; it should not be used.
- `src` — Optional. Indicates an external file that contains code to be executed.
- `type` — Optional. Replaces `language`; indicates the content type (also called MIME type) of the scripting language being used by the code block. Traditionally, this value has always been "text/javascript", though both "text/javascript" and "text/ecmascript" are deprecated. JavaScript files are typically served with the "application/x-javascript" MIME type even though setting this in the `type` attribute may cause the script to be ignored. Other values that work in non-Internet Explorer browsers are "application/javascript" and "application/ecmascript". The `type` attribute is still typically set to "text/javascript" by convention and for maximum browser compatibility. This attribute is safe to omit, as "text/javascript" is assumed when missing.

There are two ways to use the `<script>` element: embed JavaScript code directly into the page or include JavaScript from an external file.

To include inline JavaScript code, place JavaScript code inside the `<script>` element directly, as follows:

```
<script type="text/javascript">
    function sayHi(){
        alert("Hi!");
    }
</script>
```

The JavaScript code contained inside a `<script>` element is interpreted from top to bottom. In the case of this example, a function definition is interpreted and stored inside the interpreter environment. The rest of the page content is not loaded and/or displayed until after all of the code inside the `<script>` element has been evaluated.

When using inline JavaScript code, keep in mind that you cannot have the string "`</script>`" anywhere in your code. For example, the following code causes an error when loaded into a browser:

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

Because of the way that inline scripts are parsed, the browser sees the string "`</script>`" as if it were the closing `</script>` tag. This problem can be avoided easily by escaping the "/" character, as in this example:

```
<script type="text/javascript">
    function sayScript(){
        alert("<\&#x2f;&gt;/script>");
    }
</script>
```

The changes to this code make it acceptable to browsers and won't cause any errors.

To include JavaScript from an external file, the `src` attribute is required. The value of `src` is a URL linked to a file containing JavaScript code, like this:

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

In this example, an external file named `example.js` is loaded into the page. The file itself need only contain the JavaScript code that would occur between the opening `<script>` and closing `</script>` tags. As with inline JavaScript code, processing of the page is halted while the external file is interpreted. (There is also some time taken to download the file.) In XHTML documents, you can omit the closing tag, as in this example:

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

This syntax should not be used in HTML documents, because it is invalid HTML and won't be handled properly by some browsers, most notably Internet Explorer.



By convention, external JavaScript files have a `.js` extension. This is not a requirement, because browsers do not check the file extension of included JavaScript files. This leaves open the possibility of dynamically generating JavaScript code using JSP, PHP, or another server-side scripting language. Keep in mind, though, that servers often use the file extension to determine the correct MIME type to apply to the response. If you don't use a `.js` extension, double-check that your server is returning the correct MIME type.

It's important to note that a `<script>` element using the `src` attribute should not include additional JavaScript code between the `<script>` and `</script>` tags. If both are provided, the script file is downloaded and executed while the inline code is ignored.

One of the most powerful and most controversial parts of the `<script>` element is its ability to include JavaScript files from outside domains. Much like an `` element, the `<script>` element's `src` attribute may be set to a full URL that exists outside the domain on which the HTML page exists, as in this example:

```
<script type="text/javascript" src="http://www.somewhere.com/afile.js"></script>
```

Code from an external domain will be loaded and interpreted as if it were part of the page that is loading it. This capability allows you to serve up JavaScript from various domains if necessary. Be careful, however, if you are referencing JavaScript files located on a server that you don't control. A malicious programmer could, at any time, replace the file. When including JavaScript files from a different domain, make sure you are the domain owner or the domain is owned by a trusted source.

Regardless of how the code is included, the `<script>` elements are interpreted in the order in which they appear in the page so long as the `defer` and `async` attributes are not present. The first

`<script>` element's code must be completely interpreted before the second `<script>` element begins interpretation, the second must be completed before the third, and so on.

Tag Placement

Traditionally, all `<script>` elements were placed within the `<head>` element on a page, such as in this example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
    <script type="text/javascript" src="example1.js"></script>
    <script type="text/javascript" src="example2.js"></script>
  </head>
  <body>
    <!-- content here -->
  </body>
</html>
```

The main purpose of this format was to keep external file references, both CSS files and JavaScript files, in the same area. However, including all JavaScript files in the `<head>` of a document means that all of the JavaScript code must be downloaded, parsed, and interpreted before the page begins rendering (rendering begins when the browser receives the opening `<body>` tag). For pages that require a lot of JavaScript code, this can cause a noticeable delay in page rendering, during which time the browser will be completely blank. For this reason, modern web applications typically include all JavaScript references in the `<body>` element, after the page content, as shown in this example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
  </head>
  <body>
    <!-- content here -->
    <script type="text/javascript" src="example1.js"></script>
    <script type="text/javascript" src="example2.js"></script>
  </body>
</html>
```

Using this approach, the page is completely rendered in the browser before the JavaScript code is processed. The resulting user experience is perceived as faster, because the amount of time spent on a blank browser window is reduced.

Deferred Scripts

HTML 4.01 defines an attribute named `defer` for the `<script>` element. The purpose of `defer` is to indicate that a script won't be changing the structure of the page as it executes. As such, the script can be run safely after the entire page has been parsed. Setting the `defer` attribute on a `<script>`

element signals to the browser that download should begin immediately but execution should be deferred:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
    <script type="text/javascript" defer src="example1.js"></script>
    <script type="text/javascript" defer src="example2.js"></script>
  </head>
  <body>
    <!-- content here -->
  </body>
</html>
```

Even though the `<script>` elements in this example are included in the document `<head>`, they will not be executed until after the browser has received the closing `</html>` tag. The HTML5 specification indicates that scripts will be executed in the order in which they appear, so the first deferred script executes before the second deferred script, and both will execute before the `DOMContentLoaded` event (see Chapter 13 for more information). In reality, though, deferred scripts don't always execute in order or before the `DOMContentLoaded` event, so it's best to include just one when possible.

As mentioned previously, the `defer` attribute is supported only for external script files. This was a clarification made in HTML5, so browsers that support the HTML5 implementation will ignore `defer` when set on an inline script. Internet Explorer 4–7 all exhibit the old behavior, while Internet Explorer 8 and above support the HTML5 behavior.

Support for the `defer` attribute was added beginning with Internet Explorer 4, Firefox 3.5, Safari 5, and Chrome 7. All other browsers simply ignore this attribute and treat the script as it normally would. For this reason, it's still best to put deferred scripts at the bottom of the page.



For XHTML documents, specify the `defer` attribute as `defer="defer"`.

Asynchronous Scripts

HTML5 introduces the `async` attribute for `<script>` elements. The `async` attribute is similar to `defer` in that it changes the way the script is processed. Also similar to `defer`, `async` applies only to external scripts and signals the browser to begin downloading the file immediately. Unlike `defer`, scripts marked as `async` are not guaranteed to execute in the order in which they are specified. For example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
    <script type="text/javascript" async src="example1.js"></script>
    <script type="text/javascript" async src="example2.js"></script>
```

```
</head>
<body>
    <!-- content here -->
</body>
</html>
```

In this code, the second script file might execute before the first, so it's important that there are no dependencies between the two. The purpose of specifying an `async` script is to indicate that the page need not wait for the script to be downloaded and executed before continuing to load, and it also need not wait for another script to load and execute before it can do the same. Because of this, it's recommended that asynchronous scripts not modify the DOM as they are loading.

Asynchronous scripts are guaranteed to execute before the page's `load` event and may execute before or after `DOMContentLoaded` (see Chapter 13 for details). Firefox 3.6, Safari 5, and Chrome 7 support asynchronous scripts.



For XHTML documents, specify the `async` attribute as `async="async"`.

Changes in XHTML

Extensible HyperText Markup Language, or XHTML, is a reformulation of HTML as an application of XML. The rules for writing code in XHTML are stricter than those for HTML, which affects the `<script>` element when using embedded JavaScript code. Although valid in HTML, the following code block is invalid in XHTML:

```
<script type="text/javascript">
    function compare(a, b) {
        if (a < b) {
            alert("A is less than B");
        } else if (a > b) {
            alert("A is greater than B");
        } else {
            alert("A is equal to B");
        }
    }
</script>
```

In HTML, the `<script>` element has special rules governing how its contents should be parsed; in XHTML, these special rules don't apply. This means that the less-than symbol (`<`) in the statement `a < b` is interpreted as the beginning of a tag, which causes a syntax error because a less-than symbol must not be followed by a space.

There are two options for fixing the XHTML syntax error. The first is to replace all occurrences of the less-than symbol (`<`) with its HTML entity (`<`). The resulting code looks like this:

```
<script type="text/javascript">
    function compare(a, b) {
        if (a < b) {
            alert("A is less than B");
        } else if (a > b) {
            alert("A is greater than B");
        } else {
            alert("A is equal to B");
        }
    }
</script>
```

This code will now run in an XHTML page; however, the code is slightly less readable. Fortunately, there is another approach.

The second option for turning this code into a valid XHTML version is to wrap the JavaScript code in a CDATA section. In XHTML (and XML), CDATA sections are used to indicate areas of the document that contain free-form text not intended to be parsed. This enables you to use any character, including the less-than symbol, without incurring a syntax error. The format is as follows:

```
<script type="text/javascript"><![CDATA[
    function compare(a, b) {
        if (a < b) {
            alert("A is less than B");
        } else if (a > b) {
            alert("A is greater than B");
        } else {
            alert("A is equal to B");
        }
    }
]]></script>
```

In XHTML-compliant web browsers, this solves the problem. However, many browsers are still not XHTML-compliant and don't support the CDATA section. To work around this, the CDATA markup must be offset by JavaScript comments:

```
<script type="text/javascript">
//<![CDATA[
    function compare(a, b) {
        if (a < b) {
            alert("A is less than B");
        } else if (a > b) {
            alert("A is greater than B");
        } else {
            alert("A is equal to B");
        }
    }
//]]>
</script>
```

This format works in all modern browsers. Though a little bit of a hack, it validates as XHTML and degrades gracefully for pre-XHTML browsers.



XHTML mode is triggered when a page specifies its MIME type as "application/xhtml+xml". Not all browsers officially support XHTML served in this manner.

Deprecated Syntax

When the `<script>` element was originally introduced, it marked a departure from traditional HTML parsing. Special rules needed to be applied within this element, and that caused problems for browsers that didn't support JavaScript (the most notable being Mosaic). Nonsupporting browsers would output the contents of the `<script>` element onto the page, effectively ruining the page's appearance.

Netscape worked with Mosaic to come up with a solution that would hide embedded JavaScript code from browsers that didn't support it. The final solution was to enclose the script code in an HTML comment, like this:

```
<script><!--
    function sayHi(){
        alert("Hi!");
    }
//--></script>
```

Using this format, browsers like Mosaic would safely ignore the content inside of the `<script>` tag, and browsers that supported JavaScript had to look for this pattern to recognize that there was indeed JavaScript content to be parsed.

Although this format is still recognized and interpreted correctly by all web browsers, it is no longer necessary and should not be used. In XHTML mode, this also causes the script to be ignored because it is inside a valid XML comment.

INLINE CODE VERSUS EXTERNAL FILES

Although it's possible to embed JavaScript in HTML files directly, it's generally considered a best practice to include as much JavaScript as possible using external files. Keeping in mind that there are no hard and fast rules regarding this practice, the arguments for using external files are as follows:

- **Maintainability** — JavaScript code that is sprinkled throughout various HTML pages turns code maintenance into a problem. It is much easier to have a directory for all JavaScript files so that developers can edit JavaScript code independent of the markup in which it's used.
- **Caching** — Browsers cache all externally linked JavaScript files according to specific settings, meaning that if two pages are using the same file, the file is downloaded only once. This ultimately means faster page-load times.
- **Future-proof** — By including JavaScript using external files, there's no need to use the XHTML or comment hacks mentioned previously. The syntax to include external files is the same for both HTML and XHTML.

DOCUMENT MODES

Internet Explorer 5.5 introduced the concept of document modes through the use of doctype switching. The first two document modes were *quirks mode*, which made Internet Explorer behave as if it were version 5 (with several nonstandard features), and *standards mode*, which made Internet Explorer behave in a more standards-compliant way. Though the primary difference between these two modes is related to the rendering of content with regard to CSS, there are also several side effects related to JavaScript. These side effects are discussed throughout the book.

Since Internet Explorer first introduced the concept of document modes, other browsers have followed suit. As this adoption happened, a third mode called *almost standards mode* arose. That mode has a lot of the features of standards mode but isn't as strict. The main difference is in the treatment of spacing around images (most noticeable when images are used in tables).

Quirks mode is achieved in all browsers by omitting the doctype at the beginning of the document. This is considered poor practice, because quirks mode is very different across all browsers, and no level of true browser consistency can be achieved without hacks.

Standards mode is turned on when one of the following doctypes is used:

```
<!-- HTML 4.01 Strict -->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

<!-- XHTML 1.0 Strict -->
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- HTML5 -->
<!DOCTYPE html>
```

Almost standards mode is triggered by transitional and frameset doctypes, as follows:

```
<!-- HTML 4.01 Transitional -->
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<!-- HTML 4.01 Frameset -->
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">

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

<!-- XHTML 1.0 Frameset -->
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

Because almost standards mode is so close to standards mode, the distinction is rarely made. People talking about “standards mode” may be talking about either, and detection for the document mode (discussed later in this book) also doesn’t make the distinction. Throughout this book, the term *standards mode* should be taken to mean any mode other than quirks.

THE <NOSCRIPT> ELEMENT

Of particular concern to early browsers was the graceful degradation of pages when the browser didn’t support JavaScript. To that end, the `<noscript>` element was created to provide alternate content for browsers without JavaScript. This element can contain any HTML elements, aside from `<script>`, that can be included in the document `<body>`. Any content contained in a `<noscript>` element will be displayed under only the following two circumstances:

- The browser doesn’t support scripting.
- The browser’s scripting support is turned off.

If either of these conditions is met, then the content inside the `<noscript>` element is rendered. In all other cases, the browser does not render the content of `<noscript>`.

Here is a simple example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
    <script type="text/javascript" defer="defer" src="example1.js"></script>
    <script type="text/javascript" defer="defer" src="example2.js"></script>
  </head>
  <body>
    <noscript>
      <p>This page requires a JavaScript-enabled browser.</p>
    </noscript>
  </body>
</html>
```

In this example, a message is displayed to the user when the scripting is not available. For scripting-enabled browsers, this message will never be seen even though it is still a part of the page.

SUMMARY

JavaScript is inserted into HTML pages by using the `<script>` element. This element can be used to embed JavaScript into an HTML page, leaving it inline with the rest of the markup, or to include JavaScript that exists in an external file. The following are key points:

- To include external JavaScript files, the `src` attribute must be set to the URL of the file to include, which may be a file on the same server as the containing page or one that exists on a completely different domain.

- All `<script>` elements are interpreted in the order in which they occur on the page. The code contained within a `<script>` element must be completely interpreted before code in the next `<script>` element can begin so long as `defer` and `async` attributes are not used.
- For nondeferred scripts, the browser must complete interpretation of the code inside a `<script>` element before it can continue rendering the rest of the page. For this reason, `<script>` elements are usually included toward the end of the page, after the main content and just before the closing `</body>` tag.
- You can defer a script's execution until after the document has rendered by using the `defer` attribute. Deferred scripts always execute in the order in which they are specified.
- You can indicate that a script need not wait for other scripts and also not block the document rendering by using the `async` attribute. Asynchronous scripts are not guaranteed to execute in the order in which they occur in the page.

By using the `<noscript>` element, you can specify that content is to be shown only if scripting support isn't available on the browser. Any content contained in the `<noscript>` element will not be rendered if scripting is enabled on the browser.

3

Language Basics

WHAT'S IN THIS CHAPTER?

- Reviewing syntax
- Working with data types
- Working with flow-control statements
- Understanding functions

At the core of any language is a description of how it should work at the most basic level. This description typically defines syntax, operators, data types, and built-in functionality upon which complex solutions can be built. As previously mentioned, ECMA-262 defines all of this information for JavaScript in the form of a pseudolanguage called ECMAScript.

ECMAScript as defined in ECMA-262, third edition, is the most-implemented version among web browsers. The fifth edition is the next to be implemented in browsers, though, as of the end of 2011, no browser has fully implemented it. For this reason the following information is based primarily on ECMAScript as defined in the third edition with changes in the fifth edition called out.

SYNTAX

ECMAScript's syntax borrows heavily from C and other C-like languages such as Java and Perl. Developers familiar with such languages should have an easy time picking up the somewhat looser syntax of ECMAScript.

Case-sensitivity

The first concept to understand is that everything is case-sensitive; variables, function names, and operators are all case-sensitive, meaning that a variable named `test` is different from

a variable named `Test`. Similarly, `typeof` can't be the name of a function, because it's a keyword (described in the next section); however, `typeof` is a perfectly valid function name.

Identifiers

An *identifier* is the name of a variable, function, property, or function argument. Identifiers may be one or more characters in the following format:

- The first character must be a letter, an underscore (_), or a dollar sign (\$).
- All other characters may be letters, underscores, dollar signs, or numbers.

Letters in an identifier may include extended ASCII or Unicode letter characters such as Å and Æ, though this is not recommended.

By convention, ECMAScript identifiers use camel case, meaning that the first letter is lowercase and each additional word is offset by a capital letter, like this:

```
firstSecond  
myCar  
doSomethingImportant
```

Although this is not strictly enforced, it is considered a best practice to adhere to the built-in ECMAScript functions and objects that follow this format.



Keywords, reserved words, `true`, `false`, and `null` cannot be used as identifiers. See the section “Keywords and Reserved Words” coming up shortly for more detail.

Comments

ECMAScript uses C-style comments for both single-line and block comments. A single-line comment begins with two forward-slash characters, such as this:

```
//single line comment
```

A block comment begins with a forward slash and asterisk /*) and ends with the opposite (* /), as in this example:

```
/*  
 * This is a multi-line  
 * Comment  
 */
```

Note that even though the second and third lines contain an asterisk, these are not necessary and are added purely for readability. (This is the format preferred in enterprise applications.)

Strict Mode

ECMAScript 5 introduced the concept of *strict mode*. Strict mode is a different parsing and execution model for JavaScript, where some of the erratic behavior of ECMAScript 3 is addressed and errors are thrown for unsafe activities. To enable strict mode for an entire script, include the following at the top:

```
"use strict";
```

Although this may look like a string that isn't assigned to a variable, this is a pragma that tells supporting JavaScript engines to change into strict mode. The syntax was chosen specifically so as not to break ECMAScript 3 syntax.

You may also specify just a function to execute in strict mode by including the pragma at the top of the function body:

```
function doSomething(){
    "use strict";
    //function body
}
```

Strict mode changes many parts of how JavaScript is executed, and as such, strict mode distinctions are pointed out throughout the book. Internet Explorer 10+, Firefox 4+, Safari 5.1+, Opera 12+, and Chrome support strict mode.

Statements

Statements in ECMAScript are terminated by a semicolon, though omitting the semicolon makes the parser determine where the end of a statement occurs, as in the following examples:

```
var sum = a + b          //valid even without a semicolon - not recommended
var diff = a - b;        //valid - preferred
```

Even though a semicolon is not required at the end of statements, it is recommended to always include one. Including semicolons helps prevent errors of omission, such as not finishing what you were typing, and allows developers to compress ECMAScript code by removing extra white space (such compression causes syntax errors when lines do not end in a semicolon). Including semicolons also improves performance in certain situations, because parsers try to correct syntax errors by inserting semicolons where they appear to belong.

Multiple statements can be combined into a code block by using C-style syntax, beginning with a left curly brace ({) and ending with a right curly brace (}):

```
if (test){
    test = false;
    alert(test);
}
```

Control statements, such as `if`, require code blocks only when executing multiple statements. However, it is considered a best practice to always use code blocks with control statements, even if there's only one statement to be executed, as in the following examples:

```
if (test)
    alert(test);      //valid, but error-prone and should be avoided

if (test){
    alert(test);
}
```

Using code blocks for control statements makes the intent clearer, and there's less chance for errors when changes need to be made.

KEYWORDS AND RESERVED WORDS

ECMA-262 describes a set of *keywords* that have specific uses, such as indicating the beginning or end of control statements or performing specific operations. By rule, keywords are reserved and cannot be used as identifiers or property names. The complete list of keywords is as follows (those denoted with an asterisk were added in the fifth edition):

break	do	instanceof	typeof
case	else	new	var
catch	finally	return	void
continue	for	switch	while
debugger*	function	this	with
default	if	throw	
delete	in	try	

The specification also describes a set of *reserved words* that cannot be used as identifiers or property names. Though reserved words don't have any specific usage in the language, they are reserved for future use as keywords. The following is the complete list of reserved words defined in ECMA-262, third edition:

abstract	enum	int	short
boolean	export	interface	static
byte	extends	long	super
char	final	native	synchronized
class	float	package	throws
const	goto	private	transient
debugger	implements	protected	volatile
double	import	public	

The fifth edition shrinks down the list of reserved words when running in nonstrict mode to the following:

class	enum	extends	super
const	export	import	

When running in strict mode, the fifth edition also places reserved word restrictions on the following:

implements	package	public
interface	private	static
let	protected	yield

Note that `let` and `yield` are introduced as reserved words in the fifth edition; all other reserved words come from the third edition. For best compatibility, it's recommended to use the third edition list as a guideline and add `let` and `yield`.

Attempting to use a keyword as an identifier name will cause an “Identifier Expected” error in ECMAScript 3 JavaScript engines. Attempting to use a reserved word may or may not cause the same error, depending on the particular engine being used.

The fifth edition slightly changes the rules regarding keywords and reserved words. These may still not be used as identifiers but now can be used as property names in objects. Generally speaking, it's best to avoid using both keywords and reserved words as both identifiers and property names to ensure compatibility with past and future ECMAScript editions.

In addition to the list of keywords and reserved words, ECMA-262, fifth edition, places restrictions on the names `eval` and `arguments`. When running in strict mode, these two names may not be used as identifiers or property names and will throw errors when an attempt is made to do so.

VARIABLES

ECMAScript variables are loosely typed, meaning that a variable can hold any type of data. Every variable is simply a named placeholder for a value. To define a variable, use the `var` operator (note that `var` is a keyword) followed by the variable name (an identifier, as described earlier), like this:

```
var message;
```

This code defines a variable named `message` that can be used to hold any value. (Without initialization, it holds the special value `undefined`, which is discussed in the next section.) ECMAScript implements variable initialization, so it's possible to define the variable and set its value at the same time, as in this example:

```
var message = "hi";
```

Here, `message` is defined to hold a string value of “`hi`”. Doing this initialization doesn't mark the variable as being a string type; it is simply the assignment of a value to the variable. It is still possible to not only change the value stored in the variable but also change the type of value, such as this:

```
var message = "hi";
message = 100;      //legal, but not recommended
```

In this example, the variable `message` is first defined as having the string value "hi" and then overwritten with the numeric value 100. Though it's not recommended to switch the data type that a variable contains, it is completely valid in ECMAScript.

It's important to note that using the `var` operator to define a variable makes it local to the scope in which it was defined. For example, defining a variable inside of a function using `var` means that the variable is destroyed as soon as the function exits, as shown here:

```
function test() {
    var message = "hi"; //local variable
}
test();
alert(message); //error!
```

Here, the `message` variable is defined within a function using `var`. The function is called `test()`, which creates the variable and assigns its value. Immediately after that, the variable is destroyed so the last line in this example causes an error. It is, however, possible to define a variable globally by simply omitting the `var` operator as follows:

```
function test() {
    message = "hi"; //global variable
}
test();
alert(message); //"hi"
```

By removing the `var` operator from the example, the `message` variable becomes global. As soon as the function `test()` is called, the variable is defined and becomes accessible outside of the function once it has been executed.



Although it's possible to define global variables by omitting the `var` operator, this approach is not recommended. Global variables defined locally are hard to maintain and cause confusion, because it's not immediately apparent if the omission of `var` was intentional. Strict mode throws a `ReferenceError` when an undeclared variable is assigned a value.

If you need to define more than one variable, you can do it using a single statement, separating each variable (and optional initialization) with a comma like this:

```
var message = "hi",
    found = false,
    age = 29;
```

Here, three variables are defined and initialized. Because ECMAScript is loosely typed, variable initializations using different data types may be combined into a single statement. Though inserting line breaks and indenting the variables isn't necessary, it helps to improve readability.

When you are running in strict mode, you cannot define variables named `eval` or `arguments`. Doing so results in a syntax error.

DATA TYPES

There are five simple data types (also called *primitive types*) in ECMAScript: Undefined, Null, Boolean, Number, and String. There is also one complex data type called Object, which is an unordered list of name-value pairs. Because there is no way to define your own data types in ECMAScript, all values can be represented as one of these six. Having only six data types may seem like too few to fully represent data; however, ECMAScript's data types have dynamic aspects that make each single data type behave like several.

The `typeof` Operator

Because ECMAScript is loosely typed, there needs to be a way to determine the data type of a given variable. The `typeof` operator provides that information. Using the `typeof` operator on a value returns one of the following strings:

- "undefined" if the value is undefined
- "boolean" if the value is a Boolean
- "string" if the value is a string
- "number" if the value is a number
- "object" if the value is an object (other than a function) or null
- "function" if the value is a function

The `typeof` operator is called like this:



Available for
download on
Wrox.com

```
var message = "some string";
alert(typeof message);      //"string"
alert(typeof(message));    //"string"
alert(typeof 95);          //"number"
```

[TypeofExample01.htm](#)

In this example, both a variable (message) and a numeric literal are passed into the `typeof` operator. Note that because `typeof` is an operator and not a function, no parentheses are required (although they can be used).

Be aware there are a few cases where `typeof` seemingly returns a confusing but technically correct value. Calling `typeof null` returns a value of "object", as the special value `null` is considered to be an empty object reference. Safari through version 5 and Chrome through version 7 have a quirk where calling `typeof` on a regular expression returns "function" while all other browsers return "object".



Technically, functions are considered objects in ECMAScript and don't represent another data type. However, they do have some special properties, which necessitates differentiating between functions and other objects via the `typeof` operator.

The Undefined Type

The `Undefined` type has only one value, which is the special value `undefined`. When a variable is declared using `var` but not initialized, it is assigned the value of `undefined` as follows:



```
var message;
alert(message == undefined);      //true
```

[UndefinedExample01.htm](#)

In this example, the variable `message` is declared without initializing it. When compared with the literal value of `undefined`, the two are equal. This example is identical to the following:

```
var message = undefined;
alert(message == undefined);      //true
```

[UndefinedExample02.htm](#)

Here the variable `message` is explicitly initialized to be `undefined`. This is unnecessary because, by default, any uninitialized variable gets the value of `undefined`.



Generally speaking, you should never explicitly set a variable to be `undefined`. The literal `undefined` value is provided mainly for comparison and wasn't added until ECMA-262, third edition to help formalize the difference between an empty object pointer (`null`) and an uninitialized variable.

Note that a variable containing the value of `undefined` is different from a variable that hasn't been defined at all. Consider the following:

```
var message;      //this variable is declared but has a value of undefined
                  //make sure this variable isn't declared
                  //var age

alert(message);  // "undefined"
alert(age);      //causes an error
```

[UndefinedExample03.htm](#)

In this example, the first alert displays the variable `message`, which is "undefined". In the second alert, an undeclared variable called `age` is passed into the `alert()` function, which causes an error because the variable hasn't been declared. Only one useful operation can be performed on an undeclared variable: you can call `typeof` on it (calling `delete` on an undeclared variable won't cause an error, but this isn't very useful and in fact throws an error in strict mode).

The `typeof` operator returns "undefined" when called on an uninitialized variable, but it also returns "undefined" when called on an undeclared variable, which can be a bit confusing. Consider this example:



```
var message;      //this variable is declared but has a value of undefined
                  //make sure this variable isn't declared
                  //var age

alert(typeof message);  //"undefined"
alert(typeof age);      //"undefined"
```

[UndefinedExample04.htm](#)

In both cases, calling `typeof` on the variable returns the string "undefined". Logically, this makes sense because no real operations can be performed with either variable even though they are technically very different.



Even though uninitialized variables are automatically assigned a value of `undefined`, it is advisable to always initialize variables. That way, when `typeof` returns "undefined", you'll know that it's because a given variable hasn't been declared rather than was simply not initialized.

The Null Type

The Null type is the second data type that has only one value: the special value `null`. Logically, a `null` value is an empty object pointer, which is why `typeof` returns "object" when it's passed a `null` value in the following example:

```
var car = null;
alert(typeof car);    //"object"
```

[NullExample01.htm](#)

When defining a variable that is meant to later hold an object, it is advisable to initialize the variable to `null` as opposed to anything else. That way, you can explicitly check for the value `null` to determine if the variable has been filled with an object reference at a later time, such as in this example:

```
if (car != null){
    //do something with car
}
```

The value `undefined` is a derivative of `null`, so ECMA-262 defines them to be superficially equal as follows:

```
alert(null == undefined);    //true
```

[NullExample02.htm](#)

Using the equality operator (`==`) between `null` and `undefined` always returns `true`, though keep in mind that this operator converts its operands for comparison purposes (covered in detail later in this chapter).

Even though `null` and `undefined` are related, they have very different uses. As mentioned previously, you should never explicitly set the value of a variable to `undefined`, but the same does not hold true for `null`. Any time an object is expected but is not available, `null` should be used in its place. This helps to keep the paradigm of `null` as an empty object pointer and further differentiates it from `undefined`.

The Boolean Type

The Boolean type is one of the most frequently used types in ECMAScript and has only two literal values: `true` and `false`. These values are distinct from numeric values, so `true` is not equal to 1, and `false` is not equal to 0. Assignment of Boolean values to variables is as follows:

```
var found = true;
var lost = false;
```

Note that the Boolean literals `true` and `false` are case-sensitive, so `True` and `False` (and other mixings of uppercase and lowercase) are valid as identifiers but not as Boolean values.

Though there are just two literal Boolean values, all types of values have Boolean equivalents in ECMAScript. To convert a value into its Boolean equivalent, the special `Boolean()` casting function is called, like this:



Available for
download on
Wrox.com

```
var message = "Hello world!";
var messageAsBoolean = Boolean(message);
```

BooleanExample01.htm

In this example, the string `message` is converted into a Boolean value and stored in `messageAsBoolean`. The `Boolean()` casting function can be called on any type of data and will always return a Boolean value. The rules for when a value is converted to `true` or `false` depend on the data type as much as the actual value. The following table outlines the various data types and their specific conversions.

DATA TYPE	VALUES CONVERTED TO TRUE	VALUES CONVERTED TO FALSE
Boolean	<code>true</code>	<code>false</code>
String	Any nonempty string	" " (empty string)
Number	Any nonzero number (including infinity)	0, <code>Nan</code> (See the “ <code>Nan</code> ” section later in this chapter.)
Object	Any object	<code>null</code>
Undefined	n/a	<code>undefined</code>

These conversions are important to understand because flow-control statements, such as the `if` statement, automatically perform this Boolean conversion, as shown here:



```
var message = "Hello world!";
if (message) {
    alert("Value is true");
}
```

[BooleanExample02.htm](#)

In this example, the alert will be displayed because the string `message` is automatically converted into its Boolean equivalent (`true`). It's important to understand what variable you're using in a flow-control statement because of this automatic conversion. Mistakenly using an object instead of a Boolean can drastically alter the flow of your application.

The Number Type

Perhaps the most interesting data type in ECMAScript is `Number`, which uses the IEEE-754 format to represent both integers and floating-point values (also called double-precision values in some languages). To support the various types of numbers, there are several different number literal formats.

The most basic number literal format is that of a decimal integer, which can be entered directly as shown here:

```
var intNum = 55;           //integer
```

Integers can also be represented as either octal (base 8) or hexadecimal (base 16) literals. For an octal literal, the first digit must be a zero (0) followed by a sequence of octal digits (numbers 0 through 7). If a number out of this range is detected in the literal, then the leading zero is ignored and the number is treated as a decimal, as in the following examples:

```
var octalNum1 = 070;      //octal for 56
var octalNum2 = 079;      //invalid octal - interpreted as 79
var octalNum3 = 08;       //invalid octal - interpreted as 8
```

Octal literals are invalid when running in strict mode and will cause the JavaScript engine to throw a syntax error.

To create a hexadecimal literal, you must make the first two characters `0x` (case insensitive), followed by any number of hexadecimal digits (0 through 9, and A through F). Letters may be in uppercase or lowercase. Here's an example:

```
var hexNum1 = 0xA;        //hexadecimal for 10
var hexNum2 = 0x1f;       //hexadecimal for 31
```

Numbers created using octal or hexadecimal format are treated as decimal numbers in all arithmetic operations.



Because of the way that numbers are stored in JavaScript, it is actually possible to have a value of positive zero (+0) and negative zero (-0). Positive zero and negative zero are considered equivalent in all cases but are noted in this text for clarity.

Floating-Point Values

To define a floating-point value, you must include a decimal point and at least one number after the decimal point. Although an integer is not necessary before a decimal point, it is recommended. Here are some examples:

```
var floatNum1 = 1.1;
var floatNum2 = 0.1;
var floatNum3 = .1;      //valid, but not recommended
```

Because storing floating-point values uses twice as much memory as storing integer values, ECMAScript always looks for ways to convert values into integers. When there is no digit after the decimal point, the number becomes an integer. Likewise, if the number being represented is a whole number (such as 1.0), it will be converted into an integer, as in this example:

```
var floatNum1 = 1.;      //missing digit after decimal - interpreted as integer 1
var floatNum2 = 10.0;    //whole number - interpreted as integer 10
```

For very large or very small numbers, floating-point values can be represented using *e-notation*. E-notation is used to indicate a number that should be multiplied by 10 raised to a given power. The format of e-notation in ECMAScript is to have a number (integer or floating-point) followed by an uppercase or lowercase letter E, followed by the power of 10 to multiply by. Consider the following:

```
var floatNum = 3.125e7;    //equal to 31250000
```

In this example, `floatNum` is equal to 31,250,000 even though it is represented in a more compact form using e-notation. The notation essentially says, “Take 3.125 and multiply it by 10^7 .”

E-notation can also be used to represent very small numbers, such as 0.000000000000003, which can be written more succinctly as 3e-17. By default, ECMAScript converts any floating-point value with at least six zeros after the decimal point into e-notation (for example, 0.0000003 becomes 3e-7).

Floating-point values are accurate up to 17 decimal places but are far less accurate in arithmetic computations than whole numbers. For instance, adding 0.1 and 0.2 yields 0.3000000000000004 instead of 0.3. These small rounding errors make it difficult to test for specific floating-point values. Consider this example:

```
if (a + b == 0.3){           //avoid!
  alert("You got 0.3.");
}
```

Here the sum of two numbers is tested to see if it's equal to 0.3. This will work for 0.05 and 0.25 and for 0.15 and 0.15. But if applied to 0.1 and 0.2, as discussed previously, this test would fail. Therefore you should never test for specific floating-point values.



It's important to understand that rounding errors are a side effect of the way floating-point arithmetic is done in IEEE-754-based numbers and is not unique to ECMAScript. Other languages that use the same format have the same issues.

Range of Values

Not all numbers in the world can be represented in ECMAScript, because of memory constraints. The smallest number that can be represented in ECMAScript is stored in `Number.MIN_VALUE` and is `5e-324` on most browsers; the largest number is stored in `Number.MAX_VALUE` and is `1.7976931348623157e+308` on most browsers. If a calculation results in a number that cannot be represented by JavaScript's numeric range, the number automatically gets the special value of `Infinity`. Any negative number that can't be represented is `-Infinity` (negative infinity), and any positive number that can't be represented is simply `Infinity` (positive infinity).

If a calculation returns either positive or negative `Infinity`, that value cannot be used in any further calculations, because `Infinity` has no numeric representation with which to calculate. To determine if a value is finite (that is, it occurs between the minimum and the maximum), there is the `isFinite()` function. This function returns `true` only if the argument is between the minimum and the maximum values, as in this example:

```
var result = Number.MAX_VALUE + Number.MAX_VALUE;
alert(isFinite(result)); //false
```

Though it is rare to do calculations that take values outside of the range of finite numbers, it is possible and should be monitored when doing very large or very small calculations.



You can also get the values of positive and negative `Infinity` by accessing `Number.NEGATIVE_INFINITY` and `Number.POSITIVE_INFINITY`. As you may expect, these properties contain the values `-Infinity` and `Infinity`, respectively.

NaN

There is a special numeric value called `NaN`, short for *Not a Number*, which is used to indicate when an operation intended to return a number has failed (as opposed to throwing an error). For example, dividing any number by 0 typically causes an error in other programming languages, halting code execution. In ECMAScript, dividing a number by 0 returns `NaN`, which allows other processing to continue.

The value `NaN` has a couple of unique properties. First, any operation involving `NaN` always returns `NaN` (for instance, `NaN / 10`), which can be problematic in the case of multistep computations. Second, `NaN` is not equal to any value, including `NaN`. For example, the following returns `false`:

```
alert(NaN == NaN);      //false
```

For this reason, ECMAScript provides the `isNaN()` function. This function accepts a single argument, which can be of any data type, to determine if the value is “not a number.” When a value is passed into `isNaN()`, an attempt is made to convert it into a number. Some nonnumber values convert into numbers directly, such as the string `"10"` or a Boolean value. Any value that cannot be converted into a number causes the function to return `true`. Consider the following:



Available for download on
Wrox.com

```
alert(isNaN(NaN));      //true
alert(isNaN(10));       //false - 10 is a number
alert(isNaN("10"));     //false - can be converted to number 10
alert(isNaN("blue"));   //true - cannot be converted to a number
alert(isNaN(true));     //false - can be converted to number 1
```

[NumberExample03.htm](#)

This example tests five different values. The first test is on the value `NaN` itself, which, obviously, returns `true`. The next two tests use numeric `10` and the string `"10"`, which both return `false`, because the numeric value for each is `10`. The string `"blue"`, however, cannot be converted into a number, so the function returns `false`. The Boolean value of `true` can be converted into the number `1`, so the function returns `false`.



Although typically not done, `isNaN()` can be applied to objects. In that case, the object's `valueOf()` method is first called to determine if the returned value can be converted into a number. If not, the `toString()` method is called and its returned value is tested as well. This is the general way that built-in functions and operators work in ECMAScript and is discussed more in the “Operators” section later in this chapter.

Number Conversions

There are three functions to convert nonnumeric values into numbers: the `Number()` casting function, the `parseInt()` function, and the `parseFloat()` function. The first function, `Number()`, can be used on any data type; the other two functions are used specifically for converting strings to numbers. Each of these functions reacts differently to the same input.

The `Number()` function performs conversions based on these rules:

- When applied to Boolean values, `true` and `false` get converted into `1` and `0`, respectively.
- When applied to numbers, the value is simply passed through and returned.

- When applied to `null`, `Number()` returns 0.
- When applied to `undefined`, `Number()` returns `NaN`.
- When applied to strings, the following rules are applied:
 - If the string contains only numbers, optionally preceded by a plus or minus sign, it is always converted to a decimal number, so "`1`" becomes 1, "`123`" becomes 123, and "`011`" becomes 11 (note: leading zeros are ignored).
 - If the string contains a valid floating-point format, such as "`1.1`", it is converted into the appropriate floating-point numeric value (once again, leading zeros are ignored).
 - If the string contains a valid hexadecimal format, such as "`0xf`", it is converted into an integer that matches the hexadecimal value.
 - If the string is empty (contains no characters), it is converted to 0.
 - If the string contains anything other than these previous formats, it is converted into `NaN`.
- When applied to objects, the `valueOf()` method is called and the returned value is converted based on the previously described rules. If that conversion results in `NaN`, the `toString()` method is called and the rules for converting strings are applied.

Converting to numbers from various data types can get complicated, as indicated by the number of rules there are for `Number()`. Here are some concrete examples:



Available for
download on
[Wrox.com](#)

```
var num1 = Number("Hello world!"); //NaN
var num2 = Number(""); //0
var num3 = Number("000011"); //11
var num4 = Number(true); //1
```

[NumberExample04.htm](#)

In these examples, the string "`Hello world`" is converted into `NaN` because it has no corresponding numeric value, and the empty string is converted into 0. The string "`000011`" is converted to the number 11 because the initial zeros are ignored. Last, the value `true` is converted to 1.



The unary plus operator, discussed in the “Operators” section later in this chapter, works the same as the `Number()` function.

Because of the complexities and oddities of the `Number()` function when converting strings, the `parseInt()` function is usually a better option when you are dealing with integers. The `parseInt()` function examines the string much more closely to see if it matches a number pattern. Leading white space in the string is ignored until the first non-white space character is found. If this first character isn't a number, the minus sign, or the plus sign, `parseInt()` always returns `NaN`, which means the empty string returns `NaN` (unlike with `Number()`, which returns 0). If the first character is a number, plus, or minus, then the conversion goes on to the second character and continues on until either

the end of the string is reached or a nonnumeric character is found. For instance, "1234blue" is converted to 1234 because "blue" is completely ignored. Similarly, "22.5" will be converted to 22 because the decimal is not a valid integer character.

Assuming that the first character in the string is a number, the `parseInt()` function also recognizes the various integer formats (decimal, octal, and hexadecimal, as discussed previously). This means when the string begins with "0x", it is interpreted as a hexadecimal integer; if it begins with "0" followed by a number, it is interpreted as an octal value.

Here are some conversion examples to better illustrate what happens:



```
var num1 = parseInt("1234blue");      //1234
var num2 = parseInt("");            //NaN
var num3 = parseInt("0xA");        //10 - hexadecimal
var num4 = parseInt(22.5);        //22
var num5 = parseInt("70");          //70 - decimal
var num6 = parseInt("0xf");        //15 - hexadecimal
```

NumberExample05.htm

There is a discrepancy between ECMAScript 3 and 5 in regard to using `parseInt()` with a string that looks like an octal literal. For example:

```
//56 (octal) in ECMAScript 3, 0 (decimal) in ECMAScript 5
var num = parseInt("070");
```

In ECMAScript 3 JavaScript engines, the value "070" is treated as an octal literal and becomes the decimal value 56. In ECMAScript 5 JavaScript engines, the ability to parse octal values has been removed from `parseInt()`, so the leading zero is considered invalid and the value is treated the same as "0", resulting in the decimal value 0. This is true even when running ECMAScript 5 in non-strict mode.

All of the different numeric formats can be confusing to keep track of, so `parseInt()` provides a second argument: the radix (number of digits) to use. If you know that the value you're parsing is in hexadecimal format, you can pass in the radix 16 as a second argument and ensure that the correct parsing will occur, as shown here:

```
var num = parseInt("0xAF", 16);      //175
```

In fact, by providing the hexadecimal radix, you can leave off the leading "0x" and the conversion will work as follows:

```
var num1 = parseInt("AF", 16);      //175
var num2 = parseInt("AF");          //NaN
```

NumberExample06.htm

In this example, the first conversion occurs correctly, but the second conversion fails. The difference is that the radix is passed in on the first line, telling `parseInt()` that it will be passed a hexadecimal string; the second line sees that the first character is not a number and stops automatically.

Passing in a radix can greatly change the outcome of the conversion. Consider the following:



```
var num1 = parseInt("10", 2);           //2 - parsed as binary
var num2 = parseInt("10", 8);           //8 - parsed as octal
var num3 = parseInt("10", 10);          //10 - parsed as decimal
var num4 = parseInt("10", 16);          //16 - parsed as hexadecimal
```

[NumberExample07.htm](#)

Because leaving off the radix allows `parseInt()` to choose how to interpret the input, it's advisable to always include a radix to avoid errors.



Most of the time you'll be parsing decimal numbers, so it's good to always include 10 as the second argument.

The `parseFloat()` function works in a similar way to `parseInt()`, looking at each character starting in position 0. It also continues to parse the string until it reaches either the end of the string or a character that is invalid in a floating-point number. This means that a decimal point is valid the first time it appears, but a second decimal point is invalid and the rest of the string is ignored, resulting in "22.34.5" being converted to 22.34.

Another difference in `parseFloat()` is that initial zeros are always ignored. This function will recognize any of the floating-point formats discussed earlier, as well as the decimal format (leading zeros are always ignored). Hexadecimal numbers always become 0. Because `parseFloat()` parses only decimal values, there is no radix mode. A final note: if the string represents a whole number (no decimal point or only a zero after the decimal point), `parseFloat()` returns an integer. Here are some examples:

```
var num1 = parseFloat("1234blue");      //1234 - integer
var num2 = parseFloat("0xA");            //0
var num3 = parseFloat("22.5");           //22.5
var num4 = parseFloat("22.34.5");        //22.34
var num5 = parseFloat("0908.5");         //908.5
var num6 = parseFloat("3.125e7");        //31250000
```

[NumberExample08.htm](#)

The String Type

The String data type represents a sequence of zero or more 16-bit Unicode characters. Strings can be delineated by either double quotes ("") or single quotes (''), so both of the following are legal:

```
var firstName = "Nicholas";
var lastName = 'Zakas';
```

Unlike PHP, for which using double or single quotes changes how the string is interpreted, there is no difference in the two syntaxes in ECMAScript. A string using double quotes is exactly the same as a string using single quotes. Note, however, that a string beginning with a double quote must end with a double quote, and a string beginning with a single quote must end with a single quote. For example, the following will cause a syntax error:

```
var firstName = 'Nicholas";      //syntax error - quotes must match
```

Character Literals

The String data type includes several character literals to represent nonprintable or otherwise useful characters, as listed in the following table:

LITERAL	MEANING
\n	New line
\t	Tab
\b	Backspace
\r	Carriage return
\f	Form feed
\\	Backslash (\)
\'	Single quote ('') — used when the string is delineated by single quotes. Example: 'He said, \'hey.\' '.
\"	Double quote ("") — used when the string is delineated by double quotes. Example: "He said, \"hey.\\"".
\xnn	A character represented by hexadecimal code nn (where n is a hexadecimal digit 0-F). Example: \x41 is equivalent to "A".
\unnnn	A Unicode character represented by the hexadecimal code nnnn (where n is a hexadecimal digit 0-F). Example: \u03a3 is equivalent to the Greek character Σ.

These character literals can be included anywhere with a string and will be interpreted as if they were a single character, as shown here:

```
var text = "This is the letter sigma: \u03a3.";
```

In this example, the variable `text` is 28 characters long even though the escape sequence is 6 characters long. The entire escape sequence represents a single character, so it is counted as such.

The length of any string can be returned by using the `length` property as follows:

```
alert(text.length); //outputs 28
```

This property returns the number of 16-bit characters in the string. If a string contains double-byte characters, the `length` property may not accurately return the number of characters in the string.

The Nature of Strings

Strings are immutable in ECMAScript, meaning that once they are created, their values cannot change. To change the string held by a variable, the original string must be destroyed and the variable filled with another string containing a new value, like this:

```
var lang = "Java";
lang = lang + "Script";
```

Here, the variable `lang` is defined to contain the string `"Java"`. On the next line, `lang` is redefined to combine `"Java"` with `"Script"`, making its value `"JavaScript"`. This happens by creating a new string with enough space for 10 characters and then filling that string with `"Java"` and `"Script"`. The last step in the process is to destroy the original string `"Java"` and the string `"Script"`, because neither is necessary anymore. All of this happens behind the scenes, which is why older browsers (such as pre-1.0 versions of Firefox and Internet Explorer 6.0) had very slow string concatenation. These inefficiencies were addressed in later versions of these browsers.

Converting to a String

There are two ways to convert a value into a string. The first is to use the `toString()` method that almost every value has. (The nature of this method is discussed in Chapter 5.) This method's only job is to return the string equivalent of the value. Consider this example:



Available for
download on
[Wrox.com](#)

```
var age = 11;
var ageAsString = age.toString();      //the string "11"
var found = true;
var foundAsString = found.toString(); //the string "true"
```

[StringExample01.htm](#)

The `toString()` method is available on values that are numbers, Booleans, objects, and strings. (Yes, each string has a `toString()` method that simply returns a copy of itself.) If a value is `null` or `undefined`, this method is not available.

In most cases, `toString()` doesn't have any arguments. However, when used on a number value, `toString()` actually accepts a single argument: the radix in which to output the number. By default, `toString()` always returns a string that represents the number as a decimal, but by passing in a radix, `toString()` can output the value in binary, octal, hexadecimal, or any other valid base, as in this example:

```
var num = 10;
alert(num.toString());           //"10"
alert(num.toString(2));         //"1010"
alert(num.toString(8));         //"12"
alert(num.toString(10));        //"10"
alert(num.toString(16));        //"a"
```

[StringExample02.htm](#)

This example shows how the output of `toString()` can change for numbers when providing a radix. The value 10 can be output into any number of numeric formats. Note that the default (with no argument) is the same as providing a radix of 10.

If you're not sure that a value isn't `null` or `undefined`, you can use the `String()` casting function, which always returns a string regardless of the value type. The `String()` function follows these rules:

- If the value has a `toString()` method, it is called (with no arguments) and the result is returned.
- If the value is `null`, "`null`" is returned.
- If the value is `undefined`, "`undefined`" is returned.

Consider the following:



Available for download on Wrox.com

```
var value1 = 10;
var value2 = true;
var value3 = null;
var value4;

alert(String(value1));      // "10"
alert(String(value2));      // "true"
alert(String(value3));      // "null"
alert(String(value4));      // "undefined"
```

[StringExample03.htm](#)

Here, four values are converted into strings: a number, a Boolean, `null`, and `undefined`. The result for the number and the Boolean are the same as if `toString()` were called. Because `toString()` isn't available on "`null`" and "`undefined`", the `String()` method simply returns literal text for those values.



You can also convert a value to a string by adding an empty string ("") to that value using the plus operator (discussed in the "Operators" section later in this chapter).

The Object Type

Objects in ECMAScript start out as nonspecific groups of data and functionality. Objects are created by using the `new` operator followed by the name of the object type to create. Developers create their own objects by creating instances of the `Object` type and adding properties and/or methods to it, as shown here:

```
var o = new Object();
```

This syntax is similar to Java, although ECMAScript requires parentheses to be used only when providing arguments to the constructor. If there are no arguments, as in the following example, then the parentheses can be omitted safely (though that's not recommended):

```
var o = new Object; //legal, but not recommended
```

Instances of `Object` aren't very useful on their own, but the concepts are important to understand, because, similar to `java.lang.Object` in Java, the `Object` type in ECMAScript is the base from which all other objects are derived. All of the properties and methods of the `Object` type are also present on other, more specific objects.

Each `Object` instance has the following properties and methods:

- `constructor` — The function that was used to create the object. In the previous example, the constructor is the `Object()` function.
- `hasOwnProperty(propertyName)` — Indicates if the given property exists on the object instance (not on the prototype). The property name must be specified as a string (for example, `o.hasOwnProperty("name")`).
- `isPrototypeOf(object)` — Determines if the object is a prototype of another object. (Prototypes are discussed in Chapter 5.)
- `propertyIsEnumerable(propertyName)` — Indicates if the given property can be enumerated using the `for-in` statement (discussed later in this chapter). As with `hasOwnProperty()`, the property name must be a string.
- `toLocaleString()` — Returns a string representation of the object that is appropriate for the locale of execution environment.
- `toString()` — Returns a string representation of the object.
- `valueOf()` — Returns a string, number, or Boolean equivalent of the object. It often returns the same value as `toString()`.

Since `Object` is the base for all objects in ECMAScript, every object has these base properties and methods. Chapters 5 and 6 cover the specifics of how this occurs.



Technically speaking, the behavior of objects in ECMA-262 need not necessarily apply to other objects in JavaScript. Objects that exist in the browser environment, such as those in the Browser Object Model (BOM) and Document Object Model (DOM), are considered host objects since they are provided and defined by the host implementation. Host objects aren't governed by ECMA-262 and, as such, may or may not directly inherit from `Object`.

OPERATORS

ECMA-262 describes a set of *operators* that can be used to manipulate data values. The operators range from mathematical operations (such as addition and subtraction) and bitwise operators to relational operators and equality operators. Operators are unique in ECMAScript in that they can

be used on a wide range of values, including strings, numbers, Booleans, and even objects. When used on objects, operators typically call the `valueOf()` and/or `toString()` method to retrieve a value they can work with.

Unary Operators

Operators that work on only one value are called *unary operators*. They are the simplest operators in ECMAScript.

Increment/Decrement

The increment and decrement operators are taken directly from C and come in two versions: prefix and postfix. The prefix versions of the operators are placed before the variable they work on; the postfix ones are placed after the variable. To use a prefix increment, which adds 1 to a numeric value, you place two plus signs (`++`) in front of a variable like this:

```
var age = 29;  
++age;
```

In this example, the prefix increment changes the value of `age` to 30 (adding 1 to its previous value of 29). This is effectively equal to the following:

```
var age = 29;  
age = age + 1;
```

The prefix decrement acts in a similar manner, subtracting 1 from a numeric value. To use a prefix decrement, place two minus signs (`--`) before a variable, as shown here:

```
var age = 29;  
--age;
```

Here the `age` variable is decremented to 28 (subtracting 1 from 29).

When using either a prefix increment or a prefix decrement, the variable's value is changed before the statement is evaluated. (In computer science, this is usually referred to as having a *side effect*.) Consider the following:



```
var age = 29;  
var anotherAge = --age + 2;  
  
alert(age);           //outputs 28  
alert(anotherAge);  //outputs 30
```

IncrementDecrementExample01.htm

In this example, the variable `anotherAge` is initialized with the decremented value of `age` plus 2. Because the decrement happens first, `age` is set to 28, and then 2 is added, resulting in 30.

The prefix increment and decrement are equal in terms of order of precedence in a statement and are therefore evaluated left to right. Consider this example:



Available for
download on
[Wrox.com](#)

```
var num1 = 2;
var num2 = 20;
var num3 = --num1 + num2;      //equals 21
var num4 = num1 + num2;      //equals 21
```

[IncrementDecrementExample02.htm](#)

Here, num3 is equal to 21 because num1 is decremented to 1 before the addition occurs. The variable num4 also contains 21, because the addition is also done using the changed values.

The postfix versions of increment and decrement use the same syntax (++ and --, respectively) but are placed after the variable instead of before it. Postfix increment and decrement differ from the prefix versions in one important way: the increment or decrement doesn't occur until after the containing statement has been evaluated. In certain circumstances, this difference doesn't matter, as in this example:

```
var age = 29;
age++;
```

Moving the increment operator after the variable doesn't change what these statements do, because the increment is the only operation occurring. However, when mixed together with other operations, the difference becomes apparent, as in the following example:

```
var num1 = 2;
var num2 = 20;
var num3 = num1-- + num2;      //equals 22
var num4 = num1 + num2;      //equals 21
```

[IncrementDecrementExample03.htm](#)

With just one simple change in this example, using postfix decrement instead of prefix, you can see the difference. In the prefix example, num3 and num4 both ended up equal to 21, whereas this example ends with num3 equal to 22 and num4 equal to 21. The difference is that the calculation for num3 uses the original value of num1 (2) to complete the addition, whereas num4 is using the decremented value (1).

All four of these operators work on any values, meaning not just integers but strings, Booleans, floating-point values, and objects. The increment and decrement operators follow these rules regarding values:

- When used on a string that is a valid representation of a number, convert to a number and apply the change. The variable is changed from a string to a number.
- When used on a string that is not a valid number, the variable's value is set to NaN (discussed in Chapter 4). The variable is changed from a string to a number.
- When used on a Boolean value that is false, convert to 0 and apply the change. The variable is changed from a Boolean to a number.
- When used on a Boolean value that is true, convert to 1 and apply the change. The variable is changed from a Boolean to a number.

- When used on a floating-point value, apply the change by adding or subtracting 1.
- When used on an object, call its `valueOf()` method (discussed more in Chapter 5) to get a value to work with. Apply the other rules. If the result is `NaN`, then call `toString()` and apply the other rules again. The variable is changed from an object to a number.

The following example demonstrates some of these rules:



Available for download on [Wrox.com](#)

```

var s1 = "2";
var s2 = "z";
var b = false;
var f = 1.1;
var o = {
    valueOf: function() {
        return -1;
    }
};

s1++; //value becomes numeric 3
s2++; //value becomes NaN
b++; //value becomes numeric 1
f--; //value becomes 0.10000000000000009 (due to floating-point inaccuracies)
o--; //value becomes numeric -2

```

IncrementDecrementExample04.htm

Unary Plus and Minus

The *unary plus and minus operators* are familiar symbols to most developers and operate the same way in ECMAScript as they do in high-school math. The unary plus is represented by a single plus sign (+) placed before a variable and does nothing to a numeric value, as shown in this example:

```

var num = 25;
num = +num; //still 25

```

When the unary plus is applied to a nonnumeric value, it performs the same conversion as the `Number()` casting function: the Boolean values of `false` and `true` are converted to 0 and 1, string values are parsed according to a set of specific rules, and objects have their `valueOf()` and/or `toString()` method called to get a value to convert.

The following example demonstrates the behavior of the unary plus when acting on different data types:

```

var s1 = "01";
var s2 = "1.1";
var s3 = "z";
var b = false;
var f = 1.1;
var o = {
    valueOf: function() {
        return -1;
    }
};

```

```

        }
};

s1 = +s1;    //value becomes numeric 1
s2 = +s2;    //value becomes numeric 1.1
s3 = +s3;    //value becomes NaN
b = +b;      //value becomes numeric 0
f = +f;      //no change, still 1.1
o = +o;      //value becomes numeric -1

```

[UnaryPlusMinusExample01.htm](#)

The unary minus operator's primary use is to negate a numeric value, such as converting 1 into -1. The simple case is illustrated here:

```

var num = 25;
num = -num;    //becomes -25

```

When used on a numeric value, the unary minus simply negates the value (as in this example).

When used on nonnumeric values, unary minus applies all of the same rules as unary plus and then negates the result, as shown here:



Available for
download on
Wrox.com

```

var s1 = "01";
var s2 = "1.1";
var s3 = "z";
var b = false;
var f = 1.1;
var o = {
    valueOf: function() {
        return -1;
    }
};

s1 = -s1;    //value becomes numeric -1
s2 = -s2;    //value becomes numeric -1.1
s3 = -s3;    //value becomes NaN
b = -b;      //value becomes numeric 0
f = -f;      //change to -1.1
o = -o;      //value becomes numeric 1

```

[UnaryPlusMinusExample02.htm](#)

The unary plus and minus operators are used primarily for basic arithmetic but can also be useful for conversion purposes, as illustrated in the previous example.

Bitwise Operators

The next set of operators works with numbers at their very base level, with the bits that represent them in memory. All numbers in ECMAScript are stored in IEEE-754 64-bit format, but the bitwise operations do not work directly on the 64-bit representation. Instead, the value is converted

into a 32-bit integer, the operation takes place, and the result is converted back into 64 bits. To the developer, it appears that only the 32-bit integer exists, because the 64-bit storage format is transparent. With that in mind, consider how 32-bit integers work.

Signed integers use the first 31 of the 32 bits to represent the numeric value of the integer. The 32nd bit represents the sign of the number: 0 for positive or 1 for negative. Depending on the value of that bit, called the *sign bit*, the format of the rest of the number is determined. Positive numbers are stored in true binary format, with each of the 31 bits representing a power of 2, starting with the first bit (called bit 0), representing 2^0 , the second bit represents 2^1 , and so on. If any bits are unused, they are filled with 0 and essentially ignored. For example, the number 18 is represented as 000000000000000000000000000010010, or more succinctly as 10010. These are the five most significant bits and can be used, by themselves, to determine the actual value (see Figure 3-1).

Negative numbers are also stored in binary code but in a format called *two's complement*. The two's complement of a number is calculated in three steps:

1. Determine the binary representation of the absolute value (for example, to find -18, first determine the binary representation of 18).
2. Find the one's complement of the number, which essentially means that every 0 must be replaced with a 1 and vice versa.
3. Add 1 to the result.

Using this process to determine the binary representation -18, start with the binary representation of 18, which is the following:

```
0000 0000 0000 0000 0000 0001 0010
```

Next, take the one's complement, which is the inverse of this number:

```
1111 1111 1111 1111 1111 1111 1110 1101
```

Finally, add 1 to the one's complement as follows:

1111 1111 1111 1111 1111 1111 1110 1101	1
1111 1111 1111 1111 1111 1111 1110 1110	

So the binary equivalent of -18 is 111111111111111111111101110. Keep in mind that you have no access to bit 31 when dealing with signed integers.

ECMAScript does its best to keep all of this information from you. When outputting a negative number as a binary string, you get the binary code of the absolute value preceded by a minus sign, as in this example:

1	0	0	1	0
$(2^4 \times 1) + (2^3 \times 0) + (2^2 \times 0) + (2^1 \times 1) + (2^0 \times 0)$				
16	+	0	+	0
18				

FIGURE 3-1

```
var num = -18;
alert(num.toString(2));      //"-10010"
```

When you convert the number `-18` to a binary string, the result is `-10010`. The conversion process interprets the two's complement and represents it in an arguably more logical form.



By default, all integers are represented as signed in ECMAScript. There is, however, such a thing as an unsigned integer. In an unsigned integer, the 32nd bit doesn't represent the sign, because there are only positive numbers. Unsigned integers also can be larger, because the extra bit becomes part of the number instead of an indicator of the sign.

When you apply bitwise operators to numbers in ECMAScript, a conversion takes place behind the scenes: the 64-bit number is converted into a 32-bit number, the operation is performed, and then the 32-bit result is stored back into a 64-bit number. This gives the illusion that you're dealing with true 32-bit numbers, which makes the binary operations work in a way similar to the operations of other languages. A curious side effect of this conversion is that the special values `Nan` and `Infinity` both are treated as equivalent to 0 when used in bitwise operations.

If a bitwise operator is applied to a nonnumeric value, the value is first converted into a number using the `Number()` function (this is done automatically) and then the bitwise operation is applied. The resulting value is a number.

Bitwise NOT

The bitwise NOT is represented by a tilde (`~`) and simply returns the one's complement of the number. Bitwise NOT is one of just a few ECMAScript operators related to binary mathematics. Consider this example:



Available for
download on
[Wrox.com](#)

```
var num1 = 25;           //binary 0000000000000000000000000011001
var num2 = ~num1;        //binary 1111111111111111111111111100110
alert(num2);            //-26
```

[BitwiseNotExample01.htm](#)

Here, the bitwise NOT operator is used on `25`, producing `-26` as the result. This is the end effect of the bitwise NOT: it negates the number and subtracts 1. The same outcome is produced with the following code:

```
var num1 = 25;
var num2 = -num1 - 1;
alert(num2);           // "-26"
```

Realistically, though this returns the same result, the bitwise operation is much faster, because it works at the very lowest level of numeric representation.

Bitwise AND

The bitwise AND operator is indicated by the ampersand character (&) and works on two values. Essentially, bitwise AND lines up the bits in each number and then, using the rules in the following truth table, performs an AND operation between the two bits in the same position.

BIT FROM FIRST NUMBER	BIT FROM SECOND NUMBER	RESULT
1	1	1
1	0	0
0	1	0
0	0	0

The short description of a bitwise AND is that the result will be 1 only if both bits are 1. If either bit is 0, then the result is 0.

As an example, to AND the numbers 25 and 3 together, use the following code:



```
var result = 25 & 3;
alert(result);           //1
```

[BitwiseAndExample01.htm](#)

The result of a bitwise AND between 25 and 3 is 1. Why is that? Take a look:

25 = 0000 0000 0000 0000 0000 0000 0001 1001
3 = 0000 0000 0000 0000 0000 0000 0011

AND = 0000 0000 0000 0000 0000 0000 0001

As you can see, only one bit (bit 0) contains a 1 in both 25 and 3. Because of this, every other bit of the resulting number is set to 0, making the result equal to 1.

Bitwise OR

The bitwise OR operator is represented by a single pipe character (|) and also works on two numbers. Bitwise OR follows the rules in this truth table:

BIT FROM FIRST NUMBER	BIT FROM SECOND NUMBER	RESULT
1	1	1
1	0	1
0	1	1
0	0	0

A bitwise OR operation returns 1 if at least one bit is 1. It returns 0 only if both bits are 0.

Using the same example as for bitwise AND, if you want to OR the numbers 25 and 3 together, the code looks like this:



Available for download on
Wrox.com

```
var result = 25 | 3;
alert(result);           //27
```

[BitwiseOrExample01.htm](#)

The result of a bitwise OR between 25 and 3 is 27:

```
25 = 0000 0000 0000 0000 0000 0000 0001 1001
 3 = 0000 0000 0000 0000 0000 0000 0000 0011
-----
OR = 0000 0000 0000 0000 0000 0000 0001 1011
```

In each number, four bits are set to 1, so these are passed through to the result. The binary code 11011 is equal to 27.

Bitwise XOR

The bitwise XOR operator is represented by a caret (^) and also works on two values. Here is the truth table for bitwise XOR:

BIT FROM FIRST NUMBER	BIT FROM SECOND NUMBER	RESULT
1	1	0
1	0	1
0	1	1
0	0	0

Bitwise XOR is different from bitwise OR in that it returns 1 only when exactly one bit has a value of 1 (if both bits contain 1, it returns 0).

To XOR the numbers 25 and 3 together, use the following code:

```
var result = 25 ^ 3;
alert(result);           //26
```

[BitwiseXorExample01.htm](#)

The result of a bitwise XOR between 25 and 3 is 26, as shown here:

```
25 = 0000 0000 0000 0000 0000 0000 0001 1001
 3 = 0000 0000 0000 0000 0000 0000 0000 0011
-----
XOR = 0000 0000 0000 0000 0000 0000 0001 1010
```

Four bits in each number are set to 1; however, the first bit in both numbers is 1, so that becomes 0 in the result. All of the other 1s have no corresponding 1 in the other number, so they are passed directly through to the result. The binary code 11010 is equal to 26. (Note that this is one less than when performing bitwise OR on these numbers.)

Left Shift

The left shift is represented by two less-than signs (`<<`) and shifts all bits in a number to the left by the number of positions given. For example, if the number 2 (which is equal to 10 in binary) is shifted 5 bits to the left, the result is 64 (which is equal to 1000000 in binary), as shown here:



```
var oldValue = 2;           //equal to binary 10
var newValue = oldValue << 5; //equal to binary 1000000 which is decimal 64
```

LeftShiftExample01.htm

Note that when the bits are shifted, five empty bits remain to the right of the number. The left shift fills these bits with 0s to make the result a complete 32-bit number (see Figure 3-2).

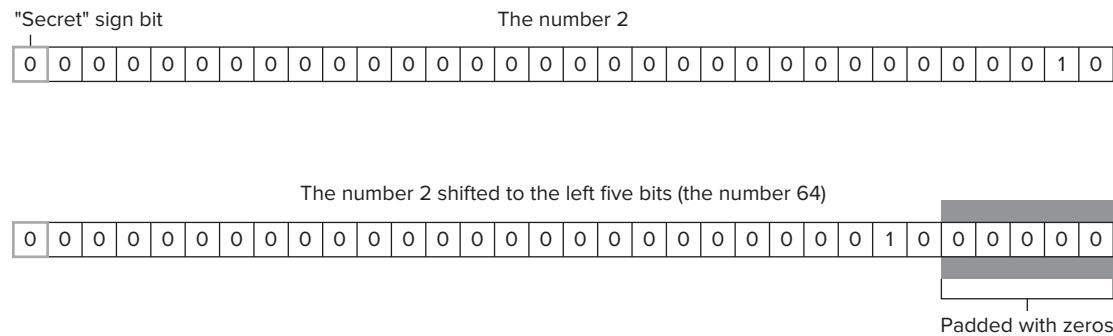


FIGURE 3-2

Note that left shift preserves the sign of the number it's operating on. For instance, if -2 is shifted to the left by five spaces, it becomes -64, not positive 64.

Signed Right Shift

The signed right shift is represented by two greater-than signs (`>>`) and shifts all bits in a 32-bit number to the right while preserving the sign (positive or negative). A signed right shift is the exact opposite of a left shift. For example, if 64 is shifted to the right five bits, it becomes 2:

```
var oldValue = 64;           //equal to binary 1000000
var newValue = oldValue >> 5; //equal to binary 10 which is decimal 2
```

SignedRightShiftExample01.htm

Once again, when bits are shifted, the shift creates empty bits. This time, the empty bits occur at the left of the number but after the sign bit (see Figure 3-3). Once again, ECMAScript fills these empty bits with the value in the sign bit to create a complete number.

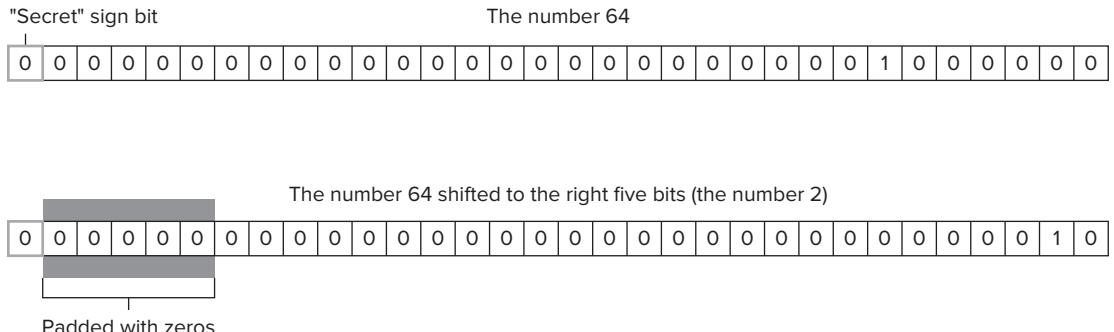


FIGURE 3-3

Unsigned Right Shift

The unsigned right shift is represented by three greater-than signs (`>>>`) and shifts all bits in a 32-bit number to the right. For numbers that are positive, the effect is the same as a signed right shift. Using the same example as for the signed-right-shift example, if 64 is shifted to the right five bits, it becomes 2:



```
var oldValue = 64;           //equal to binary 1000000
var newValue = oldValue >>> 5; //equal to binary 10 which is decimal 2
```

[UnsignedRightShiftExample01.htm](#)

For numbers that are negative, however, something quite different happens. Unlike signed right shift, the empty bits get filled with zeros regardless of the sign of the number. For positive numbers, it has the same effect as a signed right shift; for negative numbers, the result is quite different. The unsigned-right-shift operator considers the binary representation of the negative number to be representative of a positive number instead. Because the negative number is the two's complement of its absolute value, the number becomes very large, as you can see in the following example:

```
var oldValue = -64;           //equal to binary 11111111111111111111111111111110
00000
var newValue = oldValue >>> 5; //equal to decimal 134217726
```

[UnsignedRightShiftExample02.htm](#)

When an unsigned right shift is used to shift -64 to the right by five bits, the result is 134217726. This happens because the binary representation of -64 is 11111111111111111111111111111110000000,

but because the unsigned right shift treats this as a positive number, it considers the value to be 4294967232. When this value is shifted to the right by five bits, it becomes 000001111111111111111111, which is 134217726.

Boolean Operators

Almost as important as equality operators, Boolean operators are what make a programming language function. Without the capability to test relationships between two values, statements such as `if...else` and loops wouldn't be useful. There are three Boolean operators: NOT, AND, and OR.

Logical NOT

The logical NOT operator is represented by an exclamation point (!) and may be applied to any value in ECMAScript. This operator always returns a Boolean value, regardless of the data type it's used on. The logical NOT operator first converts the operand to a Boolean value and then negates it, meaning that the logical NOT behaves in the following ways:

- If the operand is an object, `false` is returned.
- If the operand is an empty string, `true` is returned.
- If the operand is a nonempty string, `false` is returned.
- If the operand is the number 0, `true` is returned.
- If the operand is any number other than 0 (including `Infinity`), `false` is returned.
- If the operand is `null`, `true` is returned.
- If the operand is `NaN`, `true` is returned.
- If the operand is `undefined`, `true` is returned.

The following example illustrates this behavior:



```
alert(!false);           //true
alert(!"blue");         //false
alert(!0);               //true
alert(!NaN);             //true
alert(!"");              //true
alert(!12345);           //false
```

LogicalNotExample01.htm

The logical NOT operator can also be used to convert a value into its Boolean equivalent. By using two NOT operators in a row, you can effectively simulate the behavior of the `Boolean()` casting function. The first NOT returns a Boolean value no matter what operand it is given. The second NOT negates that Boolean value and so gives the `true` Boolean value of a variable. The end result is the same as using the `Boolean()` function on a value, as shown here:



Available for
download on
Wrox.com

```
alert (!! "blue");      //true
alert (!! 0);          //false
alert (!!NaN);         //false
alert (!! "");          //false
alert (!!12345);        //true
```

[LogicalNotExample02.htm](#)

Logical AND

The logical AND operator is represented by the double ampersand (`&&`) and is applied to two values, such as in this example:

```
var result = true && false;
```

Logical AND behaves as described in the following truth table:

OPERAND 1	OPERAND 2	RESULT
true	true	true
true	false	false
false	true	false
false	false	false

Logical AND can be used with any type of operand, not just Boolean values. When either operand is not a primitive Boolean, logical AND does not always return a Boolean value; instead, it does one of the following:

- If the first operand is an object, then the second operand is always returned.
- If the second operand is an object, then the object is returned only if the first operand evaluates to `true`.
- If both operands are objects, then the second operand is returned.
- If either operand is `null`, then `null` is returned.
- If either operand is `NaN`, then `NaN` is returned.
- If either operand is `undefined`, then `undefined` is returned.

The logical AND operator is a short-circuited operation, meaning that if the first operand determines the result, the second operand is never evaluated. In the case of logical AND, if the first operand is `false`, no matter what the value of the second operand, the result can't be equal to `true`. Consider the following example:



```
var found = true;
var result = (found && someUndeclaredVariable);      //error occurs here
alert(result);    //this line never executes
```

[LogicalAndExample01.htm](#)

This code causes an error when the logical AND is evaluated, because the variable `someUndeclaredVariable` isn't declared. The value of the variable `found` is `true`, so the logical AND operator continued to evaluate the variable `someUndeclaredVariable`. When it did, an error occurred because `someUndeclaredVariable` is not declared and therefore cannot be used in a logical AND operation. If `found` is instead set to `false`, as in the following example, the error won't occur:

```
var found = false;
var result = (found && someUndeclaredVariable);      //no error
alert(result);    //works
```

[LogicalAndExample02.htm](#)

In this code, the alert is displayed successfully. Even though the variable `someUndeclaredVariable` is undefined, it is never evaluated, because the first operand is `false`. This means that the result of the operation must be `false`, so there is no reason to evaluate what's to the right of the `&&`. Always keep in mind short-circuiting when using logical AND.

Logical OR

The logical OR operator is represented by the double pipe (`||`) in ECMAScript, like this:

```
var result = true || false;
```

Logical OR behaves as described in the following truth table:

OPERAND 1	OPERAND 2	RESULT
true	true	true
true	false	true
false	true	true
false	false	false

Just like logical AND, if either operand is not a Boolean, logical OR will not always return a Boolean value; instead, it does one of the following:

- If the first operand is an object, then the first operand is returned.
- If the first operand evaluates to `false`, then the second operand is returned.
- If both operands are objects, then the first operand is returned.
- If both operands are `null`, then `null` is returned.

- If both operands are `Nan`, then `Nan` is returned.
- If both operands are `undefined`, then `undefined` is returned.

Also like the logical AND operator, the logical OR operator is short-circuited. In this case, if the first operand evaluates to `true`, the second operand is not evaluated. Consider this example:



Available for
download on
Wrox.com

```
var found = true;
var result = (found || someUndeclaredVariable);      //no error
alert(result);    //works
```

[LogicalOrExample01.htm](#)

As with the previous example, the variable `someUndeclaredVariable` is `undefined`. However, because the variable `found` is set to `true`, the variable `someUndeclaredVariable` is never evaluated and thus the output is "`true`". If the value of `found` is changed to `false`, an error occurs, as in the following example:

```
var found = false;
var result = (found || someUndeclaredVariable);      //error occurs here
alert(result);    //this line never executes
```

[LogicalOrExample02.htm](#)

You can also use this behavior to avoid assigning a null or `undefined` value to a variable. Consider the following:

```
var myObject = preferredObject || backupObject;
```

In this example, the variable `myObject` will be assigned one of two values. The `preferredObject` variable contains the value that is preferred if it's available, whereas the `backupObject` variable contains the backup value if the preferred one isn't available. If `preferredObject` isn't `null`, then it's assigned to `myObject`; if it is `null`, then `backupObject` is assigned to `myObject`. This pattern is used very frequently in ECMAScript for variable assignment and is used throughout this book.

Multiplicative Operators

There are three multiplicative operators in ECMAScript: multiply, divide, and modulus. These operators work in a manner similar to their counterparts in languages such as Java, C, and Perl, but they also include some automatic type conversions when dealing with nonnumeric values. If either of the operands for a multiplication operation isn't a number, it is converted to a number behind the scenes using the `Number()` casting function. This means that an empty string is treated as 0, and the Boolean value of `true` is treated as 1.

Multiply

The multiply operator is represented by an asterisk (*) and is used, as one might suspect, to multiply two numbers. The syntax is the same as in C, as shown here:

```
var result = 34 * 56;
```

However, the multiply operator also has the following unique behaviors when dealing with special values:

- If the operands are numbers, regular arithmetic multiplication is performed, meaning that two positives or two negatives equal a positive, whereas operands with different signs yield a negative. If the result cannot be represented by ECMAScript, either `Infinity` or `-Infinity` is returned.
- If either operand is `NaN`, the result is `NaN`.
- If `Infinity` is multiplied by 0, the result is `NaN`.
- If `Infinity` is multiplied by any finite number other than 0, the result is either `Infinity` or `-Infinity`, depending on the sign of the second operand.
- If `Infinity` is multiplied by `Infinity`, the result is `Infinity`.
- If either operand isn't a number, it is converted to a number behind the scenes using `Number()` and then the other rules are applied.

Divide

The divide operator is represented by a slash (/) and divides the first operand by the second operand, as shown here:

```
var result = 66 / 11;
```

The divide operator, like the multiply operator, has special behaviors for special values. They are as follows:

- If the operands are numbers, regular arithmetic division is performed, meaning that two positives or two negatives equal a positive, whereas operands with different signs yield a negative. If the result can't be represented in ECMAScript, it returns either `Infinity` or `-Infinity`.
- If either operand is `NaN`, the result is `NaN`.
- If `Infinity` is divided by `Infinity`, the result is `NaN`.
- If zero is divided by zero, the result is `NaN`.
- If a nonzero finite number is divided by zero, the result is either `Infinity` or `-Infinity`, depending on the sign of the first operand.
- If `Infinity` is divided by any number, the result is either `Infinity` or `-Infinity`, depending on the sign of the second operand.
- If either operand isn't a number, it is converted to a number behind the scenes using `Number()` and then the other rules are applied.

Modulus

The modulus (remainder) operator is represented by a percent sign (%) and is used in the following way:

```
var result = 26 % 5; //equal to 1
```

Just like the other multiplicative operators, the modulus operator behaves differently for special values, as follows:

- If the operands are numbers, regular arithmetic division is performed, and the remainder of that division is returned.
- If the dividend is an infinite number and the divisor is a finite number, the result is `NaN`.
- If the dividend is a finite number and the divisor is 0, the result is `NaN`.
- If `Infinity` is divided by `Infinity`, the result is `NaN`.
- If the dividend is a finite number and the divisor is an infinite number, then the result is the dividend.
- If the dividend is zero and the divisor is nonzero, the result is zero.
- If either operand isn't a number, it is converted to a number behind the scenes using `Number()` and then the other rules are applied.

Additive Operators

The additive operators, `add` and `subtract`, are typically the simplest mathematical operators in programming languages. In ECMAScript, however, a number of special behaviors are associated with each operator. As with the multiplicative operators, conversions occur behind the scenes for different data types. For these operators, however, the rules aren't as straightforward.

Add

The `add` operator (`+`) is used just as one would expect, as shown in the following example:

```
var result = 1 + 2;
```

If the two operands are numbers, they perform an arithmetic add and return the result according to the following rules:

- If either operand is `NaN`, the result is `NaN`.
- If `Infinity` is added to `Infinity`, the result is `Infinity`.
- If `-Infinity` is added to `-Infinity`, the result is `-Infinity`.
- If `Infinity` is added to `-Infinity`, the result is `NaN`.
- If `+0` is added to `+0`, the result is `+0`.
- If `-0` is added to `+0`, the result is `+0`.
- If `-0` is added to `-0`, the result is `-0`.

If, however, one of the operands is a string, then the following rules apply:

- If both operands are strings, the second string is concatenated to the first.
- If only one operand is a string, the other operand is converted to a string and the result is the concatenation of the two strings.

If either operand is an object, number, or Boolean, its `toString()` method is called to get a string value and then the previous rules regarding strings are applied. For `undefined` and `null`, the `String()` function is called to retrieve the values "`undefined`" and "`null`", respectively.

Consider the following:



```
var result1 = 5 + 5;      //two numbers
alert(result1);           //10
var result2 = 5 + "5";    //a number and a string
alert(result2);           //"55"
```

AddExample01.htm

This code illustrates the difference between the two modes for the add operator. Normally, `5 + 5` equals 10 (a number value), as illustrated by the first two lines of code. However, if one of the operands is changed to a string, "`5`", the result becomes "`55`" (which is a primitive string value), because the first operand gets converted to "`5`" as well.

One of the most common mistakes in ECMAScript is being unaware of the data types involved with an addition operation. Consider the following:

```
var num1 = 5;
var num2 = 10;
var message = "The sum of 5 and 10 is " + num1 + num2;
alert(message);      //"The sum of 5 and 10 is 510"
```

AddExample02.htm

In this example, the `message` variable is filled with a string that is the result of two addition operations. One might expect the final string to be "`The sum of 5 and 10 is 15`"; however, it actually ends up as "`The sum of 5 and 10 is 510`". This happens because each addition is done separately. The first combines a string with a number (5), which results in a string. The second takes that result (a string) and adds a number (10), which also results in a string. To perform the arithmetic calculation and then append that to the string, just add some parentheses like this:

```
var num1 = 5;
var num2 = 10;
var message = "The sum of 5 and 10 is " + (num1 + num2);
alert(message);      //"The sum of 5 and 10 is 15"
```

AddExample03.htm

Here, the two number variables are surrounded by parentheses, which instruct the interpreter to calculate its result before adding it to the string. The resulting string is "`The sum of 5 and 10 is 15`".

Subtract

The subtract operator (`-`) is another that is used quite frequently. Here's an example:

```
var result = 2 - 1;
```

Just like the add operator, the subtract operator has special rules to deal with the variety of type conversions present in ECMAScript. They are as follows:

- If the two operands are numbers, perform arithmetic subtract and return the result.
- If either operand is `NaN`, the result is `NaN`.
- If `Infinity` is subtracted from `Infinity`, the result is `NaN`.
- If `-Infinity` is subtracted from `-Infinity`, the result is `NaN`.
- If `-Infinity` is subtracted from `Infinity`, the result is `Infinity`.
- If `Infinity` is subtracted from `-Infinity`, the result is `-Infinity`.
- If `+0` is subtracted from `+0`, the result is `+0`.
- If `-0` is subtracted from `+0`, the result is `-0`.
- If `-0` is subtracted from `-0`, the result is `+0`.
- If either operand is a string, a Boolean, `null`, or `undefined`, it is converted to a number (using `Number()` behind the scenes) and the arithmetic is calculated using the previous rules. If that conversion results in `NaN`, then the result of the subtraction is `NaN`.
- If either operand is an object, its `valueOf()` method is called to retrieve a numeric value to represent it. If that value is `NaN`, then the result of the subtraction is `NaN`. If the object doesn't have `valueOf()` defined, then `toString()` is called and the resulting string is converted into a number.

The following are some examples of these behaviors:



Available for
download on
[Wrox.com](#)

```
var result1 = 5 - true;      //4 because true is converted to 1
var result2 = NaN - 1;       //NaN
var result3 = 5 - 3;         //2
var result4 = 5 - "";        //5 because "" is converted to 0
var result5 = 5 - "2";       //3 because "2" is converted to 2
var result6 = 5 - null;      //5 because null is converted to 0
```

[SubtractExample01.htm](#)

Relational Operators

The less-than (`<`), greater-than (`>`), less-than-or-equal-to (`<=`), and greater-than-or-equal-to (`>=`) relational operators perform comparisons between values in the same way that you learned in math class. Each of these operators returns a Boolean value, as in this example:

```
var result1 = 5 > 3;      //true
var result2 = 5 < 3;      //false
```

All snippets in this section can be found in RelationalOperatorsExample01.htm

As with other operators in ECMAScript, there are some conversions and other oddities that happen when using different data types. They are as follows:

- If the operands are numbers, perform a numeric comparison.
- If the operands are strings, compare the character codes of each corresponding character in the string.
- If one operand is a number, convert the other operand to a number and perform a numeric comparison.
- If an operand is an object, call `valueOf()` and use its result to perform the comparison according to the previous rules. If `valueOf()` is not available, call `toString()` and use that value according to the previous rules.
- If an operand is a Boolean, convert it to a number and perform the comparison.

When a relational operator is used on two strings, an interesting behavior occurs. Many expect that less-than means “alphabetically before” and greater-than means “alphabetically after,” but this is not the case. For strings, each of the first string’s character codes is numerically compared against the character codes in a corresponding location in the second string. After this comparison is complete, a Boolean value is returned. The problem here is that the character codes of uppercase letters are all lower than the character codes of lowercase letters, meaning that you can run into situations like this:

```
var result = "Brick" < "alphabet"; //true
```

In this example, the string "Brick" is considered to be less than the string "alphabet", because the letter *B* has a character code of 66 and the letter *a* has a character code of 97. To force a true alphabetic result, you must convert both operands into a common case (upper or lower) and then compare like this:

```
var result = "Brick".toLowerCase() < "alphabet".toLowerCase(); //false
```

Converting both operands to lowercase ensures that "alphabet" is correctly identified as alphabetically before "Brick".

Another sticky situation occurs when comparing numbers that are strings, such as in this example:

```
var result = "23" < "3"; //true
```

This code returns `true` when comparing the string "23" to "3". Because both operands are strings, they are compared by their character codes (the character code for "2" is 50; the character code for "3" is 51). If, however, one of the operands is changed to a number as in the following example, the result makes more sense:

```
var result = "23" < 3; //false
```

Here, the string "23" is converted into the number 23 and then compared to 3, giving the expected result. Whenever a number is compared to a string, the string is converted into a number and then

numerically compared to the other number. This works well for cases like the previous example, but what if the string can't be converted into a number? Consider this example:

```
var result = "a" < 3; //false because "a" becomes NaN
```

The letter "a" can't be meaningfully converted into a number, so it becomes NaN. As a rule, the result of any relational operation with NaN is false, which is interesting when considering the following:

```
var result1 = NaN < 3; //false
var result2 = NaN >= 3; //false
```

In most comparisons, if a value is not less than another, it is always greater than or equal to it. When using NaN, however, both comparisons return false.

Equality Operators

Determining whether two variables are equivalent is one of the most important operations in programming. This is fairly straightforward when dealing with strings, numbers, and Boolean values, but the task gets a little complicated when you take objects into account. Originally ECMAScript's equal and not-equal operators performed conversions into like types before doing a comparison. The question of whether these conversions should, in fact, take place was then raised. The end result was for ECMAScript to provide two sets of operators: *equal* and *not equal* to perform conversion before comparison, and *identically equal* and *not identically equal* to perform comparison without conversion.

Equal and Not Equal

The equal operator in ECMAScript is the double equal sign (==), and it returns true if the operands are equal. The not-equal operator is the exclamation point followed by an equal sign (!=), and it returns true if two operands are not equal. Both operators do conversions to determine if two operands are equal (often called *type coercion*).

When performing conversions, the equal and not-equal operators follow these basic rules:

- If an operand is a Boolean value, convert it into a numeric value before checking for equality. A value of false converts to 0, whereas a value of true converts to 1.
- If one operand is a string and the other is a number, attempt to convert the string into a number before checking for equality.
- If one of the operands is an object and the other is not, the valueOf() method is called on the object to retrieve a primitive value to compare according to the previous rules.

The operators also follow these rules when making comparisons:

- Values of null and undefined are equal.
- Values of null and undefined cannot be converted into any other values for equality checking.

- If either operand is `Nan`, the equal operator returns `false` and the not-equal operator returns `true`. Important note: even if both operands are `Nan`, the equal operator returns `false` because, by rule, `Nan` is not equal to `Nan`.
- If both operands are objects, then they are compared to see if they are the same object. If both operands point to the same object, then the equal operator returns `true`. Otherwise, the two are not equal.

The following table lists some special cases and their results:

EXPRESSION	VALUE
<code>null == undefined</code>	<code>true</code>
<code>"NaN" == NaN</code>	<code>false</code>
<code>5 == NaN</code>	<code>false</code>
<code>NaN == NaN</code>	<code>false</code>
<code>NaN != NaN</code>	<code>true</code>
<code>false == 0</code>	<code>true</code>
<code>true == 1</code>	<code>true</code>
<code>true == 2</code>	<code>false</code>
<code>undefined == 0</code>	<code>false</code>
<code>null == 0</code>	<code>false</code>
<code>"5" == 5</code>	<code>true</code>

[EqualityOperatorsExample01.htm](#)

Identically Equal and Not Identically Equal

The identically equal and not identically equal operators do the same thing as equal and not equal, except that they do not convert operands before testing for equality. The identically equal operator is represented by three equal signs (`==`) and returns `true` only if the operands are equal without conversion, as in this example:

```
Available for download on Wrox.com
var result1 = ("55" == 55);      //true - equal because of conversion
var result2 = ("55" === 55);     //false - not equal because different data types
```

[EqualityOperatorsExample02.htm](#)

In this code, the first comparison uses the equal operator to compare the string "55" and the number 55, which returns `true`. As mentioned previously, this happens because the string "55" is converted to the number 55 and then compared with the other number 55. The second comparison uses the identically equal operator to compare the string and the number without conversion, and of course, a string isn't equal to a number, so this outputs `false`.

The not identically equal operator is represented by an exclamation point followed by two equal signs (!==) and returns `true` only if the operands are not equal without conversion. For example:



```
var result1 = ("55" != 55);      //false - equal because of conversion
var result2 = ("55" !== 55);     //true - not equal because different data types
```

Available for download on Wrox.com

[EqualityOperatorsExample03.htm](#)

Here, the first comparison uses the not-equal operator, which converts the string "55" to the number 55, making it equal to the second operand, also the number 55. Therefore, this evaluates to `false` because the two are considered equal. The second comparison uses the not identically equal operator. It helps to think of this operation as saying, "Is the string 55 different from the number 55?" The answer to this is yes (`true`).

Keep in mind that while `null == undefined` is `true` because they are similar values, `null === undefined` is `false` because they are not the same type.



Because of the type conversion issues with the equal and not-equal operators, it is recommended to use identically equal and not identically equal instead. This helps to maintain data type integrity throughout your code.

Conditional Operator

The conditional operator is one of the most versatile in ECMAScript, and it takes on the same form as in Java, which is as follows:

```
variable = boolean_expression ? true_value : false_value;
```

This basically allows a conditional assignment to a variable depending on the evaluation of the `boolean_expression`. If it's `true`, then `true_value` is assigned to the variable; if it's `false`, then `false_value` is assigned to the variable, as in this instance:

```
var max = (num1 > num2) ? num1 : num2;
```

In this example, `max` is to be assigned the number with the highest value. The expression states that if `num1` is greater than `num2`, then `num1` is assigned to `max`. If, however, the expression is `false` (meaning that `num1` is less than or equal to `num2`), then `num2` is assigned to `max`.

Assignment Operators

Simple assignment is done with the equal sign (=) and simply assigns the value on the right to the variable on the left, as shown in the following example:

```
var num = 10;
```

Compound assignment is done with one of the multiplicative, additive, or bitwise-shift operators followed by an equal sign (=). These assignments are designed as shorthand for such common situations as this:

```
var num = 10;  
num = num + 10;
```

The second line of code can be replaced with a compound assignment:

```
var num = 10;  
num += 10;
```

Compound-assignment operators exist for each of the major mathematical operations and a few others as well. They are as follows:

- Multiply/assign (*=)
- Divide/assign (/=)
- Modulus/assign (%=)
- Add/assign (+=)
- Subtract/assign (-=)
- Left shift/assign (<<=)
- Signed right shift/assign (>>=)
- Unsigned right shift/assign (>>>=)

These operators are designed specifically as shorthand ways of achieving operations. They do not represent any performance improvement.

Comma Operator

The comma operator allows execution of more than one operation in a single statement, as illustrated here:

```
var num1=1, num2=2, num3=3;
```

Most often, the comma operator is used in the declaration of variables; however, it can also be used to assign values. When used in this way, the comma operator always returns the last item in the expression, as in the following example:

```
var num = (5, 1, 4, 8, 0); //num becomes 0
```

In this example, `num` is assigned the value of 0 because it is the last item in the expression. There aren't many times when commas are used in this way; however, it is helpful to understand that this behavior exists.

STATEMENTS

ECMA-262 describes several statements (also called *flow-control statements*). Essentially, statements define most of the syntax of ECMAScript and typically use one or more keywords to accomplish a given task. Statements can be simple, such as telling a function to exit, or complicated, such as specifying a number of commands to be executed repeatedly.

The if Statement

One of the most frequently used statements in most programming languages is the `if` statement. The `if` statement has the following syntax:

```
if (condition) statement1 else statement2
```

The condition can be any expression; it doesn't even have to evaluate to an actual Boolean value. ECMAScript automatically converts the result of the expression into a Boolean by calling the `Boolean()` casting function on it. If the condition evaluates to `true`, `statement1` is executed; if the condition evaluates to `false`, `statement2` is executed. Each of the statements can be either a single line or a code block (a group of code lines enclosed within braces). Consider this example:



```
if (i > 25)
    alert("Greater than 25.");      //one-line statement
else {
    alert("Less than or equal to 25."); //block statement
}
```

[IfStatementExample01.htm](#)

It's considered best coding practice to always use block statements, even if only one line of code is to be executed. Doing so can avoid confusion about what should be executed for each condition.

You can also chain `if` statements together like so:

```
if (condition1) statement1 else if (condition2) statement2 else statement3
```

Here's an example:

```
if (i > 25) {
    alert("Greater than 25.");
} else if (i < 0) {
    alert("Less than 0.");
} else {
    alert("Between 0 and 25, inclusive.");
}
```

[IfStatementExample02.htm](#)

The do-while Statement

The `do-while` statement is a post-test loop, meaning that the escape condition is evaluated only after the code inside the loop has been executed. The body of the loop is always executed at least once before the expression is evaluated. Here's the syntax:

```
do {
    statement
} while (expression);
```

And here's an example of its usage:



```
var i = 0;
do {
    i += 2;
} while (i < 10);
```

DoWhileStatementExample01.htm

In this example, the loop continues as long as `i` is less than 10. The variable starts at 0 and is incremented by two each time through the loop.



Post-test loops such as this are most often used when the body of the loop should be executed at least once before exiting.

The while Statement

The `while` statement is a pretest loop. This means the escape condition is evaluated before the code inside the loop has been executed. Because of this, it is possible that the body of the loop is never executed. Here's the syntax:

```
while(expression) statement
```

And here's an example of its usage:

```
var i = 0;
while (i < 10) {
    i += 2;
}
```

WhileStatementExample01.htm

In this example, the variable `i` starts out equal to 0 and is incremented by two each time through the loop. As long as the variable is less than 10, the loop will continue.

The for Statement

The `for` statement is also a pretest loop with the added capabilities of variable initialization before entering the loop and defining postloop code to be executed. Here's the syntax:

```
for (initialization; expression; post-loop-expression) statement
```

And here's an example of its usage:



```
var count = 10;
for (var i=0; i < count; i++){
    alert(i);
}
```

[ForStatementExample01.htm](#)

This code defines a variable `i` that begins with the value 0. The `for` loop is entered only if the conditional expression (`i < count`) evaluates to `true`, making it possible that the body of the code might not be executed. If the body is executed, the postloop expression is also executed, iterating the variable `i`. This `for` loop is the same as the following:

```
var count = 10;
var i = 0;
while (i < count){
    alert(i);
    i++;
}
```

Nothing can be done with a `for` loop that can't be done using a `while` loop. The `for` loop simply encapsulates the loop-related code into a single location.

It's important to note that there's no need to use the `var` keyword inside the `for` loop initialization. It can be done outside the initialization as well, such as the following:

```
var count = 10;
var i;
for (i=0; i < count; i++){
    alert(i);
}
```

[ForStatementExample02.htm](#)

This code has the same affect as having the declaration of the variable inside the loop initialization. There are no block-level variables in ECMAScript (discussed further in Chapter 4), so a variable defined inside the loop is accessible outside the loop as well. For example:

```
var count = 10;
for (var i=0; i < count; i++){
    alert(i);
}
alert(i); //10
```

[ForStatementExample03.htm](#)

In this example, an alert displays the final value of the variable `i` after the loop has completed. This displays the number 10, because the variable `i` is still accessible even though it was defined inside the loop.

The initialization, control expression, and postloop expression are all optional. You can create an infinite loop by omitting all three, like this:

```
for (;;) { //infinite loop
    doSomething();
}
```

Including only the control expression effectively turns a `for` loop into a `while` loop, as shown here:



```
var count = 10;
var i = 0;
for (; i < count; ){
    alert(i);
    i++;
}
```

ForStatementExample04.htm

This versatility makes the `for` statement one of the most used in the language.

The for-in Statement

The `for-in` statement is a strict iterative statement. It is used to enumerate the properties of an object. Here's the syntax:

```
for (property in expression) statement
```

And here's an example of its usage:

```
for (var propName in window) {
    document.write(propName);
}
```

ForInStatementExample01.htm

Here, the `for-in` statement is used to display all the properties of the BOM `window` object. Each time through the loop, the `propName` variable is filled with the name of a property that exists on the `window` object. This continues until all of the available properties have been enumerated over. As with the `for` statement, the `var` operator in the control statement is not necessary but is recommended for ensuring the use of a local variable.

Object properties in ECMAScript are unordered, so the order in which property names are returned in a `for-in` statement cannot necessarily be predicted. All enumerable properties will be returned once, but the order may differ across browsers.

Note that the `for-in` statement will throw an error if the variable representing the object to iterate over is `null` or `undefined`. ECMAScript 5 updates this behavior to not throw an error and simply doesn't execute the body of the loop. For best cross-browser compatibility, it's recommended to check that the object value isn't `null` or `undefined` before attempting to use a `for-in` loop.



In versions of Safari earlier than 3, the `for-in` statement had a bug in which some properties were returned twice.

Labeled Statements

It is possible to label statements for later use with the following syntax:

```
label: statement
```

Here's an example:

```
start: for (var i=0; i < count; i++) {
    alert(i);
}
```

In this example, the label `start` can be referenced later by using the `break` or `continue` statement. Labeled statements are typically used with nested loops.

The `break` and `continue` Statements

The `break` and `continue` statements provide stricter control over the execution of code in a loop. The `break` statement exits the loop immediately, forcing execution to continue with the next statement after the loop. The `continue` statement, on the other hand, exits the loop immediately, but execution continues from the top of the loop. Here's an example:



Available for
download on
[Wrox.com](#)

```
var num = 0;
for (var i=1; i < 10; i++) {
    if (i % 5 == 0) {
        break;
    }
    num++;
}
alert(num);      //4
```

[BreakStatementExample01.htm](#)

In this code, the `for` loop increments the variable `i` from 1 to 10. In the body of loop, an `if` statement checks to see if the value of `i` is evenly divisible by 5 (using the modulus operator). If so, the `break` statement is executed and the loop is exited. The `num` variable starts out at 0 and indicates

the number of times the loop has been executed. After the `break` statement has been hit, the next line of code to be executed is the `alert`, which displays 4. So the number of times the loop has been executed is four because when `i` equals 5, the `break` statement causes the loop to be exited before `num` can be incremented. A different effect can be seen if `break` is replaced with `continue` like this:



Available for download on Wrox.com

```
var num = 0;

for (var i=1; i < 10; i++) {
    if (i % 5 == 0) {
        continue;
    }
    num++;
}

alert(num);      //8
```

ContinueStatementExample01.htm

Here, the `alert` displays 8, the number of times the loop has been executed. When `i` reaches a value of 5, the loop is exited before `num` is incremented, but execution continues with the next iteration, when the value is 6. The loop then continues until its natural completion, when `i` is 10. The final value of `num` is 8 instead of 9, because one increment didn't occur because of the `continue` statement.

Both the `break` and `continue` statements can be used in conjunction with labeled statements to return to a particular location in the code. This is typically used when there are loops inside of loops, as in the following example:

```
var num = 0;

outermost:
for (var i=0; i < 10; i++) {
    for (var j=0; j < 10; j++) {
        if (i == 5 && j == 5) {
            break outermost;
        }
        num++;
    }
}

alert(num);      //55
```

BreakStatementExample02.htm

In this example, the `outermost` label indicates the first `for` statement. Each loop normally executes 10 times, meaning that the `num++` statement is normally executed 100 times and, consequently, `num` should be equal to 100 when the execution is complete. The `break` statement here is given one argument: the label to break to. Adding the label allows the `break` statement to break not just out of the inner `for` statement (using the variable `j`) but also out of the outer `for` statement (using the variable `i`). Because of this, `num` ends up with a value of 55, because execution is halted when

both `i` and `j` are equal to 5. The `continue` statement can be used in the same way, as shown in the following example:



Available for download on Wrox.com

```
var num = 0;

outermost:
for (var i=0; i < 10; i++) {
    for (var j=0; j < 10; j++) {
        if (i == 5 && j == 5) {
            continue outermost;
        }
        num++;
    }
}

alert(num);      //95
```

[ContinueStatementExample02.htm](#)

In this case, the `continue` statement forces execution to continue — not in the inner loop but in the outer loop. When `j` is equal to 5, `continue` is executed, which means that the inner loop misses five iterations, leaving `num` equal to 95.

Using labeled statements in conjunction with `break` and `continue` can be very powerful but can cause debugging problems if overused. Always use descriptive labels and try not to nest more than a few loops.

The `with` Statement

The `with` statement sets the scope of the code within a particular object. The syntax is as follows:

```
with (expression) statement;
```

The `with` statement was created as a convenience for times when a single object was being coded to over and over again, such as in this example:

```
var qs = location.search.substring(1);
var hostName = location.hostname;
var url = location.href;
```

Here, the `location` object is used on every line. This code can be rewritten using the `with` statement as follows:

```
with(location){
    var qs = search.substring(1);
    var hostName = hostname;
    var url = href;
}
```

[WithStatementExample01.htm](#)

In this rewritten version of the code, the `with` statement is used in conjunction with the `location` object. This means that each variable inside the statement is first considered to be a local variable. If it's not found to be a local variable, the `location` object is searched to see if it has a property of the same name. If so, then the variable is evaluated as a property of `location`.

In strict mode, the `with` statement is not allowed and is considered a syntax error.



It is widely considered a poor practice to use the `with` statement in production code because of its negative performance impact and the difficulty in debugging code contained in the `with` statement.

The switch Statement

Closely related to the `if` statement is the `switch` statement, another flow-control statement adopted from other languages. The syntax for the `switch` statement in ECMAScript closely resembles the syntax in other C-based languages, as you can see here:

```
switch (expression) {  
    case value: statement  
        break;  
    default: statement  
}
```

Each case in a `switch` statement says, “If the expression is equal to the value, execute the statement.” The `break` keyword causes code execution to jump out of the `switch` statement. Without the `break` keyword, code execution falls through the original case into the following one. The `default` keyword indicates what is to be done if the expression does not evaluate to one of the cases. (In effect, it is an `else` statement.)

Essentially, the `switch` statement prevents a developer from having to write something like this:

```
if (i == 25){  
    alert("25");  
} else if (i == 35) {  
    alert("35");  
} else if (i == 45) {  
    alert("45");  
} else {  
    alert("Other");  
}
```

The equivalent `switch` statement is as follows:



```
switch (i) {
    case 25:
        alert("25");
        break;
    case 35:
        alert("35");
        break;
    case 45:
        alert("45");
        break;
    default:
        alert("Other");
}
```

[SwitchStatementExample01.htm](#)

It's best to always put a `break` statement after each case to avoid having cases fall through into the next one. If you need a case statement to fall through, include a comment indicating that the omission of the `break` statement is intentional, such as this:

```
switch (i) {
    case 25:
        /* falls through */
    case 35:
        alert("25 or 35");
        break;
    case 45:
        alert("45");
        break;
    default:
        alert("Other");
}
```

[SwitchStatementExample02.htm](#)

Although the `switch` statement was borrowed from other languages, it has some unique characteristics in ECMAScript. First, the `switch` statement works with all data types (in many languages it works only with numbers), so it can be used with strings and even with objects. Second, the case values need not be constants; they can be variables and even expressions. Consider the following example:

```
switch ("hello world") {
    case "hello" + " world":
        alert("Greeting was found.");
        break;
    case "goodbye":
        alert("Closing was found.");
        break;
    default:
        alert("Unexpected message was found.");
}
```

[SwitchStatementExample03.htm](#)

In this example, a string value is used in a `switch` statement. The first case is actually an expression that evaluates a string concatenation. Because the result of this concatenation is equal to the `switch` argument, the alert displays "Greeting was found." The ability to have case expressions also allows you to do things like this:



```
var num = 25;
switch (true) {
    case num < 0:
        alert("Less than 0.");
        break;
    case num >= 0 && num <= 10:
        alert("Between 0 and 10.");
        break;
    case num > 10 && num <= 20:
        alert("Between 10 and 20.");
        break;
    default:
        alert("More than 20.");
}
```

[SwitchStatementExample04.htm](#)

Here, a variable `num` is defined outside the `switch` statement. The expression passed into the `switch` statement is `true`, because each case is a conditional that will return a Boolean value. Each case is evaluated, in order, until a match is found or until the `default` statement is encountered (which is the case here).



The switch statement compares values using the identically equal operator, so no type coercion occurs (for example, the string "10" is not equal to the number 10).

FUNCTIONS

Functions are the core of any language, because they allow the encapsulation of statements that can be run anywhere and at any time. Functions in ECMAScript are declared using the `function` keyword, followed by a set of arguments and then the body of the function. The basic syntax is as follows:

```
function functionName(arg0, arg1,...,argN) {
    statements
}
```

Here's an example:

```
function sayHi(name, message) {
    alert("Hello " + name + ", " + message);
}
```

[FunctionExample01.htm](#)

This function can then be called by using the function name, followed by the function arguments enclosed in parentheses (and separated by commas, if there are multiple arguments). The code to call the `sayHi()` function looks like this:

```
sayHi("Nicholas", "how are you today?");
```

The output of this function call is, "Hello Nicholas, how are you today?" The named arguments `name` and `message` are used as part of a string concatenation that is ultimately displayed in an alert.

Functions in ECMAScript need not specify whether they return a value. Any function can return a value at any time by using the `return` statement followed by the value to return. Consider this example:



Available for
download on
Wrox.com

```
function sum(num1, num2) {
    return num1 + num2;
}
```

[FunctionExample02.htm](#)

The `sum()` function adds two values together and returns the result. Note that aside from the `return` statement, there is no special declaration indicating that the function returns a value. This function can be called using the following:

```
var result = sum(5, 10);
```

Keep in mind that a function stops executing and exits immediately when it encounters the `return` statement. Therefore, any code that comes after a `return` statement will never be executed. For example:

```
function sum(num1, num2) {
    return num1 + num2;
    alert("Hello world"); //never executed
}
```

In this example, the alert will never be displayed because it appears after the `return` statement.

It's also possible to have more than one `return` statement in a function, like this:

```
function diff(num1, num2) {
    if (num1 < num2) {
        return num2 - num1;
    } else {
        return num1 - num2;
    }
}
```

[FunctionExample03.htm](#)

Here, the `diff()` function determines the difference between two numbers. If the first number is less than the second, it subtracts the first from the second; otherwise it subtracts the second from the first. Each branch of the code has its own `return` statement that does the correct calculation.

The `return` statement can also be used without specifying a return value. When used in this way, the function stops executing immediately and returns `undefined` as its value. This is typically used in functions that don't return a value to stop function execution early, as in the following example, where the alert won't be displayed:



```
function sayHi(name, message) {
    return;
    alert("Hello " + name + ", " + message);      //never called
}
```

FunctionExample04.htm



It's recommended that a function either always return a value or never return a value. Writing a function that sometimes returns a value causes confusion, especially during debugging.

Strict mode places several restrictions on functions:

- No function can be named `eval` or `arguments`.
- No named parameter can be named `eval` or `arguments`.
- No two named parameters can have the same name.

If these occur, it's considered a syntax error and the code will not execute.

Understanding Arguments

Function arguments in ECMAScript don't behave in the same way as function arguments in most other languages. An ECMAScript function doesn't care how many arguments are passed in, nor does it care about the data types of those arguments. Just because you define a function to accept two arguments doesn't mean you can pass in only two arguments. You could pass in one or three or none, and the interpreter won't complain. This happens because arguments in ECMAScript are represented as an array internally. The array is always passed to the function, but the function doesn't care what (if anything) is in the array. If the array arrives with zero items, that's fine; if it arrives with more, that's okay too. In fact, there actually is an `arguments` object that can be accessed while inside a function to retrieve the values of each argument that was passed in.

The `arguments` object acts like an array (though it isn't an instance of `Array`) in that you can access each argument using bracket notation (the first argument is `arguments[0]`, the second is `arguments[1]`, and so on) and determine how many arguments were passed in by using the `length` property. In the previous example, the `sayHi()` function's first argument is named `name`. The same

value can be accessed by referencing `arguments[0]`. Therefore, the function can be rewritten without naming the arguments explicitly, like this:



```
function sayHi() {
    alert("Hello " + arguments[0] + ", " + arguments[1]);
}
```

[FunctionExample05.htm](#)

In this rewritten version of the function, there are no named arguments. The `name` and `message` arguments have been removed, yet the function will behave appropriately. This illustrates an important point about functions in ECMAScript: named arguments are a convenience, not a necessity. Unlike in other languages, naming your arguments in ECMAScript does not create a function signature that must be matched later on; there is no validation against named arguments.

The `arguments` object can also be used to check the number of arguments passed into the function via the `length` property. The following example outputs the number of arguments passed into the function each time it is called:

```
function howManyArgs() {
    alert(arguments.length);
}

howManyArgs("string", 45);      //2
howManyArgs();                  //0
howManyArgs(12);                //1
```

[FunctionExample06.htm](#)

This example shows alerts displaying 2, 0, and 1 (in that order). In this way, developers have the freedom to let functions accept any number of arguments and behave appropriately. Consider the following:

```
function doAdd() {
    if(arguments.length == 1) {
        alert(arguments[0] + 10);
    } else if (arguments.length == 2) {
        alert(arguments[0] + arguments[1]);
    }
}

doAdd(10);          //20
doAdd(30, 20);     //50
```

[FunctionExample07.htm](#)

The function `doAdd()` adds 10 to a number only if there is one argument; if there are two arguments, they are simply added together and returned. So `doAdd(10)` returns 20, whereas

`doAdd(30, 20)` returns 50. It's not quite as good as overloading, but it is a sufficient workaround for this ECMAScript limitation.

Another important thing to understand about arguments is that the `arguments` object can be used in conjunction with named arguments, such as the following:



```
function doAdd(num1, num2) {  
    if(arguments.length == 1) {  
        alert(num1 + 10);  
    } else if (arguments.length == 2) {  
        alert(arguments[0] + num2);  
    }  
}
```

FunctionExample08.htm

In this rewrite of the `doAdd()` function, two-named arguments are used in conjunction with the `arguments` object. The named argument `num1` holds the same value as `arguments[0]`, so they can be used interchangeably (the same is true for `num2` and `arguments[1]`).

Another interesting behavior of `arguments` is that its values always stay in sync with the values of the corresponding named parameters. For example:

```
function doAdd(num1, num2) {  
    arguments[1] = 10;  
    alert(arguments[0] + num2);  
}
```

FunctionExample09.htm

This version of `doAdd()` always overwrites the second argument with a value of 10. Because values in the `arguments` object are automatically reflected by the corresponding named `arguments`, the change to `arguments[1]` also changes the value of `num2`, so both have a value of 10. This doesn't mean that both access the same memory space, though; their memory spaces are separate but happen to be kept in sync. This effect goes only one way: changing the named argument does *not* result in a change to the corresponding value in `arguments`. Another thing to keep in mind: if only one argument is passed in, then setting `arguments[1]` to a value will not be reflected by the named argument. This is because the length of the `arguments` object is set based on the number of arguments passed in, not the number of named arguments listed for the function.

Any named argument that is not passed into the function is automatically assigned the value `undefined`. This is akin to defining a variable without initializing it. For example, if only one argument is passed into the `doAdd()` function, then `num2` has a value of `undefined`.

Strict mode makes several changes to how the `arguments` object can be used. First, assignment, as in the previous example, no longer works. The value of `num2` remains `undefined` even though `arguments[1]` has been assigned to 10. Second, trying to overwrite the value of `arguments` is a syntax error. (The code will not execute.)



All arguments in ECMAScript are passed by value. It is not possible to pass arguments by reference.

No Overloading

ECMAScript functions cannot be overloaded in the traditional sense. In other languages, such as Java, it is possible to write two definitions of a function so long as their signatures (the type and number of arguments accepted) are different. As just covered, functions in ECMAScript don't have signatures, because the arguments are represented as an array containing zero or more values. Without function signatures, true overloading is not possible.

If two functions are defined to have the same name in ECMAScript, it is the last function that becomes the owner of that name. Consider the following example:



Available for download on
Wrox.com

```
function addSomeNumber(num) {
    return num + 100;
}

function addSomeNumber(num) {
    return num + 200;
}

var result = addSomeNumber(100);      //300
```

[FunctionExample10.htm](#)

Here, the function `addSomeNumber()` is defined twice. The first version of the function adds 100 to the argument, and the second adds 200. When the last line is called, it returns 300 because the second function has overwritten the first.

As mentioned previously, it's possible to simulate overloading of methods by checking the type and number of arguments that have been passed into a function and then reacting accordingly.

SUMMARY

The core language features of JavaScript are defined in ECMA-262 as a pseudolanguage named ECMAScript. ECMAScript contains all of the basic syntax, operators, data types, and objects necessary to complete basic computing tasks, though it provides no way to get input or to produce output. Understanding ECMAScript and its intricacies is vital to a complete understanding of JavaScript as implemented in web browsers. The most widely implemented version of ECMAScript is the one defined in ECMA-262, third edition, though many are starting to implement the fifth edition. The following are some of the basic elements of ECMAScript:

- The basic data types in ECMAScript are `Undefined`, `Null`, `Boolean`, `Number`, and `String`.
- Unlike other languages, there's no separate data type for integers versus floating-point values; the `Number` type represents all numbers.

- There is also a complex data type, Object, that is the base type for all objects in the language.
- A strict mode places restrictions on certain error-prone parts of the language.
- ECMAScript provides a lot of the basic operators available in C and other C-like languages, including arithmetic operators, Boolean operators, relational operators, equality operators, and assignment operators.
- The language features flow-control statements borrowed heavily from other languages, such as the `if` statement, the `for` statement, and the `switch` statement.

Functions in ECMAScript behave differently than functions in other languages:

- There is no need to specify the return value of the function since any function can return any value at any time.
- Functions that don't specify a return value actually return the special value `undefined`.
- There is no such thing as a function signature, because arguments are passed as an array containing zero or more values.
- Any number of arguments can be passed into a function and are accessible through the `arguments` object.
- Function overloading is not possible because of the lack of function signatures.

4

Variables, Scope, and Memory

WHAT'S IN THIS CHAPTER?

- Working with primitive and reference values in variables
- Understanding execution context
- Understanding garbage collection

The nature of variables in JavaScript, as defined in ECMA-262, is quite unique compared to that of other languages. Being loosely typed, a variable is literally just a name for a particular value at a particular time. Because there are no rules defining the type of data that a variable must hold, a variable's value and data type can change during the lifetime of a script. Though this is an interesting, powerful, and problematic feature, there are many more complexities related to variables.

PRIMITIVE AND REFERENCE VALUES

ECMAScript variables may contain two different types of data: primitive values and reference values. *Primitive values* are simple atomic pieces of data, while *reference values* are objects that may be made up of multiple values.

When a value is assigned to a variable, the JavaScript engine must determine if it's a primitive or a reference. The five primitive types were discussed in the previous chapter: Undefined, Null, Boolean, Number, and String. These variables are said to be accessed *by value*, because you are manipulating the actual value stored in the variable.

Reference values are objects stored in memory. Unlike other languages, JavaScript does not permit direct access of memory locations, so direct manipulation of the object's memory space is not allowed. When you manipulate an object, you're really working on a *reference* to that object rather than the actual object itself. For this reason, such values are said to be accessed *by reference*.



In many languages, strings are represented by objects and are therefore considered to be reference types. ECMAScript breaks away from this tradition.

Dynamic Properties

Primitive and reference values are defined similarly: a variable is created and assigned a value. What you can do with those values once they're stored in a variable, however, is quite different. When you work with reference values, you can add, change, or delete properties and methods at any time. Consider this example:



```
var person = new Object();
person.name = "Nicholas";
alert(person.name);      // "Nicholas"
```

DynamicPropertiesExample01.htm

Here, an object is created and stored in the variable `person`. Next, a property called `name` is added and assigned the string value of `"Nicholas"`. The new property is then accessible from that point on, until the object is destroyed or the property is explicitly removed.

Primitive values can't have properties added to them even though attempting to do so won't cause an error. Here's an example:

```
var name = "Nicholas";
name.age = 27;
alert(name.age);      // undefined
```

DynamicPropertiesExample02.htm

Here, a property called `age` is defined on the string `name` and assigned a value of 27. On the very next line, however, the property is gone. Only reference values can have properties defined dynamically for later use.

Copying Values

Aside from differences in how they are stored, primitive and reference values act differently when copied from one variable to another. When a primitive value is assigned from one variable to another, the value stored on the variable object is created and copied into the location for the new variable. Consider the following example:

```
var num1 = 5;
var num2 = num1;
```

Here, `num1` contains the value of 5. When `num2` is initialized to `num1`, it also gets the value of 5. This value is completely separate from the one that is stored in `num1`, because it's a copy of that value.

Each of these variables can now be used separately with no side effects. This process is diagrammed in Figure 4-1.

When a reference value is assigned from one variable to another, the value stored on the variable object is also copied into the location for the new variable. The difference is that this value is actually a pointer to an object stored on the heap. Once the operation is complete, two variables point to exactly the same object, so changes to one are reflected on the other, as in the following example:

```
var obj1 = new Object();
var obj2 = obj1;
obj1.name = "Nicholas";
alert(obj2.name); // "Nicholas"
```

In this example, the variable `obj1` is filled with a new instance of an object. This value is then copied into `obj2`, meaning that both variables are now pointing to the same object. When the property `name` is set on `obj1`, it can later be accessed from `obj2` because they both point to the same object. Figure 4-2 shows the relationship between the variables on the variable object and the object on the heap.

Variable object before copy

num1	5 (Number type)

Variable object after copy

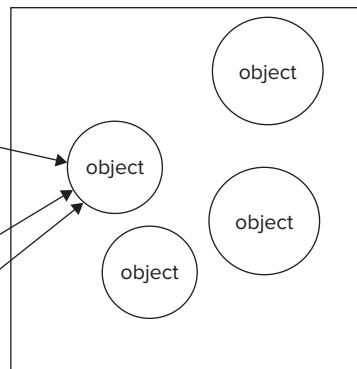
num2	5 (Number type)
num1	5 (Number type)

FIGURE 4-1

Variable object before copy

obj1	(Object type)

Heap



Variable object after copy

obj2	(Object type)
obj1	(Object type)

FIGURE 4-2

Argument Passing

All function arguments in ECMAScript are passed by value. This means that the value outside of the function is copied into an argument on the inside of the function the same way a value is copied from one variable to another. If the value is primitive, then it acts just like a primitive variable copy, and if the value is a reference, it acts just like a reference variable copy. This is often a point of confusion for developers, because variables are accessed both by value and by reference, but arguments are passed only by value.

When an argument is passed by value, the value is copied into a local variable (a named argument and, in ECMAScript, a slot in the `arguments` object). When an argument is passed by reference, the location of the value in memory is stored into a local variable, which means that changes to the local variable are reflected outside of the function. (This is not possible in ECMAScript.) Consider the following example:



```
function addTen(num) {
    num += 10;
    return num;
}

var count = 20;
var result = addTen(count);
alert(count);      //20 - no change
alert(result);     //30
```

FunctionArgumentsExample01.htm

Here, the function `addTen()` has an argument `num`, which is essentially a local variable. When called, the variable `count` is passed in as an argument. This variable has a value of 20, which is copied into the argument `num` for use inside of `addTen()`. Within the function, the argument `num` has its value changed by adding 10, but this doesn't change the original variable `count` that exists outside of the function. The argument `num` and the variable `count` do not recognize each other; they only happen to have the same value. If `num` had been passed by reference, then the value of `count` would have changed to 30 to reflect the change made inside the function. This fact is obvious when using primitive values such as numbers, but things aren't as clear when using objects. Take this for example:

```
function setName(obj) {
    obj.name = "Nicholas";
}

var person = new Object();
setName(person);
alert(person.name);      //"Nicholas"
```

FunctionArgumentsExample02.htm

In this code, an object is created and stored in the variable `person`. This object is then passed into the `setName()` method, where it is copied into `obj`. Inside the function, `obj` and `person` both point to the same object. The result is that `obj` is accessing an object by reference, even though it

was passed into the function by value. When the `name` property is set on `obj` inside the function, this change is reflected outside the function, because the object that it points to exists globally on the heap. Many developers incorrectly assume that when a local change to an object is reflected globally, that means an argument was passed by reference. To prove that objects are passed by value, consider the following modified code:

```
function setName(obj) {
    obj.name = "Nicholas";
    obj = new Object();
    obj.name = "Greg";
}

var person = new Object();
setName(person);
alert(person.name);      // "Nicholas"
```

The only change between this example and the previous one are two lines added to `setName()` that redefine `obj` as a new object with a different name. When `person` is passed into `setName()`, its `name` property is set to "Nicholas". Then the variable `obj` is set to be a new object and its `name` property is set to "Greg". If `person` were passed by reference, then `person` would automatically be changed to point to the object whose name is "Greg". However, when `person.name` is accessed again, its value is "Nicholas", indicating that the original reference remained intact even though the argument's value changed inside the function. When `obj` is overwritten inside the function, it becomes a pointer to a local object. That local object is destroyed as soon as the function finishes executing.



Think of function arguments in ECMAScript as nothing more than local variables.

Determining Type

The `typeof` operator, introduced in the previous chapter, is the best way to determine if a variable is a primitive type. More specifically, it's the best way to determine if a variable is a string, number, Boolean, or `undefined`. If the value is an object or `null`, then `typeof` returns "object", as in this example:



Available for
download on
[Wrox.com](#)

```
var s = "Nicholas";
var b = true;
var i = 22;
var u;
var n = null;
var o = new Object();

alert(typeof s);    //string
alert(typeof i);    //number
alert(typeof b);    //boolean
alert(typeof u);    //undefined
alert(typeof n);    //object
alert(typeof o);    //object
```

[DeterminingTypeExample01.htm](#)

Although `typeof` works well for primitive values, it's of little use for reference values. Typically, you don't care that a value is an object — what you really want to know is what type of object it is. To aid in this identification, ECMAScript provides the `instanceof` operator, which is used with the following syntax:

```
result = variable instanceof constructor
```

The `instanceof` operator returns `true` if the variable is an instance of the given reference type (identified by its prototype chain, as discussed in Chapter 6). Consider this example:

```
alert(person instanceof Object); //is the variable person an Object?  
alert(colors instanceof Array); //is the variable colors an Array?  
alert(pattern instanceof RegExp); //is the variable pattern a RegExp?
```

All reference values, by definition, are instances of `Object`, so the `instanceof` operator always returns `true` when used with a reference value and the `Object` constructor. Similarly, if `instanceof` is used with a primitive value, it will always return `false`, because primitives aren't objects.



The `typeof` operator also returns "function" when used on a function. When used on a regular expression in Safari (through version 5) and Chrome (through version 7), `typeof` returns "function" because of an implementation detail. ECMA-262 specifies that any object implementing the internal `[[Call]]` method should return "function" from `typeof`. Since regular expressions implement this method in these browsers, `typeof` returns "function". In Internet Explorer and Firefox, `typeof` returns "object" for regular expressions.

EXECUTION CONTEXT AND SCOPE

The concept of execution context, referred to as *context* for simplicity, is of the utmost importance in JavaScript. The execution context of a variable or function defines what other data it has access to, as well as how it should behave. Each execution context has an associated *variable object* upon which all of its defined variables and functions exist. This object is not accessible by code but is used behind the scenes to handle data.

The global execution context is the outermost one. Depending on the host environment for an ECMAScript implementation, the object representing this context may differ. In web browsers, the global context is said to be that of the `window` object (discussed in Chapter 8), so all global variables and functions are created as properties and methods on the `window` object. When an execution context has executed all of its code, it is destroyed, taking with it all of the variables and functions defined within it (the global context isn't destroyed until the application exits, such as when a web page is closed or a web browser is shut down).

Each function call has its own execution context. Whenever code execution flows into a function, the function's context is pushed onto a context stack. After the function has finished executing,

the stack is popped, returning control to the previously executing context. This facility controls execution flow throughout an ECMAScript program.

When code is executed in a context, a *scope chain* of variable objects is created. The purpose of the scope chain is to provide ordered access to all variables and functions that an execution context has access to. The front of the scope chain is always the variable object of the context whose code is executing. If the context is a function, then the *activation object* is used as the variable object. An activation object starts with a single defined variable called `arguments`. (This doesn't exist for the global context.) The next variable object in the chain is from the containing context, and the next after that is from the next containing context. This pattern continues until the global context is reached; the global context's variable object is always the last of the scope chain.

Identifiers are resolved by navigating the scope chain in search of the identifier name. The search always begins at the front of the chain and proceeds to the back until the identifier is found. (If the identifier isn't found, typically an error occurs.)

Consider the following code:



Available for download on Wrox.com

```
var color = "blue";

function changeColor(){
    if (color === "blue"){
        color = "red";
    } else {
        color = "blue";
    }
}

changeColor();
```

[ExecutionContextExample01.htm](#)

In this simple example, the function `changeColor()` has a scope chain with two objects in it: its own variable object (upon which the `arguments` object is defined) and the global context's variable object. The variable `color` is therefore accessible inside the function, because it can be found in the scope chain.

Additionally, locally defined variables can be used interchangeably with global variables in a local context. Here's an example:

```
var color = "blue";

function changeColor(){
    var anotherColor = "red";

    function swapColors(){
        var tempColor = anotherColor;
        anotherColor = color;
        color = tempColor;

        //color, anotherColor, and tempColor are all accessible here
    }

    //color and anotherColor are accessible here, but not tempColor
}
```

```

    swapColors();
}

//only color is accessible here
changeColor();

```

There are three execution contexts in this code: global context, the local context of `changeColor()`, and the local context of `swapColors()`. The global context has one variable, `color`, and one function, `changeColor()`. The local context of `changeColor()` has one variable named `anotherColor` and one function named `swapColors()`, but it can also access the variable `color` from the global context. The local context of `swapColors()` has one variable, named `tempColor`, that is accessible only within that context. Neither the global context nor the local context of `swapColors()` has access to `tempColor`. Within `swapColors()`, though, the variables of the other two contexts are fully accessible, because they are parent execution contexts. Figure 4-3 represents the scope chain for the previous example.

In this figure, the rectangles represent specific execution contexts. An inner context can access everything from all outer contexts through the scope chain, but the outer contexts cannot access anything within an inner context. The connection between the contexts is linear and ordered. Each context can search up the scope chain for variables and functions, but no context can search down the scope chain into another execution context. There are three objects in the scope chain for the local context of `swapColors()`: the `swapColors()` variable object, the variable object from `changeColor()`, and the global variable object. The local context of `swapColors()` begins its search for variable and function names in its own variable object before moving along the chain. The scope chain for the `changeColor()` context has only two objects: its own variable object and the global variable object. This means that it cannot access the context of `swapColors()`.

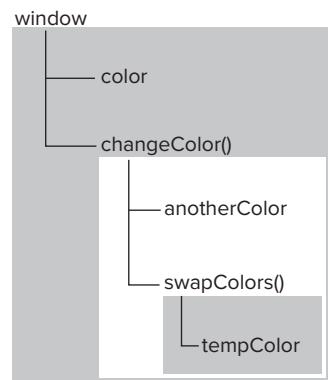


FIGURE 4-3



Function arguments are considered to be variables and follow the same access rules as any other variable in the execution context.

Scope Chain Augmentation

Even though there are only two primary types of execution contexts, global and function (the third exists inside of a call to `eval()`), there are other ways to augment the scope chain. Certain statements cause a temporary addition to the front of the scope chain that is later removed after code execution. There are two times when this occurs, specifically when execution enters either of the following:

- The catch block in a try-catch statement
- A with statement

Both of these statements add a variable object to the front of the scope chain. For the `with` statement, the specified object is added to the scope chain; for the `catch` statement, a new variable object is created and contains a declaration for the thrown error object. Consider the following:



```
function buildUrl() {
    var qs = "?debug=true";
    with(location) {
        var url = href + qs;
    }
    return url;
}
```

ExecutionContextExample03.htm

In this example, the `with` statement is acting on the `location` object, so `location` itself is added to the front of the scope chain. There is one variable, `qs`, defined in the `buildUrl()` function. When the variable `href` is referenced, it's actually referring to `location.href`, which is in its own variable object. When the variable `qs` is referenced, it's referring to the variable defined in `buildUrl()`, which is in the function context's variable object. Inside the `with` statement is a variable declaration for `url`, which becomes part of the function's context and can, therefore, be returned as the function value.



There is a deviation in the Internet Explorer implementation of JavaScript through Internet Explorer 8, where the error caught in a `catch` statement is added to the execution context's variable object rather than the `catch` statement's variable object, making it accessible even outside the `catch` block. This was fixed in Internet Explorer 9.

No Block-Level Scopes

JavaScript's lack of block-level scopes is a common source of confusion. In other C-like languages, code blocks enclosed by brackets have their own scope (more accurately described as their own execution context in ECMAScript), allowing conditional definition of variables. For example, the following code may not act as expected:

```
if (true) {
    var color = "blue";
}
alert(color); // "blue"
```

Here, the variable `color` is defined inside an `if` statement. In languages such as C, C++, and Java, that variable would be destroyed after the `if` statement is executed. In JavaScript, however, the variable declaration adds a variable into the current execution context (the global context, in

this case). This is important to keep in mind when dealing with the `for` statement, which is typically written like this:

```
for (var i=0; i < 10; i++){
    doSomething(i);
}

alert(i);      //10
```

In languages with block-level scoping, the initialization part of the `for` statement defines variables that exist only within the context of the loop. In JavaScript, the `i` variable is created by the `for` statement and continues to exist outside the loop after the statement executes.

Variable Declaration

When a variable is declared using `var`, it is automatically added to the most immediate context available. In a function, the most immediate one is the function's local context; in a `with` statement, the most immediate is the function context. If a variable is initialized without first being declared, it gets added to the global context automatically, as in this example:



```
function add(num1, num2) {
    var sum = num1 + num2;
    return sum;
}

var result = add(10, 20); //30
alert(sum);             //causes an error since sum is not a valid variable
```

ExecutionContextExample04.htm

Here, the function `add()` defines a local variable named `sum` that contains the result of an addition operation. This value is returned as the function value, but the variable `sum` isn't accessible outside the function. If the `var` keyword is omitted from this example, `sum` becomes accessible after `add()` has been called, as shown here:

```
function add(num1, num2) {
    sum = num1 + num2;
    return sum;
}

var result = add(10, 20); //30
alert(sum);             //30
```

ExecutionContextExample05.htm

Here, the variable `sum` is initialized to a value without ever having been declared using `var`. When `add()` is called, `sum` is created in the global context and continues to exist even after the function has completed, allowing you to access it later.



Initializing variables without declaring them is a very common mistake in JavaScript programming and can lead to errors. It's advisable to always declare variables before initializing them to avoid such issues. In strict mode, initializing variables without declaration causes an error.

Identifier Lookup

When an identifier is referenced for either reading or writing within a particular context, a search must take place to determine what identifier it represents. The search starts at the front of the scope chain, looking for an identifier with the given name. If it finds that identifier name in the local context, then the search stops and the variable is set; if the search doesn't find the variable name, it continues along the scope chain (note that objects in the scope chain also have a prototype chain, so searching may include each object's prototype chain). This process continues until the search reaches the global context's variable object. If the identifier isn't found there, it hasn't been declared.

To better illustrate how identifier lookup occurs, consider the following example:



Available for download on Wrox.com

```
var color = "blue";
function getColor() {
    return color;
}
alert(getColor()); // "blue"
```

[ExecutionContextExample06.htm](#)

When the function `getColor()` is called in this example, the variable `color` is referenced. At that point, a two-step search begins. First `getColor()`'s variable object is searched for an identifier named `color`. When it isn't found, the search goes to the next variable object (from the global context) and then searches for an identifier named `color`. Because that variable object is where the variable is defined, the search ends. Figure 4-4 illustrates this search process.

Given this search process, referencing local variables automatically stops the search from going into another variable object. This means that identifiers in a parent context cannot be referenced if an identifier in the local context has the same name, as in this example:

```
var color = "blue";
function getColor() {
    var color = "red";
    return color;
}
alert(getColor()); // "red"
```

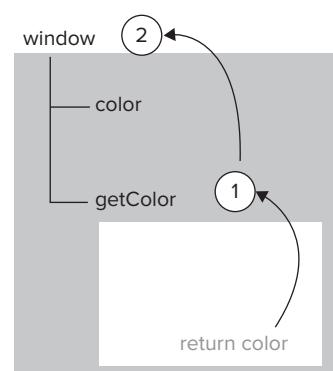


FIGURE 4-4

[ExecutionContextExample07.htm](#)

In this modified code, a local variable named `color` is declared inside the `getColor()` function. When the function is called, the variable is declared. When the second line of the function is executed, it knows that a variable named `color` must be used. The search begins in the local context, where it finds a variable named `color` with a value of "red". Because the variable was found, the search stops and the local variable is used, meaning that the function returns "red". Any lines of code appearing after the declaration of `color` as a local variable cannot access the global `color` variable without qualifying it as `window.color`. If one of the operands is an object and the other is not, the `valueOf()` method is called on the object to retrieve a primitive value to compare according to the previous rules.



Variable lookup doesn't come without a price. It's faster to access local variables than global variables because there's no search up the scope chain. JavaScript engines are getting better at optimizing identifier lookup, though, so this difference may end up negligible in the future.

GARBAGE COLLECTION

JavaScript is a garbage-collected language, meaning that the execution environment is responsible for managing the memory required during code execution. In languages like C and C++, keeping track of memory usage is a principle concern and the source of many issues for developers. JavaScript frees developers from worrying about memory management by automatically allocating what is needed and reclaiming memory that is no longer being used. The basic idea is simple: figure out which variables aren't going to be used and free the memory associated with them. This process is periodic, with the garbage collector running at specified intervals (or at predefined collection moments in code execution).

Consider the normal life cycle of a local variable in a function. The variable comes into existence during the execution of the function. At that time, memory is allocated on the stack (and possibly on the heap) to provide storage space for the value. The variable is used inside the function and then the function ends. At that point this variable is no longer needed, so its memory can be reclaimed for later use. In this situation, it's obvious that the variable isn't needed, but not all situations are as obvious. The garbage collector must keep track of which variables can and can't be used so it can identify likely candidates for memory reclamation. The strategy for identifying the unused variables may differ on an implementation basis, though two strategies have traditionally been used in browsers.

Mark-and-Sweep

The most popular form of garbage collection for JavaScript is called *mark-and-sweep*. When a variable comes into context, such as when a variable is declared inside a function, it is flagged as being in context. Variables that are in context, logically, should never have their memory freed, because they may be used as long as execution continues in that context. When a variable goes out of context, it is also flagged as being out of context.

Variables can be flagged in any number of ways. There may be a specific bit that is flipped when a variable is in context, or there may be an “in-context” variable list and an “out-of-context” variable list between which variables are moved. The implementation of the flagging is unimportant; it’s really the theory that is key.

When the garbage collector runs, it marks all variables stored in memory (once again, in any number of ways). It then clears its mark off of variables that are in context and variables that are referenced by in-context variables. The variables that are marked after that are considered ready for deletion, because they can’t be reached by any in-context variables. The garbage collector then does a *memory sweep*, destroying each of the marked values and reclaiming the memory associated with them.

As of 2008, Internet Explorer, Firefox, Opera, Chrome, and Safari all use mark-and-sweep garbage collection (or variations thereof) in their JavaScript implementations, though the timing of garbage collection differs.

Reference Counting

A second, less-popular type of garbage collection is *reference counting*. The idea is that every value keeps track of how many references are made to it. When a variable is declared and a reference value is assigned, the reference count is one. If another variable is then assigned to the same value, the reference count is incremented. Likewise, if a variable with a reference to that value is overwritten with another value, then the reference count is decremented. When the reference count of a value reaches zero, there is no way to reach that value and it is safe to reclaim the associated memory. The garbage collector frees the memory for values with a reference count of zero the next time it runs.

Reference counting was initially used by Netscape Navigator 3.0 and was immediately met with a serious issue: circular references. A *circular reference* occurs when object A has a pointer to object B and object B has a reference to object A, such as in the following example:

```
function problem() {
    var objectA = new Object();
    var objectB = new Object();

    objectA.someOtherObject = objectB;
    objectB.anotherObject = objectA;
}
```

In this example, `objectA` and `objectB` reference each other through their properties, meaning that each has a reference count of two. In a mark-and-sweep system, this wouldn’t be a problem because both objects go out of scope after the function has completed. In a reference-counting system, though, `objectA` and `objectB` will continue to exist after the function has exited, because their reference counts will never reach zero. If this function were called repeatedly, it would lead to a large amount of memory never being reclaimed. For this reason, Netscape abandoned a reference-counting garbage-collection routine in favor of a mark-and-sweep implementation in version 4.0. Unfortunately, that’s not where the reference-counting problem ended.

Not all objects in Internet Explorer 8 and earlier are native JavaScript objects. Objects in the Browser Object Model (BOM) and Document Object Model (DOM) are implemented as COM (Component Object Model) objects in C++, and COM objects use reference counting for garbage

collection. So even though the Internet Explorer JavaScript engine uses a mark-and-sweep implementation, any COM objects that are accessed in JavaScript still use reference counting, meaning circular references are still a problem when COM objects are involved. The following simple example demonstrates a circular reference with a COM object:

```
var element = document.getElementById("some_element");
var myObject = new Object();
myObject.element = element;
element.someObject = myObject;
```

This example sets up a circular reference between a DOM element (`element`) and a native JavaScript object (`myObject`). The `myObject` variable has a property called `element` that points to `element`, and the `element` variable has a property called `someObject` that points back to `myObject`. Because of this circular reference, the memory for the DOM element will never be reclaimed even if it is removed from the page.

To avoid circular reference problems such as this, you should break the connection between native JavaScript objects and DOM elements when you're finished using them. For example, the following code cleans up the circular references in the previous example:

```
myObject.element = null;
element.someObject = null;
```

Setting a variable to `null` effectively severs the connection between the variable and the value it previously referenced. The next time the garbage collector runs, these values will be deleted and the memory will be reclaimed.

Internet Explorer 9 remedied some of these problems by making BOM and DOM objects into true JavaScript objects, thus avoiding the problem of having two different garbage-collection algorithms and eliminating common memory leak issues.



There are several other patterns that may cause circular references, which will be covered throughout this book.

Performance

The garbage collector runs periodically and can potentially be an expensive process if there are a large number of variable allocations in memory, so the timing of the garbage-collection process is important. Internet Explorer was infamous for its performance issues related to how often the garbage collector ran — it ran based on the number of allocations, specifically 256 variable allocations, 4,096 object/array literals and array slots, or 64kb of strings. If any of these thresholds were reached, the garbage collector would run. The problem with this implementation is that a script with so many variables will probably continue to have that many variables throughout its lifetime, meaning the garbage collector will run quite frequently. This issue caused serious performance problems that led to changes in the garbage-collection routine in Internet Explorer 7.

With the release of Internet Explorer 7, the JavaScript engine's garbage-collection routine was tuned to dynamically change the allocation threshold of variables, literals, and/or array slots that triggered garbage collection. The Internet Explorer 7 thresholds start out equal to those in Internet Explorer 6. If the garbage-collection routine reclaims less than 15 percent of the allocations, the threshold for variables, literals, and/or array slots doubles. If the routine ever reclaims 85 percent of the allocations, then the threshold is reset to the default. This simple change greatly improved the performance of the browser on JavaScript-heavy web pages.



It's possible, though not recommended, to trigger the garbage-collection process in some browsers. In Internet Explorer, the `window.collectGarbage()` method causes garbage collection to occur immediately. In Opera 7 and higher, calling `window.opera.collect()` initiates the garbage-collection process.

Managing Memory

In a garbage-collected programming environment, developers typically don't have to worry about memory management. However, JavaScript runs in an environment where memory management and garbage collection operate uniquely. The amount of memory available for use in web browsers is typically much less than is available for desktop applications. This is more of a security feature than anything else, ensuring that a web page running JavaScript can't crash the operating system by using up all the system memory. The memory limits affect not only variable allocation but also the call stack and the number of statements that can be executed in a single thread.

Keeping the amount of used memory to a minimum leads to better page performance. The best way to optimize memory usage is to ensure that you're keeping around only data that is necessary for the execution of your code. When data is no longer necessary, it's best to set the value to `null`, freeing up the reference — this is called *dereferencing* the value. This advice applies mostly to global values and properties of global objects. Local variables are dereferenced automatically when they go out of context, as in this example:

```
function createPerson(name) {
    var localPerson = new Object();
    localPerson.name = name;
    return localPerson;
}

var globalPerson = createPerson("Nicholas");
//do something with globalPerson
globalPerson = null;
```

In this code, the variable `globalPerson` is filled with a value returned from the `createPerson()` function. Inside `createPerson()`, `localPerson` creates an object and adds a `name` property to it. The variable `localPerson` is returned as the function value and assigned to `globalPerson`. Because `localPerson` goes out of context after `createPerson()` has finished executing, it doesn't need to be

dereferenced explicitly. Because `globalPerson` is a global variable, it should be dereferenced when it's no longer needed, which is what happens in the last line.

Keep in mind that dereferencing a value doesn't automatically reclaim the memory associated with it. The point of dereferencing is to make sure the value is out of context and will be reclaimed the next time garbage collection occurs.

SUMMARY

Two types of values can be stored in JavaScript variables: primitive values and reference values. Primitive values have one of the five primitive data types: `Undefined`, `Null`, `Boolean`, `Number`, and `String`. Primitive and reference values have the following characteristics:

- Primitive values are of a fixed size and so are stored in memory on the stack.
- Copying primitive values from one variable to another creates a second copy of the value.
- Reference values are objects and are stored in memory on the heap.
- A variable containing a reference value actually contains just a pointer to the object, not the object itself.
- Copying a reference value to another variable copies just the pointer, so both variables end up referencing the same object.
- The `typeof` operator determines a value's primitive type, whereas the `instanceof` operator is used to determine the reference type of a value.

All variables, primitive and reference, exist within an execution context (also called a scope) that determines the lifetime of the variable and which parts of the code can access it. Execution context can be summarized as follows:

- Execution contexts exist globally (called the global context) and within functions.
- Each time a new execution context is entered, it creates a scope chain to search for variables and functions.
- Contexts that are local to a function have access not only to variables in that scope but also to variables in any containing contexts and the global context.
- The global context has access only to variables and functions in the global context and cannot directly access any data inside local contexts.
- The execution context of variables helps to determine when memory will be freed.

JavaScript is a garbage-collected programming environment where the developer need not be concerned with memory allocation or reclamation. JavaScript's garbage-collection routine can be summarized as follows:

- Values that go out of scope will automatically be marked for reclamation and will be deleted during the garbage-collection process.
- The predominant garbage-collection algorithm is called mark-and-sweep, which marks values that aren't currently being used and then goes back to reclaim that memory.

- Another algorithm is reference counting, which keeps track of how many references there are to a particular value. JavaScript engines no longer use this algorithm, but it still affects Internet Explorer because of nonnative JavaScript objects (such as DOM elements) being accessed in JavaScript.
- Reference counting causes problems when circular references exist in code.
- Dereferencing variables helps not only with circular references but also with garbage collection in general. To aid in memory reclamation, global objects, properties on global objects, and circular references should all be dereferenced when no longer needed.

5

Reference Types

WHAT'S IN THIS CHAPTER?

- Working with objects
- Creating and manipulating arrays
- Understanding basic JavaScript data types
- Working with primitives and primitive wrappers

A reference value (object) is an instance of a specific *reference type*. In ECMAScript, reference types are structures used to group data and functionality together and are often incorrectly called *classes*. Although technically an object-oriented language, ECMAScript lacks some basic constructs that have traditionally been associated with object-oriented programming, including classes and interfaces. Reference types are also sometimes called *object definitions*, because they describe the properties and methods that objects should have.



Even though reference types are similar to classes, the two concepts are not equivalent. To avoid any confusion, the term class is not used in the rest of this book.

Again, objects are considered to be *instances* of a particular reference type. New objects are created by using the `new` operator followed by a *constructor*. A constructor is simply a function whose purpose is to create a new object. Consider the following line of code:

```
var person = new Object();
```

This code creates a new instance of the `Object` reference type and stores it in the variable `person`. The constructor being used is `Object()`, which creates a simple object with only the

default properties and methods. ECMAScript provides a number of native reference types, such as `Object`, to help developers with common computing tasks.

THE OBJECT TYPE

Up to this point, most of the reference-value examples have used the `Object` type, which is one of the most often-used types in ECMAScript. Although instances of `Object` don't have much functionality, they are ideally suited to storing and transmitting data around an application.

There are two ways to explicitly create an instance of `Object`. The first is to use the `new` operator with the `Object` constructor like this:



```
var person = new Object();
person.name = "Nicholas";
person.age = 29;
```

ObjectTypeExample01.htm

The other way is to use *object literal* notation. Object literal notation is a shorthand form of object definition designed to simplify creating an object with numerous properties. For example, the following defines the same `person` object from the previous example using object literal notation:

```
var person = {
    name : "Nicholas",
    age : 29
};
```

ObjectTypeExample02.htm

In this example, the left curly brace (`{`) signifies the beginning of an object literal, because it occurs in an *expression context*. An expression context in ECMAScript is a context in which a value (expression) is expected. Assignment operators indicate that a value is expected next, so the left curly brace indicates the beginning of an expression. The same curly brace, when appearing in a *statement context*, such as follows an `if` statement condition, indicates the beginning of a block statement.

Next, the `name` property is specified, followed by a colon, followed by the property's value. A comma is used to separate properties in an object literal, so there's a comma after the string "Nicholas" but not after the value 29, because `age` is the last property in the object. Including a comma after the last property causes an error in Internet Explorer 7 and earlier and Opera.

Property names can also be specified as strings or numbers when using object literal notation, such as in this example:

```
var person = {
    "name" : "Nicholas",
    "age" : 29,
    5: true
};
```

This example produces an object with a name property, an age property, and a property “5”. Note that numeric property names are automatically converted to strings.

It's also possible to create an object with only the default properties and methods using object literal notation by leaving the space between the curly braces empty, such as this:

```
var person = {};                                //same as new Object()
person.name = "Nicholas";
person.age = 29;
```

This example is equivalent to the first one in this section, though it looks a little strange. It's recommended to use object literal notation only when you're going to specify properties for readability.



When defining an object via object literal notation, the Object constructor is never actually called (Firefox 2 and earlier did call the Object constructor; this was changed in Firefox 3).

Though it's acceptable to use either method of creating Object instances, developers tend to favor object literal notation, because it requires less code and visually encapsulates all related data. In fact, object literals have become a preferred way of passing a large number of optional arguments to a function, such as in this example:



Available for download on Wrox.com

```
function displayInfo(args) {
    var output = "";
    if (typeof args.name == "string") {
        output += "Name: " + args.name + "\n";
    }
    if (typeof args.age == "number") {
        output += "Age: " + args.age + "\n";
    }
    alert(output);
}

displayInfo({
    name: "Nicholas",
    age: 29
});

displayInfo({
    name: "Greg"
});
```

[ObjectTypeExample04.htm](#)

Here, the function `displayInfo()` accepts a single argument named `args`. The argument may come in with a property called `name` or `age` or both or neither of those. The function is set up to test for the existence of each property using the `typeof` operator and then to construct a message to display based on availability. This function is then called twice, each time with different data specified in an object literal. The function works correctly in both cases.



This pattern for argument passing is best used when there is a large number of optional arguments that can be passed into the function. Generally speaking, named arguments are easier to work with but can get unwieldy when there are numerous optional arguments. The best approach is to use named arguments for those that are required and an object literal to encompass multiple optional arguments.

Although object properties are typically accessed using *dot notation*, which is common to many object-oriented languages, it's also possible to access properties via *bracket notation*. When you use bracket notation, a string containing the property name is placed between the brackets, as in this example:

```
alert(person["name"]);      // "Nicholas"
alert(person.name);        // "Nicholas"
```

Functionally, there is no difference between the two approaches. The main advantage of bracket notation is that it allows you to use variables for property access, such as in this example:

```
var propertyName = "name";
alert(person[propertyName]);    // "Nicholas"
```

You can also use bracket notation when the property name contains characters that would be either a syntax error or a keyword/reserved word. For example:

```
person["first name"] = "Nicholas";
```

Since the name `"first name"` contains a space, you can't use dot notation to access it. However, property names can contain nonalphanumeric characters, you just need to use bracket notation to access them.

Generally speaking, dot notation is preferred unless variables are necessary to access properties by name.

THE ARRAY TYPE

After the `Object` type, the `Array` type is probably the most used in ECMAScript. An ECMAScript array is very different from arrays in most other programming languages. As in other languages, ECMAScript arrays are ordered lists of data, but unlike in other languages, they can hold any type

of data in each slot. This means that it's possible to create an array that has a string in the first position, a number in the second, an object in the third, and so on. ECMAScript arrays are also dynamically sized, automatically growing to accommodate any data that is added to them.

Arrays can be created in two basic ways. The first is to use the `Array` constructor, as in this line:

```
var colors = new Array();
```

If you know the number of items that will be in the array, you can pass the count into the constructor, and the `length` property will automatically be created with that value. For example, the following creates an array with an initial `length` value of 20:

```
var colors = new Array(20);
```

The `Array` constructor can also be passed items that should be included in the array. The following creates an array with three string values:

```
var colors = new Array("red", "blue", "green");
```

An array can be created with a single value by passing it into the constructor. This gets a little bit tricky, because providing a single argument that is a number always creates an array with the given number of items, whereas an argument of any other type creates a one-item array that contains the specified value. Here's an example:



Available for
download on
Wrox.com

```
var colors = new Array(3);           //create an array with three items
var names = new Array("Greg");     //create an array with one item, the string "Greg"
```

[ArrayTypeExample01.htm](#)

It's possible to omit the `new` operator when using the `Array` constructor. It has the same result, as you can see here:

```
var colors = Array(3);           //create an array with three items
var names = Array("Greg");      //create an array with one item, the string "Greg"
```

The second way to create an array is by using *array literal* notation. An array literal is specified by using square brackets and placing a comma-separated list of items between them, as in this example:

```
var colors = ["red", "blue", "green"]; //creates an array with three strings
var names = [];                      //creates an empty array
var values = [1,2,];                 //AVOID! Creates an array with 2 or 3 items
var options = [,,,,,];               //AVOID! creates an array with 5 or 6 items
```

[ArrayTypeExample02.htm](#)

In this code, the first line creates an array with three string values. The second line creates an empty array by using empty square brackets. The third line shows the effects of leaving a comma after the last value in an array literal: in Internet Explorer 8 and earlier, `values` becomes a three-item

array containing the values 1, 2, and `undefined`; in all other browsers, `values` is a two-item array containing the values 1 and 2. This is due to a bug regarding array literals in the Internet Explorer implementation of ECMAScript through version 8 of the browser. Another instance of this bug is shown in the last line, which creates an array with either five (in Internet Explorer 9+, Firefox, Opera, Safari, and Chrome) or six (in Internet Explorer 8 and earlier) items. By omitting `values` between the commas, each item gets a value of `undefined`, which is logically the same as calling the `Array` constructor and passing in the number of items. However, because of the inconsistent implementation of early versions of Internet Explorer, using this syntax is strongly discouraged.



As with objects, the `Array` constructor isn't called when an array is created using array literal notation (except in Firefox prior to version 3).

To get and set array values, you use square brackets and provide the zero-based numeric index of the value, as shown here:

```
var colors = ["red", "blue", "green"];           //define an array of strings
alert(colors[0]);                            //display the first item
colors[2] = "black";                         //change the third item
colors[3] = "brown";                          //add a fourth item
```

The index provided within the square brackets indicates the value being accessed. If the index is less than the number of items in the array, then it will return the value stored in the corresponding item, as `colors[0]` displays "red" in this example. Setting a value works in the same way, replacing the value in the designated position. If a value is set to an index that is past the end of the array, as with `colors[3]` in this example, the array length is automatically expanded to be that index plus 1 (so the length becomes 4 in this example because the index being used is 3).

The number of items in an array is stored in the `length` property, which always returns 0 or more, as shown in the following example:

```
var colors = ["red", "blue", "green"];      //creates an array with three strings
var names = [];                           //creates an empty array

alert(colors.length);        //3
alert(names.length);         //0
```

A unique characteristic of `length` is that it's not read-only. By setting the `length` property, you can easily remove items from or add items to the end of the array. Consider this example:



```
var colors = ["red", "blue", "green"];      //creates an array with three strings
colors.length = 2;                        //changes the length to 2
alert(colors[2]);                         //outputs undefined
```

[ArrayFilterExample03.htm](#)

Here, the array `colors` starts out with three values. Setting the `length` to 2 removes the last item (in position 2), making it no longer accessible using `colors[2]`. If the `length` were set to a number greater than the number of items in the array, the new items would each get filled with the value of `undefined`, such as in this example:



Available for download on
Wrox.com

```
var colors = ["red", "blue", "green"];      //creates an array with three strings
colors.length = 4;
alert(colors[3]);           //undefined
```

[ArrayFilterExample04.htm](#)

This code sets the `length` of the `colors` array to 4 even though it contains only three items. Position 3 does not exist in the array, so trying to access its value results in the special value `undefined` being returned.

The `length` property can also be helpful in adding items to the end of an array, as in this example:

```
var colors = ["red", "blue", "green"];      //creates an array with three strings
colors[colors.length] = "black";          //add a color (position 3)
colors[colors.length] = "brown";         //add another color (position 4)
```

[ArrayFilterExample05.htm](#)

The last item in an array is always at position `length - 1`, so the next available open slot is at position `length`. Each time an item is added after the last one in the array, the `length` property is automatically updated to reflect the change. That means `colors[colors.length]` assigns a value to position 3 in the second line of this example and to position 4 in the last line. The new length is automatically calculated when an item is placed into a position that's outside of the current array size, which is done by adding 1 to the position, as in this example:

```
var colors = ["red", "blue", "green"];      //creates an array with three strings
colors[99] = "black";                      //add a color (position 99)
alert(colors.length);   //100
```

[ArrayFilterExample06.htm](#)

In this code, the `colors` array has a value inserted into position 99, resulting in a new `length` of 100 ($99 + 1$). Each of the other items, positions 3 through 98, doesn't actually exist and so returns `undefined` when accessed.



Arrays can contain a maximum of 4,294,967,295 items, which should be plenty for almost all programming needs. If you try to add more than that number, an exception occurs. Trying to create an array with an initial size approaching this maximum may cause a long-running script error.

Detecting Arrays

Ever since ECMAScript 3 was defined, one of the classic problems has been truly determining whether a given object is an array. When dealing with a single web page, and therefore a single global scope, the `instanceof` operator works well:

```
if (value instanceof Array){
    //do something on the array
}
```

The one problem with `instanceof` is that it assumes a single global execution context. If you are dealing with multiple frames in a web page, you're really dealing with two distinct global execution contexts and therefore two versions of the `Array` constructor. If you were to pass an array from one frame into a second frame, that array has a different constructor function than an array created natively in the second frame.

To work around this problem, ECMAScript 5 introduced the `Array.isArray()` method. The purpose of this method is to definitively determine if a given value is an array regardless of the global execution context in which it was created. Example usage:

```
if (Array.isArray(value)){
    //do something on the array
}
```

Internet Explorer 9+, Firefox 4+, Safari 5+, Opera 10.5+, and Chrome have all implemented `Array.isArray()`. For definitive array detection in browsers that haven't yet implemented this method, see the section titled "Safe Type Detection" in Chapter 22.

Conversion Methods

As mentioned previously, all objects have `toLocaleString()`, `toString()`, and `valueOf()` methods. The `toString()` and `valueOf()` methods return the same value when called on an array. The result is a comma-separated string that contains the string equivalents of each value in the array, which is to say that each item has its `toString()` method called to create the final string. Take a look at this example:



```
var colors = ["red", "blue", "green"];      //creates an array with three strings
alert(colors.toString());      //red,blue,green
alert(colors.valueOf());      //red,blue,green
alert(colors);                //red,blue,green
```

[ArrayFilterExample07.htm](#)

In this code, the `toString()` and `valueOf()` methods are first called explicitly to return the string representation of the array, which combines the strings, separating them by commas. The last line passes the array directly into `alert()`. Because `alert()` expects a string, it calls `toString()` behind the scenes to get the same result as when `toString()` is called directly.

The `toLocaleString()` method may end up returning the same value as `toString()` and `valueOf()`, but not always. When `toLocaleString()` is called on an array, it creates a comma-delimited string of the array values. The only difference between this and the two other methods is that `toLocaleString()` calls each item's `toLocaleString()` instead of `toString()` to get its string value. Consider the following example:



```
var person1 = {
    toLocaleString : function () {
        return "Nikolaos";
    },
    toString : function() {
        return "Nicholas";
    }
};

var person2 = {
    toLocaleString : function () {
        return "Grigorios";
    },
    toString : function() {
        return "Greg";
    }
};

var people = [person1, person2];
alert(people); //Nicholas, Greg
alert(people.toString()); //Nicholas, Greg
alert(people.toLocaleString()); //Nikolaos, Grigorios
```

[ArrayTypeExample08.htm](#)

Here, two objects are defined, `person1` and `person2`. Each object defines both a `toString()` method and a `toLocaleString()` method that return different values. An array, `people`, is created to contain both objects. When passed into `alert()`, the output is "Nicholas, Greg", because the `toString()` method is called on each item in the array (the same as when `toString()` is called explicitly on the next line). When `toLocaleString()` is called on the array, the result is "Nikolaos, Grigorios", because this calls `toLocaleString()` on each array item.

The inherited methods `toLocaleString()`, `toString()`, and `valueOf()` each return the array items as a comma-separated string. It's possible to construct a string with a different separator using the `join()` method. The `join()` method accepts one argument, which is the string separator to use, and returns a string containing all items. Consider this example:

```
var colors = ["red", "green", "blue"];
alert(colors.join(", ")); //red, green, blue
alert(colors.join("||")); //red||green||blue
```

[ArrayTypeJoinExample01.htm](#)

Here, the `join()` method is used on the `colors` array to duplicate the output of `toString()`. By passing in a comma, the result is a comma-separated list of values. On the last line, double pipes are passed in, resulting in the string "red||green||blue". If no value or `undefined` is passed into the `join()` method, then a comma is used as the separator. Internet Explorer 7 and earlier incorrectly use the string "`undefined`" as the separator.



If an item in the array is null or undefined, it is represented by an empty string in the result of `join()`, `toLocaleString()`, `toString()`, and `valueOf()`.

Stack Methods

One of the interesting things about ECMAScript arrays is that they provide a method to make an array behave like other data structures. An array object can act just like a stack, which is one of a group of data structures that restrict the insertion and removal of items. A stack is referred to as a *last-in-first-out* (LIFO) structure, meaning that the most recently added item is the first one removed. The insertion (called a *push*) and removal (called a *pop*) of items in a stack occur at only one point: the top of the stack. ECMAScript arrays provide `push()` and `pop()` specifically to allow stack-like behavior.

The `push()` method accepts any number of arguments and adds them to the end of the array, returning the array's new length. The `pop()` method, on the other hand, removes the last item in the array, decrements the array's `length`, and returns that item. Consider this example:



```
var colors = new Array();                                //create an array
var count = colors.push("red", "green");                //push two items
alert(count);  //2

count = colors.push("black");                           //push another item on
alert(count);  //3

var item = colors.pop();                               //get the last item
alert(item);  //"black"
alert(colors.length); //2
```

ArrayTypeExample09.htm

In this code, an array is created for use as a stack (note that there's no special code required to make this work; `push()` and `pop()` are default methods on arrays). First, two strings are pushed onto the end of the array using `push()`, and the result is stored in the variable `count` (which gets the value of 2). Then, another value is pushed on, and the result is once again stored in `count`. Because there are now three items in the array, `push()` returns 3. When `pop()` is called, it returns the last item in the array, which is the string "black". The array then has only two items left.

The stack methods may be used in combination with all of the other array methods as well, as in this example:



```
var colors = ["red", "blue"];
colors.push("brown");                                //add another item
colors[3] = "black";                                 //add an item
alert(colors.length); //4

var item = colors.pop();                            //get the last item
alert(item); // "black"
```

[ArrayTypeExample10.htm](#)

Here, an array is initialized with two values. A third value is added via `push()`, and a fourth is added by direct assignment into position 3. When `pop()` is called, it returns the string "black", which was the last value added to the array.

Queue Methods

Just as stacks restrict access in a LIFO data structure, queues restrict access in a *first-in-first-out* (FIFO) data structure. A queue adds items to the end of a list and retrieves items from the front of the list. Because the `push()` method adds items to the end of an array, all that is needed to emulate a queue is a method to retrieve the first item in the array. The array method for this is called `shift()`, which removes the first item in the array and returns it, decrementing the length of the array by one. Using `shift()` in combination with `push()` allows arrays to be used as queues:

```
var colors = new Array();                           //create an array
var count = colors.push("red", "green");           //push two items
alert(count); //2

count = colors.push("black");                      //push another item on
alert(count); //3

var item = colors.shift();                         //get the first item
alert(item); // "red"
alert(colors.length); //2
```

[ArrayTypeExample11.htm](#)

This example creates an array of three colors using the `push()` method. The highlighted line shows the `shift()` method being used to retrieve the first item in the array, which is "red". With that item removed, "green" is moved into the first position and "black" is moved into the second, leaving the array with two items.

ECMAScript also provides an `unshift()` method for arrays. As the name indicates, `unshift()` does the opposite of `shift()`: it adds any number of items to the front of an array and returns the new array length. By using `unshift()` in combination with `pop()`, it's possible to emulate a queue in the opposite direction, where new values are added to the front of the array and values are retrieved off the back, as in this example:

```
var colors = new Array();                           //create an array
var count = colors.unshift("red", "green");        //push two items
alert(count); //2

count = colors.unshift("black");                  //push another item on
```

```

alert(count); //3

var item = colors.pop(); //get the first item
alert(item); // "green"
alert(colors.length); //2

```

[ArrayTypeExample12.htm](#)

In this code, an array is created and then populated by using `unshift()`. First "red" and "green" are added to the array, and then "black" is added, resulting in an order of "black", "red", "green". When `pop()` is called, it removes the last item, "green", and returns it.



Internet Explorer 7 and earlier always return undefined, instead of the new length of the array, for `unshift()`. Internet Explorer 8 returns the length correctly when not in compatibility mode.

Reordering Methods

Two methods deal directly with the reordering of items already in the array: `reverse()` and `sort()`. As one might expect, the `reverse()` method simply reverses the order of items in an array. Take this code for example:



Available for download on
Wrox.com

```

var values = [1, 2, 3, 4, 5];
values.reverse();
alert(values); //5,4,3,2,1

```

[ArrayTypeExample13.htm](#)

Here, the array's `values` are initially set to 1, 2, 3, 4, and 5, in that order. Calling `reverse()` on the array reverses the order to 5, 4, 3, 2, 1. This method is fairly straightforward but doesn't provide much flexibility, which is where the `sort()` method comes in.

By default, the `sort()` method puts the items in ascending order — with the smallest value first and the largest value last. To do this, the `sort()` method calls the `String()` casting function on every item and then compares the strings to determine the correct order. This occurs even if all items in an array are numbers, as in this example:

```

var values = [0, 1, 5, 10, 15];
values.sort();
alert(values); //0,1,10,15,5

```

[ArrayTypeExample14.htm](#)

Even though the values in this example begin in correct numeric order, the `sort()` method changes that order based on their string equivalents. So even though 5 is less than 10, the string "10" comes

before "5" when doing a string comparison, so the array is updated accordingly. Clearly, this is not an optimal solution in many cases, so the `sort()` method allows you to pass in a *comparison function* that indicates which value should come before which.

A comparison function accepts two arguments and returns a negative number if the first argument should come before the second, a zero if the arguments are equal, or a positive number if the first argument should come after the second. Here's an example of a simple comparison function:



Available for
download on
Wrox.com

```
function compare(value1, value2) {
    if (value1 < value2) {
        return -1;
    } else if (value1 > value2) {
        return 1;
    } else {
        return 0;
    }
}
```

[ArrayTypeExample15.htm](#)

This comparison function works for most data types and can be used by passing it as an argument to the `sort()` method, as in the following example:

```
var values = [0, 1, 5, 10, 15];
values.sort(compare);
alert(values);      //0,1,5,10,15
```

When the comparison function is passed to the `sort()` method, the numbers remain in the correct order. Of course, the comparison function could produce results in descending order if you simply switch the return values like this:

```
function compare(value1, value2) {
    if (value1 < value2) {
        return 1;
    } else if (value1 > value2) {
        return -1;
    } else {
        return 0;
    }
}

var values = [0, 1, 5, 10, 15];
values.sort(compare);
alert(values);      //15,10,5,1,0
```

[ArrayTypeExample16.htm](#)

In this modified example, the comparison function returns 1 if the first value should come after the second and -1 if the first value should come before the second. Swapping these means the larger value will come first and the array will be sorted in descending order. Of course, if you just want to reverse the order of the items in the array, `reverse()` is a much faster alternative than sorting.



Both reverse() and sort() return a reference to the array on which they were applied.

A much simpler version of the comparison function can be used with numeric types, and objects whose `valueOf()` method returns numeric values (such as the `Date` object). In either case, you can simply subtract the second value from the first as shown here:

```
function compare(value1, value2) {
    return value2 - value1;
}
```

Because comparison functions work by returning a number less than zero, zero, or a number greater than zero, the subtraction operation handles all of the cases appropriately.

Manipulation Methods

There are various ways to work with the items already contained in an array. The `concat()` method, for instance, allows you to create a new array based on all of the items in the current array. This method begins by creating a copy of the array and then appending the method arguments to the end and returning the newly constructed array. When no arguments are passed in, `concat()` simply clones the array and returns it. If one or more arrays are passed in, `concat()` appends each item in these arrays to the end of the result. If the values are not arrays, they are simply appended to the end of the resulting array. Consider this example:



Available for
download on
[Wrox.com](#)

```
var colors = ["red", "green", "blue"];
var colors2 = colors.concat("yellow", ["black", "brown"]);

alert(colors);      //red,green,blue
alert(colors2);    //red,green,blue,yellow,black,brown
```

[ArrayTypeConcatExample01.htm](#)

This code begins with the `colors` array containing three values. The `concat()` method is called on `colors`, passing in the string `"yellow"` and an array containing `"black"` and `"brown"`. The result, stored in `colors2`, contains `"red"`, `"green"`, `"blue"`, `"yellow"`, `"black"`, and `"brown"`. The original array, `colors`, remains unchanged.

The next method, `slice()`, creates an array that contains one or more items already contained in an array. The `slice()` method may accept one or two arguments: the starting and stopping positions of the items to return. If only one argument is present, the method returns all items between that position and the end of the array. If there are two arguments, the method returns all items between the start position and the end position, not including the item in the end position. Keep in mind that this operation does not affect the original array in any way. Consider the following:



Available for
download on
Wrox.com

```
var colors = ["red", "green", "blue", "yellow", "purple"];
var colors2 = colors.slice(1);
var colors3 = colors.slice(1, 4);

alert(colors2); //green,blue,yellow,purple
alert(colors3); //green,blue,yellow
```

[ArrayTypeSliceExample01.htm](#)

In this example, the `colors` array starts out with five items. Calling `slice()` and passing in 1 yields an array with four items, omitting "red" because the operation began copying from position 1, which contains "green". The resulting `colors2` array contains "green", "blue", "yellow", and "purple". The `colors3` array is constructed by calling `slice()` and passing in 1 and 4, meaning that the method will begin copying from the item in position 1 and stop copying at the item in position 3. As a result, `colors3` contains "green", "blue", and "yellow".



If either the start or end position of `slice()` is a negative number, then the number is subtracted from the length of the array to determine the appropriate locations. For example, calling `slice(-2, -1)` on an array with five items is the same as calling `slice(3, 4)`. If the end position is smaller than the start, then an empty array is returned.

Perhaps the most powerful array method is `splice()`, which can be used in a variety of ways. The main purpose of `splice()` is to insert items into the middle of an array, but there are three distinct ways of using this method. They are as follows:

- **Deletion** — Any number of items can be deleted from the array by specifying just two arguments: the position of the first item to delete and the number of items to delete. For example, `splice(0, 2)` deletes the first two items.
- **Insertion** — Items can be inserted into a specific position by providing three or more arguments: the starting position, 0 (the number of items to delete), and the item to insert. Optionally, you can specify a fourth parameter, fifth parameter, or any number of other parameters to insert. For example, `splice(2, 0, "red", "green")` inserts the strings "red" and "green" into the array at position 2.
- **Replacement** — Items can be inserted into a specific position while simultaneously deleting items, if you specify three arguments: the starting position, the number of items to delete, and any number of items to insert. The number of items to insert doesn't have to match the number of items to delete. For example, `splice(2, 1, "red", "green")` deletes one item at position 2 and then inserts the strings "red" and "green" into the array at position 2.



Available for
download on
Wrox.com

```
var colors = ["red", "green", "blue"];
var removed = colors.splice(0,1); //remove the first item
alert(colors); //green,blue
alert(removed); //red - one item array

removed = colors.splice(1, 0, "yellow", "orange"); //insert two items at position 1
alert(colors); //green,yellow,orange,blue
alert(removed); //empty array

removed = colors.splice(1, 1, "red", "purple"); //insert two values, remove one
alert(colors); //green,red,purple,orange,blue
alert(removed); //yellow - one item array
```

[ArrayTypeSpliceExample01.htm](#)

This example begins with the `colors` array containing three items. When `splice` is called the first time, it simply removes the first item, leaving `colors` with the items "green" and "blue". The second time `splice()` is called, it inserts two items at position 1, resulting in `colors` containing "green", "yellow", "orange", and "blue". No items are removed at this point, so an empty array is returned. The last time `splice()` is called, it removes one item, beginning in position 1, and inserts "red" and "purple". After all of this code has been executed, the `colors` array contains "green", "red", "purple", "orange", and "blue".

Location Methods

ECMAScript 5 adds two item location methods to array instances: `indexOf()` and `lastIndexOf()`. Each of these methods accepts two arguments: the item to look for and an optional index from which to start looking. The `indexOf()` method starts searching from the front of the array (item 0) and continues to the back, whereas `lastIndexOf()` starts from the last item in the array and continues to the front.

The methods each return the position of the item in the array or `-1` if the item isn't in the array. An identity comparison is used when comparing the first argument to each item in the array, meaning that the items must be strictly equal as if compared using `==`. Here are some examples of this usage:

```
var numbers = [1,2,3,4,5,4,3,2,1];

alert(numbers.indexOf(4)); //3
alert(numbers.lastIndexOf(4)); //5

alert(numbers.indexOf(4, 4)); //5
alert(numbers.lastIndexOf(4, 4)); //3

var person = { name: "Nicholas" };
var people = [{ name: "Nicholas" }];
```

```
var morePeople = [person];
alert(people.indexOf(person));      // -1
alert(morePeople.indexOf(person));  // 0
```

[ArrayIndexOfExample01.htm](#)

The `indexOf()` and `lastIndexOf()` methods make it trivial to locate specific items inside of an array and are supported in Internet Explorer 9+, Firefox 2+, Safari 3+, Opera 9.5+, and Chrome.

Iterative Methods

ECMAScript 5 defines five iterative methods for arrays. Each of the methods accepts two arguments: a function to run on each item and an optional scope object in which to run the function (affecting the value of `this`). The function passed into one of these methods will receive three arguments: the array item value, the position of the item in the array, and the array object itself. Depending on the method, the results of this function's execution may or may not affect the method's return value. The iterative methods are as follows:

- `every()` — Runs the given function on every item in the array and returns `true` if the function returns `true` for every item.
- `filter()` — Runs the given function on every item in the array and returns an array of all items for which the function returns `true`.
- `forEach()` — Runs the given function on every item in the array. This method has no return value.
- `map()` — Runs the given function on every item in the array and returns the result of each function call in an array.
- `some()` — Runs the given function on every item in the array and returns `true` if the function returns `true` for any one item.

These methods do not change the values contained in the array.

Of these methods, the two most similar are `every()` and `some()`, which both query the array for items matching some criteria. For `every()`, the passed-in function must return `true` for every item in order for the method to return `true`; otherwise, it returns `false`. The `some()` method, on the other hand, returns `true` if even one of the items causes the passed-in function to return `true`. Here is an example:



Available for
download on
Wrox.com

```
var numbers = [1,2,3,4,5,4,3,2,1];
var everyResult = numbers.every(function(item, index, array) {
    return (item > 2);
});
alert(everyResult);           // false

var someResult = numbers.some(function(item, index, array) {
```

```

        return (item > 2);
});

alert(someResult);      //true

```

[ArrayEveryAndSomeExample01.htm](#)

This code calls both `every()` and `some()` with a function that returns `true` if the given item is greater than 2. For `every()`, the result is `false`, because only some of the items fit the criteria. For `some()`, the result is `true`, because at least one of the items is greater than 2.

The next method is `filter()`, which uses the given function to determine if an item should be included in the array that it returns. For example, to return an array of all numbers greater than 2, the following code can be used:



Available for download on
Wrox.com

```

var numbers = [1,2,3,4,5,4,3,2,1];

var filterResult = numbers.filter(function(item, index, array){
    return (item > 2);
});

alert(filterResult);    //[3,4,5,4,3]

```

[ArrayFilterExample01.htm](#)

Here, an array containing the values 3, 4, 5, 4, and 3 is created and returned by the call to `filter()`, because the passed-in function returns `true` for each of those items. This method is very helpful when querying an array for all items matching some criteria.

The `map()` method also returns an array. Each item in the array is the result of running the passed-in function on the original array item in the same location. For example, you can multiply every number in an array by two and are returned an array of those numbers, as shown here:

```

var numbers = [1,2,3,4,5,4,3,2,1];

var mapResult = numbers.map(function(item, index, array){
    return item * 2;
});

alert(mapResult);    //[2,4,6,8,10,8,6,4,2]

```

[ArrayMapExample01.htm](#)

The code in this example returns an array containing the result of multiplying each number by two. This method is helpful when creating arrays whose items correspond to one another.

The last method is `forEach()`, which simply runs the given function on every item in an array. There is no return value and is essentially the same as iterating over an array using a `for` loop. Here's an example:

```
var numbers = [1,2,3,4,5,4,3,2,1];
numbers.forEach(function(item, index, array) {
    //do something here
});
```

All of these array methods ease the processing of arrays by performing a number of different operations. The iterative methods are supported in Internet Explorer 9+, Firefox 2+, Safari 3+, Opera 9.5+, and Chrome.

Reduction Methods

ECMAScript 5 also introduced two reduction methods for arrays: `reduce()` and `reduceRight()`. Both methods iterate over all items in the array and build up a value that is ultimately returned. The `reduce()` method does this starting at the first item and traveling toward the last, whereas `reduceRight()` starts at the last and travels toward the first.

Both methods accept two arguments: a function to call on each item and an optional initial value upon which the reduction is based. The function passed into `reduce()` or `reduceRight()` accepts four arguments: the previous value, the current value, the item's index, and the array object. Any value returned from the function is automatically passed in as the first argument for the next item. The first iteration occurs on the second item in the array, so the first argument is the first item in the array and the second argument is the second item in the array.

You can use the `reduce()` method to perform operations such as adding all numbers in an array. Here's an example:



```
var values = [1,2,3,4,5];
var sum = values.reduce(function(prev, cur, index, array) {
    return prev + cur;
});
alert(sum); //15
```

[ArrayReductionExample01.htm](#)

The first time the callback function is executed, `prev` is 1 and `cur` is 2. The second time, `prev` is 3 (the result of adding 1 and 2), and `cur` is 3 (the third item in the array). This sequence continues until all items have been visited and the result is returned.

The `reduceRight()` method works in the same way, just in the opposite direction. Consider the following example:

```
var values = [1,2,3,4,5];
var sum = values.reduceRight(function(prev, cur, index, array) {
    return prev + cur;
});
alert(sum); //15
```

In this version of the code, `prev` is 5 and `cur` is 4 the first time the callback function is executed. The result is the same, of course, since the operation is simple addition.

The decision to use `reduce()` or `reduceRight()` depends solely on the direction in which the items in the array should be visited. They are exactly equal in every other way.

The reduction methods are supported in Internet Explorer 9+, Firefox 3+, Safari 4+, Opera 10.5, and Chrome.

THE DATE TYPE

The ECMAScript `Date` type is based on an early version of `java.util.Date` from Java. As such, the `Date` type stores dates as the number of milliseconds that have passed since midnight on January 1, 1970 UTC (Universal Time Code). Using this data storage format, the `Date` type can accurately represent dates 285,616 years before or after January 1, 1970.

To create a date object, use the `new` operator along with the `Date` constructor, like this:



```
var now = new Date();
```

Available for
download on
Wrox.com

[DateTypeExample01.htm](#)

When the `Date` constructor is used without any arguments, the created object is assigned the current date and time. To create a date based on another date or time, you must pass in the millisecond representation of the date (the number of milliseconds after midnight, January 1, 1970 UTC). To aid in this process, ECMAScript provides two methods: `Date.parse()` and `Date.UTC()`.

The `Date.parse()` method accepts a string argument representing a date. It attempts to convert the string into a millisecond representation of a date. ECMA-262 fifth edition defines which date formats `Date.parse()` should support, filling in a void left by the third edition. All implementations must now support the following date formats:

- month/date/year (such as 6/13/2004)
- month_name date, year (such as January 12, 2004)
- day_of_week month_name date year hours:minutes:seconds time_zone (such as Tue May 25 2004 00:00:00 GMT-0700)
- ISO 8601 extended format YYYY-MM-DDTHH:mm:ss.sssZ (such as 2004-05-25T00:00:00). This works only in ECMAScript 5-compliant implementations.

For instance, to create a date object for May 25, 2004, you can use the following code:

```
var someDate = new Date(Date.parse("May 25, 2004"));
```

[DateTypeExample01.htm](#)

If the string passed into `Date.parse()` doesn't represent a date, then it returns `Nan`. The `Date` constructor will call `Date.parse()` behind the scenes if a string is passed in directly, meaning that the following code is identical to the previous example:

```
var someDate = new Date("May 25, 2004");
```

This code produces the same result as the previous example.



There are a lot of quirks surrounding the Date type and its implementation in various browsers. There is a tendency to replace out-of-range values with the current value to produce an output, so when trying to parse "January 32, 2007", some browsers will interpret it as "February 1, 2007", whereas Opera tends to insert the current day of the current month, returning "January current_day, 2007". This means running the code on September 21 returns "January 21, 2007".

The `Date.UTC()` method also returns the millisecond representation of a date but constructs that value using different information than `Date.parse()`. The arguments for `Date.UTC()` are the year, the zero-based month (January is 0, February is 1, and so on), the day of the month (1 through 31), and the hours (0 through 23), minutes, seconds, and milliseconds of the time. Of these arguments, only the first two (year and month) are required. If the day of the month isn't supplied, it's assumed to be 1, while all other omitted arguments are assumed to be 0. Here are two examples of `Date.UTC()` in action:



Available for download on
Wrox.com

```
//January 1, 2000 at midnight GMT
var y2k = new Date(Date.UTC(2000, 0));

//May 5, 2005 at 5:55:55 PM GMT
var allFives = new Date(Date.UTC(2005, 4, 5, 17, 55, 55));
```

[DateTypeUTCExample01.htm](#)

Two dates are created in this example. The first date is for midnight (GMT) on January 1, 2000, which is represented by the year 2000 and the month 0 (which is January). Because the other arguments are filled in (the day of the month as 1 and everything else as 0), the result is the first day of the month at midnight. The second date represents May 5, 2005, at 5:55:55 PM GMT. Even though the date and time contain only fives, creating this date requires some different numbers: the month must be set to 4 because months are zero-based, and the hour must be set to 17 because hours are represented as 0 through 23. The rest of the arguments are as expected.

As with `Date.parse()`, `Date.UTC()` is mimicked by the `Date` constructor but with one major difference: the date and time created are in the local time zone, not in GMT. However, the `Date` constructor takes the same arguments as `Date.UTC()`, so if the first argument is a number, the constructor assumes that it is the year of a date, the second argument is the month, and so on. The preceding example can then be rewritten as this:

```
//January 1, 2000 at midnight in local time
var y2k = new Date(2000, 0);

//May 5, 2005 at 5:55:55 PM local time
var allFives = new Date(2005, 4, 5, 17, 55, 55);
```

[DateTypeConstructorExample01.htm](#)

This code creates the same two dates as the previous example, but this time both dates are in the local time zone as determined by the system settings.

ECMAScript 5 adds `Date.now()`, which returns the millisecond representation of the date and time at which the method is executed. This method makes it trivial to use `Date` objects for code profiling, such as:

```
//get start time
var start = Date.now();

//call a function
doSomething();

//get stop time
var stop = Date.now(),
    result = stop - start;
```

The `Date.now()` method has been implemented in Internet Explorer 9+, Firefox 3+, Safari 3+, Opera 10.5, and Chrome. For browsers that don't yet support this method, you can simulate the same behavior by using the `+` operator to convert a `Date` object into a number:

```
//get start time
var start = +new Date();

//call a function
doSomething();

//get stop time
var stop = +new Date(),
    result = stop - start;
```

Inherited Methods

As with the other reference types, the `Date` type overrides `toLocaleString()`, `toString()`, and `valueOf()`, though unlike the previous types, each method returns something different. The `Date` type's `toLocaleString()` method returns the date and time in a format appropriate for the locale in which the browser is being run. This often means that the format includes AM or PM for the time and doesn't include any time-zone information (the exact format varies from browser to browser). The `toString()` method typically returns the date and time with time-zone information, and the time is typically indicated in 24-hour notation (hours ranging from 0 to 23). The following list displays the formats that various browsers use for `toLocaleString()` and `toString()` when representing the date/time of February 1, 2007, at midnight PST (Pacific Standard Time) in the “en-US” locale:

Internet Explorer 8

```
toLocaleString() - Thursday, February 01, 2007 12:00:00 AM
toString() - Thu Feb 1 00:00:00 PST 2007
```

Firefox 3.5

```
toLocaleString() - Thursday, February 01, 2007 12:00:00 AM
toString() - Thu Feb 01 2007 00:00:00 GMT-0800 (Pacific Standard Time)
```

Safari 4

```
toLocaleString() - Thursday, February 01, 2007 00:00:00
toString() - Thu Feb 01 2007 00:00:00 GMT-0800 (Pacific Standard Time)
```

Chrome 4

```
toLocaleString() - Thu Feb 01 2007 00:00:00 GMT-0800 (Pacific Standard Time)
toString() - Thu Feb 01 2007 00:00:00 GMT-0800 (Pacific Standard Time)
```

Opera 10

```
toLocaleString() - 2/1/2007 12:00:00 AM
toString() - Thu, 01 Feb 2007 00:00:00 GMT-0800
```

As you can see, there are some pretty significant differences between the formats that browsers return for each method. These differences mean `toLocaleString()` and `toString()` are really useful only for debugging purposes, not for display purposes.

The `valueOf()` method for the `Date` type doesn't return a string at all, because it is overridden to return the milliseconds representation of the date so that operators (such as less-than and greater-than) will work appropriately for date values. Consider this example:



```
var date1 = new Date(2007, 0, 1);           // "January 1, 2007"
var date2 = new Date(2007, 1, 1);           // "February 1, 2007"

alert(date1 < date2); //true
alert(date1 > date2); //false
```

[DateTypeValueOfExample01.htm](#)

The date January 1, 2007, comes before February 1, 2007, so it would make sense to say that the former is less than the latter. Because the milliseconds representation of January 1, 2007, is less than that of February 1, 2007, the less-than operator returns `true` when the dates are compared, providing an easy way to determine the order of dates.

Date-Formatting Methods

There are several `Date` type methods used specifically to format the date as a string. They are as follows:

- `toDateString()` — Displays the date's day of the week, month, day of the month, and year in an implementation-specific format.

- `toTimeString()` — Displays the date's hours, minutes, seconds, and time zone in an implementation-specific format.
- `toLocaleDateString()` — Displays the date's day of the week, month, day of the month, and year in an implementation- and locale-specific format.
- `toLocaleTimeString()` — Displays the date's hours, minutes, and seconds in an implementation-specific format.
- `toUTCString()` — Displays the complete UTC date in an implementation-specific format.

The output of these methods, as with `toLocaleString()` and `toString()`, varies widely from browser to browser and therefore can't be employed in a user interface for consistent display of a date.



There is also a method called `toGMTString()`, which is equivalent to `toUTCString()` and is provided for backwards compatibility. However, the specification recommends that new code use `toUTCString()` exclusively.

Date/Time Component Methods

The remaining methods of the `Date` type (listed in the following table) deal directly with getting and setting specific parts of the date value. Note that references to a UTC date mean the date value when interpreted without a time-zone offset (the date when converted to GMT).

METHOD	DESCRIPTION
<code>getTime()</code>	Returns the milliseconds representation of the date; same as <code>valueOf()</code> .
<code>setTime(milliseconds)</code>	Sets the milliseconds representation of the date, thus changing the entire date.
<code>getFullYear()</code>	Returns the four-digit year (2007 instead of just 07).
<code>getUTCFullYear()</code>	Returns the four-digit year of the UTC date value.
<code>setFullYear(year)</code>	Sets the year of the date. The year must be given with four digits (2007 instead of just 07).
<code>setUTCFullYear(year)</code>	Sets the year of the UTC date. The year must be given with four digits (2007 instead of just 07).
<code>getMonth()</code>	Returns the month of the date, where 0 represents January and 11 represents December.
<code>getUTCMonth()</code>	Returns the month of the UTC date, where 0 represents January and 11 represents December.

METHOD	DESCRIPTION
<code>setMonth(month)</code>	Sets the month of the date, which is any number 0 or greater. Numbers greater than 11 add years.
<code>setUTCMonth(month)</code>	Sets the month of the UTC date, which is any number 0 or greater. Numbers greater than 11 add years.
<code>getDate()</code>	Returns the day of the month (1 through 31) for the date.
<code>getUTCDate()</code>	Returns the day of the month (1 through 31) for the UTC date.
<code> setDate(date)</code>	Sets the day of the month for the date. If the date is greater than the number of days in the month, the month value also gets increased.
<code>setUTCDate(date)</code>	Sets the day of the month for the UTC date. If the date is greater than the number of days in the month, the month value also gets increased.
<code>getDay()</code>	Returns the date's day of the week as a number (where 0 represents Sunday and 6 represents Saturday).
<code>getUTCDay()</code>	Returns the UTC date's day of the week as a number (where 0 represents Sunday and 6 represents Saturday).
<code>getHours()</code>	Returns the date's hours as a number between 0 and 23.
<code>getUTCHours()</code>	Returns the UTC date's hours as a number between 0 and 23.
<code>setHours(hours)</code>	Sets the date's hours. Setting the hours to a number greater than 23 also increments the day of the month.
<code>setUTCHours(hours)</code>	Sets the UTC date's hours. Setting the hours to a number greater than 23 also increments the day of the month.
<code>getMinutes()</code>	Returns the date's minutes as a number between 0 and 59.
<code>getUTCMinutes()</code>	Returns the UTC date's minutes as a number between 0 and 59.
<code>setMinutes(minutes)</code>	Sets the date's minutes. Setting the minutes to a number greater than 59 also increments the hour.
<code>setUTCMinutes(minutes)</code>	Sets the UTC date's minutes. Setting the minutes to a number greater than 59 also increments the hour.
<code>getSeconds()</code>	Returns the date's seconds as a number between 0 and 59.
<code>getUTCSeconds()</code>	Returns the UTC date's seconds as a number between 0 and 59.
<code>setSeconds(seconds)</code>	Sets the date's seconds. Setting the seconds to a number greater than 59 also increments the minutes.

continues

(continued)

METHOD	DESCRIPTION
<code>setUTCSeconds(seconds)</code>	Sets the UTC date's seconds. Setting the seconds to a number greater than 59 also increments the minutes.
<code>getMilliseconds()</code>	Returns the date's milliseconds.
<code>getUTCMilliseconds()</code>	Returns the UTC date's milliseconds.
<code>setMilliseconds(milliseconds)</code>	Sets the date's milliseconds.
<code>setUTCMilliseconds(milliseconds)</code>	Sets the UTC date's milliseconds.
<code>getTimezoneOffset()</code>	Returns the number of minutes that the local time zone is offset from UTC. For example, Eastern Standard Time returns 300. This value changes when an area goes into Daylight Saving Time.

THE REGEXP TYPE

ECMAScript supports regular expressions through the `RegExp` type. Regular expressions are easy to create using syntax similar to Perl, as shown here:

```
var expression = /pattern	flags;
```

The pattern part of the expression can be any simple or complicated regular expression, including character classes, quantifiers, grouping, lookaheads, and backreferences. Each expression can have zero or more flags indicating how the expression should behave. Three supported flags represent matching modes, as follows:

- `g` — Indicates global mode, meaning the pattern will be applied to all of the string instead of stopping after the first match is found.
- `i` — Indicates case-insensitive mode, meaning the case of the pattern and the string are ignored when determining matches.
- `m` — Indicates multiline mode, meaning the pattern will continue looking for matches after reaching the end of one line of text.

A regular expression is created using a combination of a pattern and these flags to produce different results, as in this example:

```
/*
 * Match all instances of "at" in a string.
 */
var pattern1 = /at/g;
/*
```

```

/*
 * Match the first instance of "bat" or "cat", regardless of case.
 */
var pattern2 = /[bc]at/i;

/*
 * Match all three-character combinations ending with "at", regardless of case.
 */
var pattern3 = /.at/gi;

```

As with regular expressions in other languages, all *metacharacters* must be escaped when used as part of the pattern. The metacharacters are as follows:

```
( [ { \ ^ $ | ) ] } ? * + .
```

Each metacharacter has one or more uses in regular-expression syntax and so must be escaped by a backslash when you want to match the character in a string. Here are some examples:

```

/*
 * Match the first instance of "bat" or "cat", regardless of case.
 */
var pattern1 = /[bc]at/i;

/*
 * Match the first instance of "[bc]at", regardless of case.
 */
var pattern2 = /\[bc\]at/i;

/*
 * Match all three-character combinations ending with "at", regardless of case.
 */
var pattern3 = /.at/gi;

/*
 * Match all instances of ".at", regardless of case.
 */
var pattern4 = /\.at/gi;

```

In this code, pattern1 matches all instances of "bat" or "cat", regardless of case. To match "[bc]at" directly, both square brackets need to be escaped with a backslash, as in pattern2. In pattern3, the dot indicates that any character can precede "at" to be a match. If you want to match ".at", then the dot needs to be escaped, as in pattern4.

The preceding examples all define regular expressions using the literal form. Regular expressions can also be created by using the `RegExp` constructor, which accepts two arguments: a string pattern to match and an optional string of flags to apply. Any regular expression that can be defined using literal syntax can also be defined using the constructor, as in this example:

```

/*
 * Match the first instance of "bat" or "cat", regardless of case.
 */
var pattern1 = /[bc]at/i;

```

```
/*
 * Same as pattern1, just using the constructor.
 */
var pattern2 = new RegExp("[bc]at", "i");
```

Here, `pattern1` and `pattern2` define equivalent regular expressions. Note that both arguments of the `RegExp` constructor are strings (regular-expression literals should not be passed into the `RegExp` constructor). Because the pattern argument of the `RegExp` constructor is a string, there are some instances in which you need to double-escape characters. All metacharacters must be double-escaped, as most characters that are already escaped, such as `\n` (the `\` character, which is normally escaped in strings as `\\\` becomes `\\\\\\` when used in a regular-expression string). The following table shows some patterns in their literal form and the equivalent string that would be necessary to use the `RegExp` constructor.

LITERAL PATTERN	STRING EQUIVALENT
<code>/\\ [bc\\]at/</code>	<code>"\\\\ [bc\\\\]at"</code>
<code>/\\.at/</code>	<code>"\\\\.at"</code>
<code>/name\\/age/</code>	<code>"name\\\\/age"</code>
<code>/\\d.\\d{1,2}/</code>	<code>"\\\\d.\\\\d{1,2}"</code>
<code>/\\w\\hello\\\\123/</code>	<code>"\\\\w\\\\\\hello\\\\\\123"</code>

Keep in mind that creating a regular expression using a literal is not exactly the same as creating a regular expression using the `RegExp` constructor. In ECMAScript 3, regular-expression literals always share the same `RegExp` instance, while creating a new `RegExp` via constructor always results in a new instance. Consider the following:

```
var re = null,
    i;

for (i=0; i < 10; i++){
    re = /cat/g;
    re.test("catastrope");
}

for (i=0; i < 10; i++){
    re = new RegExp("cat", "g");
    re.test("catastrophe");
}
```

In the first loop, there is only one instance of `RegExp` created for `/cat/`, even though it is specified in the body of the loop. Instance properties (mentioned in the next section) are not reset, so calling `test()` fails every other time through the loop. This happens because the "cat" is found in the first call to `test()`, but the second call begins its search from index 3 (the end of the last match) and can't find it. Since the end of the string is found, the subsequent call to `test()` starts at the beginning again.

The second loop uses the `RegExp` constructor to create the regular expression each time through the loop. Each call to `test()` returns `true` since a new instance of `RegExp` is created for each iteration.

ECMAScript 5 clarifies the behavior of regular-expression literals by explicitly stating that regular-expression literals must create new instances of `RegExp` as if the `RegExp` constructor were called directly. This change was made in Internet Explorer 9+, Firefox 4+, and Chrome.

RegExp Instance Properties

Each instance of `RegExp` has the following properties that allow you to get information about the pattern:

- `global` — A Boolean value indicating whether the `g` flag has been set.
- `ignoreCase` — A Boolean value indicating whether the `i` flag has been set.
- `lastIndex` — An integer indicating the character position where the next match will be attempted in the source string. This value always begins as 0.
- — A Boolean value indicating whether the `m` flag has been set.
- `source` — The string source of the regular expression. This is always returned as if specified in literal form (without opening and closing slashes) rather than a string pattern as passed into the constructor.

These properties are helpful in identifying aspects of a regular expression; however, they typically don't have much use, because the information is available in the pattern declaration. Here's an example:



Available for
download on
Wrox.com

```
var pattern1 = /\[bc\]at/i;
alert(pattern1.global);      //false
alert(pattern1.ignoreCase); //true
alert(pattern1.multiline);  //false
alert(pattern1.lastIndex); //0
alert(pattern1.source);    //"\[bc\]at"

var pattern2 = new RegExp("\\\[bc\\]at", "i");

alert(pattern2.global);      //false
alert(pattern2.ignoreCase); //true
alert(pattern2.multiline);  //false
alert(pattern2.lastIndex); //0
alert(pattern2.source);    //"\[bc\]at"
```

[RegExpInstancePropertiesExample01.htm](#)

Note that the `source` properties of each pattern are equivalent even though the first pattern is in literal form and the second uses the `RegExp` constructor. The `source` property normalizes the string into the form you'd use in a literal.

RegExp Instance Methods

The primary method of a `RegExp` object is `exec()`, which is intended for use with capturing groups. This method accepts a single argument, which is the string on which to apply the pattern, and returns an array of information about the first match or `null` if no match was found. The returned array, though an instance of `Array`, contains two additional properties: `index`, which is the location in the string where the pattern was matched, and `input`, which is the string that the expression was run against. In the array, the first item is the string that matches the entire pattern. Any additional items represent captured groups inside the expression (if there are no capturing groups in the pattern, then the array has only one item). Consider the following:



```
var text = "mom and dad and baby";
var pattern = /mom( and dad( and baby)?)/gi;

var matches = pattern.exec(text);
alert(matches.index);      //0
alert(matches.input);     //"mom and dad and baby"
alert(matches[0]);         //"mom and dad and baby"
alert(matches[1]);         // " and dad and baby"
alert(matches[2]);         // " and baby"
```

RegExpExecExample01.htm

In this example, the pattern has two capturing groups. The innermost one matches " and baby", and its enclosing group matches " and dad" or " and dad and baby". When `exec()` is called on the string, a match is found. Because the entire string matches the pattern, the `index` property on the `matches` array is set to 0. The first item in the array is the entire matched string, the second contains the contents of the first capturing group, and the third contains the contents of the third capturing group.

The `exec()` method returns information about one match at a time even if the pattern is global. When the global flag is not specified, calling `exec()` on the same string multiple times will always return information about the first match. With the `g` flag set on the pattern, each call to `exec()` moves further into the string looking for matches, as in this example:

```
var text = "cat, bat, sat, fat";
var pattern1 = /.at/;

var matches = pattern1.exec(text);
alert(matches.index);      //0
alert(matches[0]);         //cat
alert(pattern1.lastIndex); //0

matches = pattern1.exec(text);
alert(matches.index);      //0
alert(matches[0]);         //cat
alert(pattern1.lastIndex); //0

var pattern2 = /.at/g;
var matches = pattern2.exec(text);
```

```

alert(matches.index);           //0
alert(matches[0]);             //cat
alert(pattern2.lastIndex);     //0

matches = pattern2.exec(text);
alert(matches.index);           //5
alert(matches[0]);             //bat
alert(pattern2.lastIndex);     //8

```

[RegExpExecExample02.htm](#)

The first pattern in this example, `pattern1`, is not global, so each call to `exec()` returns the first match only ("cat"). The second pattern, `pattern2`, is global, so each call to `exec()` returns the next match in the string until the end of the string is reached. Note also how the pattern's `lastIndex` property is affected. In global matching mode, `lastIndex` is incremented after each call to `exec()`, but it remains unchanged in nonglobal mode.



A deviation in the Internet Explorer implementation of JavaScript causes `lastIndex` to always be updated, even in nonglobal mode.

Another method of regular expressions is `test()`, which accepts a string argument and returns `true` if the pattern matches the argument and `false` if it does not. This method is useful when you want to know if a pattern is matched, but you have no need for the actual matched text. The `test()` method is often used in `if` statements, such as the following:

```

var text = "000-00-0000";
var pattern = /\d{3}-\d{2}-\d{4}/;

if (pattern.test(text)){
    alert("The pattern was matched.");
}

```

In this example, the regular expression tests for a specific numeric sequence. If the input text matches the pattern, then a message is displayed. This functionality is often used for validating user input, when you care only if the input is valid, not necessarily why it's invalid.

The inherited methods of `toLocaleString()` and `toString()` each return the literal representation of the regular expression, regardless of how it was created. Consider this example:



```

var pattern = new RegExp("\\[bc\\]at", "gi");
alert(pattern.toString());           // /\[bc\]at/gi
alert(pattern.toLocaleString());     // /\[bc\]at/gi

```

[RegExpToStringExample01.htm](#)

Even though the pattern in this example is created using the `RegExp` constructor, the `toLocaleString()` and `toString()` methods return the pattern as if it were specified in literal format.



The `valueOf()` method for a regular expression returns the regular expression itself.

RegExp Constructor Properties

The `RegExp` constructor function has several properties. (These would be considered static properties in other languages.) These properties apply to all regular expressions that are in scope, and they change based on the last regular-expression operation that was performed. Another unique element of these properties is that they can be accessed in two different ways. Each property has a verbose property name and a shorthand name (except in Opera, which doesn't support the short names). The `RegExp` constructor properties are listed in the following table.

VERBOSE NAME	SHORT NAME	DESCRIPTION
<code>input</code>	<code>\$_</code>	The last string matched against. This is not implemented in Opera.
<code>lastMatch</code>	<code>\$&</code>	The last matched text. This is not implemented in Opera.
<code>lastParen</code>	<code>\$+</code>	The last matched capturing group. This is not implemented in Opera.
<code>leftContext</code>	<code>\$`</code>	The text that appears in the <code>input</code> string prior to <code>lastMatch</code> .
<code>multiline</code>	<code>\$*</code>	A Boolean value specifying whether all expressions should use multiline mode. This is not implemented in IE or Opera.
<code>rightContext</code>	<code>\$'</code>	The text that appears in the <code>input</code> string after <code>lastMatch</code> .

These properties can be used to extract specific information about the operation performed by `exec()` or `test()`. Consider this example:



```
var text = "this has been a short summer";
var pattern = /(.)hort/g;
/*
 * Note: Opera doesn't support input, lastMatch, lastParen, or multiline.
 * Internet Explorer doesn't support multiline.
 */
if (pattern.test(text)) {
```

Available for
download on
Wrox.com

```

        alert(RegExp.input);           //this has been a short summer
        alert(RegExp.leftContext);    //this has been a
        alert(RegExp.rightContext);   // summer
        alert(RegExp.lastMatch);     //short
        alert(RegExp.lastParen);     //s
        alert(RegExp.multiline);     //false
    }

```

[RegExpConstructorPropertiesExample01.htm](#)

This code creates a pattern that searches for any character followed by "hort" and puts a capturing group around the first letter. The various properties are used as follows:

- The `input` property contains the original string.
- The `leftContext` property contains the characters of the string before the word "short", and the `rightContext` property contains the characters after the word "short".
- The `lastMatch` property contains the last string that matches the entire regular expression, which is "short".
- The `lastParen` property contains the last matched capturing group, which is "s" in this case.

These verbose property names can be replaced with the short property names, although you must use bracket notation to access them, as shown in the following example, because most are illegal identifiers in ECMAScript:



Available for
download on
Wrox.com

```

var text = "this has been a short summer";
var pattern = /(.)hort/g;

/*
 * Note: Opera doesn't short property names.
 * Internet Explorer doesn't support multiline.
 */
if (pattern.test(text)){
    alert(RegExp.$_);           //this has been a short summer
    alert(RegExp[$`"]);          //this has been a
    alert(RegExp[$'"]);          // summer
    alert(RegExp[$&]);          //short
    alert(RegExp[$+"]);          //s
    alert(RegExp[$*"]);          //false
}

```

[RegExpConstructorPropertiesExample02.htm](#)

There are also constructor properties that store up to nine capturing-group matches. These properties are accessed via `RegExp.$1`, which contains the first capturing-group match through

`RegExp.$9`, which contains the ninth capturing-group match. These properties are filled in when calling either `exec()` or `test()`, allowing you to do things like this:



```
var text = "this has been a short summer";
var pattern = /(..)or(..)/g;
if (pattern.test(text)){
    alert(RegExp.$1);           //sh
    alert(RegExp.$2);           //t
}
```

RegExpConstructorPropertiesExample03.htm

In this example, a pattern with two matching groups is created and tested against a string. Even though `test()` simply returns a Boolean value, the properties `$1` and `$2` are filled in on the `RegExp` constructor.

Pattern Limitations

Although ECMAScript's regular-expression support is fully developed, it does lack some of the advanced regular-expression features available in languages such as Perl. The following features are not supported in ECMAScript regular expressions (for more information, see www.regular-expressions.info):

- The `\A` and `\Z` anchors (matching the start or end of a string, respectively)
- Lookbehinds
- Union and intersection classes
- Atomic grouping
- Unicode support (except for matching a single character at a time)
- Named capturing groups
- The `s` (single-line) and `x` (free-spacing) matching modes
- Conditionals
- Regular-expression comments

Despite these limitations, ECMAScript's regular-expression support is powerful enough for doing most pattern-matching tasks.

THE FUNCTION TYPE

Some of the most interesting parts of ECMAScript are its functions, primarily because functions actually are objects. Each function is an instance of the `Function` type that has properties and methods just like any other reference type. Because functions are objects, function names are simply pointers to function objects and are not necessarily tied to the function itself. Functions are typically defined using function-declaration syntax, as in this example:

```
function sum (num1, num2) {
    return num1 + num2;
}
```

This is almost exactly equivalent to using a function expression, such as this:

```
var sum = function(num1, num2) {
    return num1 + num2;
};
```

In this code, a variable `sum` is defined and initialized to be a function. Note that there is no name included after the `function` keyword, because it's not needed — the function can be referenced by the variable `sum`. Also note that there is a semicolon after the end of the function, just as there would be after any variable initialization.

The last way to define functions is by using the `Function` constructor, which accepts any number of arguments. The last argument is always considered to be the function body, and the previous arguments enumerate the new function's arguments. Take this for example:

```
var sum = new Function("num1", "num2", "return num1 + num2"); //not recommended
```

This syntax is not recommended because it causes a double interpretation of the code (once for the regular ECMAScript code and once for the strings that are passed into the constructor) and thus can affect performance. However, it's important to think of functions as objects and function names as pointers — this syntax is great at representing that concept.

Because function names are simply pointers to functions, they act like any other variable containing a pointer to an object. This means it's possible to have multiple names for a single function, as in this example:



Available for
download on
Wrox.com

```
function sum(num1, num2){
    return num1 + num2;
}
alert(sum(10,10)); //20

var anotherSum = sum;
alert(anotherSum(10,10)); //20

sum = null;
alert(anotherSum(10,10)); //20
```

FunctionTypeExample01.htm

This code defines a function named `sum()` that adds two numbers together. A variable, `anotherSum`, is declared and set equal to `sum`. Note that using the function name without parentheses accesses the function pointer instead of executing the function. At this point, both `anotherSum` and `sum` point to the same function, meaning that `anotherSum()` can be called and a result returned. When `sum` is set to `null`, it severs its relationship with the function, although `anotherSum()` can still be called without any problems.

No Overloading (Revisited)

Thinking of function names as pointers also explains why there can be no function overloading in ECMAScript. Recall the following example from Chapter 3:

```
function addSomeNumber(num) {
    return num + 100;
}

function addSomeNumber(num) {
    return num + 200;
}

var result = addSomeNumber(100); //300
```

In this example, it's clear that declaring two functions with the same name always results in the last function overwriting the previous one. This code is almost exactly equivalent to the following:

```
var addSomeNumber = function (num) {
    return num + 100;
};

addSomeNumber = function (num) {
    return num + 200;
};

var result = addSomeNumber(100); //300
```

In this rewritten code, it's much easier to see exactly what is going on. The variable `addSomeNumber` is simply being overwritten when the second function is created.

Function Declarations versus Function Expressions

Throughout this section, the function declaration and function expression are referred to as being almost equivalent. This hedging is due to one major difference in the way that a JavaScript engine loads data into the execution context. Function declarations are read and available in an execution context before any code is executed, whereas function expressions aren't complete until the execution reaches that line of code. Consider the following:



Available for
download on
Wrox.com

```
alert(sum(10,10));
function sum(num1, num2){
    return num1 + num2;
}
```

[FunctionDeclarationExample01.htm](#)

This code runs perfectly, because function declarations are read and added to the execution context before the code begins running through a process called *function declaration hoisting*. As the code is being evaluated, the JavaScript engine does a first pass for function declarations and pulls them to the top of the source tree. So even though the function declaration appears after its usage in the

actual source code, the engine changes this to *hoist* the function declarations to the top. Changing the function declaration to an equivalent function expression, as in the following example, will cause an error during execution:



```
alert(sum(10,10));
var sum = function(num1, num2){
    return num1 + num2;
};
```

[FunctionInitializationExample01.htm](#)

This updated code will cause an error, because the function is part of an initialization statement, not part of a function declaration. That means the function isn't available in the variable sum until the highlighted line has been executed, which won't happen, because the first line causes an "unexpected identifier" error.

Aside from this difference in when the function is available by the given name, the two syntaxes are equivalent.



It is possible to have named function expressions that look like declarations, such as var sum = function sum() {}. See Chapter 7 for a longer discussion on function expressions.

Functions as Values

Because function names in ECMAScript are nothing more than variables, functions can be used anywhere any other value can be used. This means it's possible not only to pass a function into another function as an argument but also to return a function as the result of another function. Consider the following function:

```
function callSomeFunction(someFunction, someArgument) {
    return someFunction(someArgument);
}
```

This function accepts two arguments. The first argument should be a function, and the second argument should be a value to pass to that function. Any function can then be passed in as follows:

```
function add10(num) {
    return num + 10;
}

var result1 = callSomeFunction(add10, 10);
alert(result1); //20

function getGreeting(name) {
    return "Hello, " + name;
```

```

}

var result2 = callSomeFunction(getGreeting, "Nicholas");
alert(result2); //Hello, Nicholas

```

[FunctionAsAnArgumentExample01.htm](#)

The `callSomeFunction()` function is generic, so it doesn't matter what function is passed in as the first argument — the result will always be returned from the first argument being executed. Remember that to access a function pointer instead of executing the function, you must leave off the parentheses, so the variables `add10` and `getGreeting` are passed into `callSomeFunction()` instead of their results being passed in.

Returning a function from a function is also possible and can be quite useful. For instance, suppose that you have an array of objects and want to sort the array on an arbitrary object property. A comparison function for the array's `sort()` method accepts only two arguments, which are the values to compare, but really you need a way to indicate which property to sort by. This problem can be addressed by defining a function to create a comparison function based on a property name, as in the following example:



Available for
download on
Wrox.com

```

function createComparisonFunction(propertyName) {
    return function(object1, object2){
        var value1 = object1[propertyName];
        var value2 = object2[propertyName];

        if (value1 < value2){
            return -1;
        } else if (value1 > value2){
            return 1;
        } else {
            return 0;
        }
    };
}

```

[FunctionReturningFunctionExample01.htm](#)

This function's syntax may look complicated, but essentially it's just a function inside of a function, preceded by the `return` operator. The `propertyName` argument is accessible from the inner function and is used with bracket notation to retrieve the value of the given property. Once the property values are retrieved, a simple comparison can be done. This function can be used as in the following example:

```

var data = [{name: "Zachary", age: 28}, {name: "Nicholas", age: 29}];

data.sort(createComparisonFunction("name"));
alert(data[0].name); //Nicholas

data.sort(createComparisonFunction("age"));
alert(data[0].name); //Zachary

```

In this code, an array called `data` is created with two objects. Each object has a `name` property and an `age` property. By default, the `sort()` method would call `toString()` on each object to determine the sort order, which wouldn't give logical results in this case. Calling `createComparisonFunction("name")` creates a comparison function that sorts based on the `name` property, which means the first item will have the name "Nicholas" and an age of 29. When `createComparisonFunction("age")` is called, it creates a comparison function that sorts based on the `age` property, meaning the first item will be the one with its `name` equal to "Zachary" and `age` equal to 28.

Function Internals

Two special objects exist inside a function: `arguments` and `this`. The `arguments` object, as discussed in Chapter 3, is an array-like object that contains all of the arguments that were passed into the function. Though its primary use is to represent function arguments, the `arguments` object also has a property named `callee`, which is a pointer to the function that owns the `arguments` object. Consider the following classic factorial function:

```
function factorial(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * factorial(num-1)
    }
}
```

Factorial functions are typically defined to be recursive, as in this example, which works fine when the name of the function is set and won't be changed. However, the proper execution of this function is tightly coupled with the function name "factorial". It can be decoupled by using `arguments.callee` as follows:



Available for
download on
Wrox.com

```
function factorial(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * arguments.callee(num-1)
    }
}
```

[FunctionTypeArgumentsExample01.htm](#)

In this rewritten version of the `factorial()` function, there is no longer a reference to the name "factorial" in the function body, which ensures that the recursive call will happen on the correct function no matter how the function is referenced. Consider the following:

```
var trueFactorial = factorial;

factorial = function(){
    return 0;
};

alert(trueFactorial(5)); //120
alert(factorial(5)); //0
```

Here, the variable `trueFactorial` is assigned the value of `factorial`, effectively storing the function pointer in a second location. The `factorial` variable is then reassigned to a function that simply returns 0. Without using `argumentscallee` in the original `factorial()` function's body, the call to `trueFactorial()` would return 0. However, with the function decoupled from the function name, `trueFactorial()` correctly calculates the factorial, and `factorial()` is the only function that returns 0.

The other special object is called `this`, which operates similar to the `this` object in Java and C# though isn't exactly the same. It is a reference to the context object that the function is operating on — often called the `this value` (when a function is called in the global scope of a web page, the `this` object points to `window`). Consider the following:



Available for
download on
Wrox.com

```
window.color = "red";
var o = { color: "blue" };

function sayColor(){
    alert(this.color);
}

sayColor();      // "red"

o.sayColor = sayColor;
o.sayColor();    // "blue"
```

FunctionTypeThisExample01.htm

The function `sayColor()` is defined globally but references the `this` object. The value of `this` is not determined until the function is called, so its value may not be consistent throughout the code execution. When `sayColor()` is called in the global scope, it outputs "red" because `this` is pointing to `window`, which means `this.color` evaluates to `window.color`. By assigning the function to the object `o` and then calling `o.sayColor()`, the `this` object points to `o`, so `this.color` evaluates to `o.color` and "blue" is displayed.



Remember that function names are simply variables containing pointers, so the global `sayColor()` function and `o.sayColor()` point to the same function even though they execute in different contexts.

ECMAScript 5 also formalizes an additional property on a function object: `caller`. Though not defined in ECMAScript 3, all browsers except earlier versions of Opera supported this property, which contains a reference to the function that called this function or `null` if the function was called from the global scope. For example:

```
function outer(){
    inner();
}

function inner(){}
```

```

        alert(inner.caller);
    }

outer();

```

[FunctionTypeArgumentsCallerExample01.htm](#)

This code displays an alert with the source text of the `outer()` function. Because `outer()` calls `inner()`, then `inner.caller` points back to `outer()`. For looser coupling, you can also access the same information via `argumentscallee.caller`:



```

function outer(){
    inner();
}

function inner(){
    alert(argumentscallee.caller);
}

outer();

```

[FunctionTypeArgumentsCallerExample02.htm](#)

The `caller` property is supported in all versions of Internet Explorer, Firefox, Chrome, and Safari, as well as Opera 9.6.

When function code executes in strict mode, attempting to access `argumentscallee` results in an error. ECMAScript 5 also defines `arguments.caller`, which also results in an error in strict mode and is always undefined outside of strict mode. This is to clear up confusion between `arguments.caller` and the `caller` property of functions. These changes were made as security additions to the language, so third-party code could not inspect other code running in the same context.

Strict mode places one additional restriction: you cannot assign a value to the `caller` property of a function. Doing so results in an error.

Function Properties and Methods

Functions are objects in ECMAScript and, as mentioned previously, therefore have properties and methods. Each function has two properties: `length` and `prototype`. The `length` property indicates the number of named arguments that the function expects, as in this example:

```

function sayName(name) {
    alert(name);
}

function sum(num1, num2) {
    return num1 + num2;
}

function sayHi() {

```

```

        alert("hi");
    }

    alert(sayName.length); //1
    alert(sum.length); //2
    alert(sayHi.length); //0

```

FunctionTypeLengthPropertyExample01.htm

This code defines three functions, each with a different number of named arguments. The `sayName()` function specifies one argument, so its `length` property is set to 1. Similarly, the `sum()` function specifies two arguments, so its `length` property is 2, and `sayHi()` has no named arguments, so its `length` is 0.

The `prototype` property is perhaps the most interesting part of the ECMAScript core. The `prototype` is the actual location of all instance methods for reference types, meaning methods such as `toString()` and `valueOf()` actually exist on the `prototype` and are then accessed from the object instances. This property is very important in terms of defining your own reference types and inheritance. (These topics are covered in Chapter 6.) In ECMAScript 5, the `prototype` property is not enumerable and so will not be found using `for-in`.

There are two additional methods for functions: `apply()` and `call()`. These methods both call the function with a specific `this` value, effectively setting the value of the `this` object inside the function body. The `apply()` method accepts two arguments: the value of `this` inside the function and an array of arguments. This second argument may be an instance of `Array`, but it can also be the `arguments` object. Consider the following:



```

function sum(num1, num2){
    return num1 + num2;
}

function callSum1(num1, num2){
    return sum.apply(this, arguments); //passing in arguments object
}

function callSum2(num1, num2){
    return sum.apply(this, [num1, num2]); //passing in array
}

alert(callSum1(10,10)); //20
alert(callSum2(10,10)); //20

```

FunctionTypeApplyMethodExample01.htm

In this example, `callSum1()` executes the `sum()` method, passing in `this` as the `this` value (which is equal to `window` because it's being called in the global scope) and also passing in the `arguments` object. The `callSum2()` method also calls `sum()`, but it passes in an array of the arguments instead. Both functions will execute and return the correct result.



In strict mode, the this value of a function called without a context object is not coerced to window. Instead, this becomes undefined unless explicitly set by either attaching the function to an object or using apply() or call().

The `call()` method exhibits the same behavior as `apply()`, but arguments are passed to it differently. The first argument is the `this` value, but the remaining arguments are passed directly into the function. Using `call()` arguments must be enumerated specifically, as in this example:



Available for download on Wrox.com

```
function sum(num1, num2){
    return num1 + num2;
}

function callSum(num1, num2){
    return sum.call(this, num1, num2);
}

alert(callSum(10,10)); //20
```

[FunctionTypeCallMethodExample01.htm](#)

Using the `call()` method, `callSum()` must pass in each of its arguments explicitly. The result is the same as using `apply()`. The decision to use either `apply()` or `call()` depends solely on the easiest way for you to pass arguments into the function. If you intend to pass in the `arguments` object directly or if you already have an array of data to pass in, then `apply()` is the better choice; otherwise, `call()` may be a more appropriate choice. (If there are no arguments to pass in, these methods are identical.)

The true power of `apply()` and `call()` lies not in their ability to pass arguments but rather in their ability to augment the `this` value inside of the function. Consider the following example:

```
window.color = "red";
var o = { color: "blue" };

function sayColor(){
    alert(this.color);
}

sayColor(); //red

sayColor.call(this); //red
sayColor.call(window); //red
sayColor.call(o); //blue
```

[FunctionTypeCallExample01.htm](#)

This example is a modified version of the one used to illustrate the `this` object. Once again, `sayColor()` is defined as a global function, and when it's called in the global scope, it displays

"red" because `this.color` evaluates to `window.color`. You can then call the function explicitly in the global scope by using `sayColor.call(this)` and `sayColor.call(window)`, which both display "red". Running `sayColor.call(o)` switches the context of the function such that `this` points to `o`, resulting in a display of "blue".

The advantage of using `call()` (or `apply()`) to augment the scope is that the object doesn't need to know anything about the method. In the first version of this example, the `sayColor()` function was placed directly on the object `o` before it was called; in the updated example, that step is no longer necessary.

ECMAScript 5 defines an additional method called `bind()`. The `bind()` method creates a new function instance whose `this` value is *bound* to the value that was passed into `bind()`. For example:



```
window.color = "red";
var o = { color: "blue" };

function sayColor(){
    alert(this.color);
}
var objectSayColor = sayColor.bind(o);
objectSayColor(); //blue
```

[FunctionTypeBindMethodExample01.htm](#)

Here, a new function called `objectSayColor()` is created from `sayColor()` by calling `bind()` and passing in the object `o`. The `objectSayColor()` function has a `this` value equivalent to `o`, so calling the function, even as a global call, results in the string "blue" being displayed. The advantages of this technique are discussed in Chapter 22.

The `bind()` method is supported in Internet Explorer 9+, Firefox 4+, Safari 5.1+, Opera 12+, and Chrome.

For functions, the inherited methods `toLocaleString()` and `toString()` always return the function's code. The exact format of this code varies from browser to browser — some return your code exactly as it appeared in the source code, including comments, whereas others return the internal representation of your code, which has comments removed and possibly some code changes that the interpreter made. Because of these differences, you can't rely on what is returned for any important functionality, though this information may be useful for debugging purposes. The inherited method `valueOf()` simply returns the function itself.

PRIMITIVE WRAPPER TYPES

Three special reference types are designed to ease interaction with primitive values: the `Boolean` type, the `Number` type, and the `String` type. These types can act like the other reference types described in this chapter, but they also have a special behavior related to their primitive-type equivalents. Every time a primitive value is read, an object of the corresponding primitive wrapper

type is created behind the scenes, allowing access to any number of methods for manipulating the data. Consider the following example:

```
var s1 = "some text";
var s2 = s1.substring(2);
```

In this code, `s1` is a variable containing a string, which is a primitive value. On the next line, the `substring()` method is called on `s1` and stored in `s2`. Primitive values aren't objects, so logically they shouldn't have methods, though this still works as you would expect. In truth, there is a lot going on behind the scenes to allow this seamless operation. When `s1` is accessed in the second line, it is being accessed in read mode, which is to say that its value is being read from memory. Any time a string value is accessed in read mode, the following three steps occur:

- 1.** Create an instance of the `String` type.
- 2.** Call the specified method on the instance.
- 3.** Destroy the instance.

You can think of these three steps as they're used in the following three lines of ECMAScript code:

```
var s1 = new String("some text");
var s2 = s1.substring(2);
s1 = null;
```

This behavior allows the primitive string value to act like an object. These same three steps are repeated for Boolean and numeric values using the `Boolean` and `Number` types, respectively.

The major difference between reference types and primitive wrapper types is the lifetime of the object. When you instantiate a reference type using the `new` operator, it stays in memory until it goes out of scope, whereas automatically created primitive wrapper objects exist for only one line of code before they are destroyed. This means that properties and methods cannot be added at runtime. Take this for example:

```
var s1 = "some text";
s1.color = "red";
alert(s1.color); //undefined
```

Here, the second line attempts to add a `color` property to the string `s1`. However, when `s1` is accessed on the third line, the `color` property is gone. This happens because the `String` object that was created in the second line is destroyed by the time the third line is executed. The third line creates its own `String` object, which doesn't have the `color` property.

It is possible to create the primitive wrapper objects explicitly using the `Boolean`, `Number`, and `String` constructors. This should be done only when absolutely necessary, because it is often confusing for developers as to whether they are dealing with a primitive or reference value. Calling `typeof` on an instance of a primitive wrapper type returns `"object"`, and all primitive wrapper objects convert to the Boolean value `true`.

The `Object` constructor also acts as a factory method and is capable of returning an instance of a primitive wrapper based on the type of value passed into the constructor. For example:

```
var obj = new Object("some text");
alert(obj instanceof String); //true
```

When a string is passed into the `Object` constructor, an instance of `String` is created; a number argument results in an instance of `Number`, while a Boolean argument returns an instance of `Boolean`.

Keep in mind that calling a primitive wrapper constructor using `new` is not the same as calling the casting function of the same name. For example:

```
var value = "25";
var number = Number(value); //casting function
alert(typeof number); //"number"

var obj = new Number(value); //constructor
alert(typeof obj); //"object"
```

In this example, the variable `number` is filled with a primitive number value of 25 while the variable `obj` is filled with an instance of `Number`. For more on casting functions, see Chapter 3.

Even though it's not recommended to create primitive wrapper objects explicitly, their functionality is important in being able to manipulate primitive values. Each primitive wrapper type has methods that make data manipulation easier.

The Boolean Type

The `Boolean` type is the reference type corresponding to the Boolean values. To create a `Boolean` object, use the `Boolean` constructor and pass in either `true` or `false`, as in the following example:

```
var booleanObject = new Boolean(true);
```

Instances of `Boolean` override the `valueOf()` method to return a primitive value of either `true` or `false`. The `toString()` method is also overridden to return a string of "`true`" or "`false`" when called. Unfortunately, not only are `Boolean` objects of little use in ECMAScript, they can actually be rather confusing. The problem typically occurs when trying to use `Boolean` objects in Boolean expressions, as in this example:



```
var falseObject = new Boolean(false);
var result = falseObject && true;
alert(result); //true

var falseValue = false;
result = falseValue && true;
alert(result); //false
```

[BooleanTypeExample01.htm](#)

In this code, a `Boolean` object is created with a value of `false`. That same object is then ANDed with the primitive value `true`. In Boolean math, `false` AND `true` is equal to `false`. However, in this

line of code, it is the object named `falseObject` being evaluated, not its value (`false`). As discussed earlier, all objects are automatically converted to `true` in Boolean expressions, so `falseObject` actually is given a value of `true` in the expression. Then, `true` ANDed with `true` is equal to `true`.

There are a couple of other differences between the primitive and the reference Boolean types. The `typeof` operator returns "boolean" for the primitive but "object" for the reference. Also, a Boolean object is an instance of the Boolean type and will return `true` when used with the `instanceof` operator, whereas a primitive value returns `false`, as shown here:

```
alert(typeof falseObject);    //object
alert(typeof falseValue);    //boolean
alert(falseObject instanceof Boolean); //true
alert(falseValue instanceof Boolean); //false
```

It's very important to understand the difference between a primitive Boolean value and a Boolean object — it is recommended to never use the latter.

The Number Type

The `Number` type is the reference type for numeric values. To create a `Number` object, use the `Number` constructor and pass in any number. Here's an example:



Available for
download on
Wrox.com

```
var numberObject = new Number(10);
```

[NumberTypeExample01.htm](#)

As with the `Boolean` type, the `Number` type overrides `valueOf()`, `toLocaleString()`, and `toString()`. The `valueOf()` method returns the primitive numeric value represented by the object, whereas the other two methods return the number as a string. As mentioned in Chapter 3, the `toString()` method optionally accepts a single argument indicating the radix in which to represent the number, as shown in the following examples:

```
var num = 10;
alert(num.toString());      //"10"
alert(num.toString(2));    //"1010"
alert(num.toString(8));    //"12"
alert(num.toString(10));   //"10"
alert(num.toString(16));   //"a"
```

[NumberTypeExample01.htm](#)

Aside from the inherited methods, the `Number` type has several additional methods used to format numbers as strings.

The `toFixed()` method returns a string representation of a number with a specified number of decimal points, as in this example:

```
var num = 10;
alert(num.toFixed(2));     //"10.00"
```

[NumberTypeExample01.htm](#)

Here, the `toFixed()` method is given an argument of 2, which indicates how many decimal places should be displayed. As a result, the method returns the string "10.00", filling out the empty decimal places with zeros. If the number has more than the given number of decimal places, the result is rounded to the nearest decimal place, as shown here:

```
var num = 10.005;
alert(num.toFixed(2));      //"10.01"
```

The rounding nature of `toFixed()` may be useful for applications dealing with currency, though it's worth noting that rounding using this method differs between browsers. Internet Explorer through version 8 incorrectly rounds numbers in the range $\{(-0.94, -0.5], [0.5, 0.94)\}$ when zero is passed to `toFixed()`. In these cases, Internet Explorer returns 0 when it should return either -1 or 1 (depending on the sign); other browsers behave as expected, and Internet Explorer 9 fixes this issue.



The `toFixed()` method can represent numbers with 0 through 20 decimal places. Some browsers may support larger ranges, but this is the typically implemented range.

Another method related to formatting numbers is the `toExponential()` method, which returns a string with the number formatted in exponential notation (aka e-notation). Just as with `toFixed()`, `toExponential()` accepts one argument, which is the number of decimal places to output. Consider this example:

```
var num = 10;
alert(num.toExponential(1));      //"1.0e+1"
```

This code outputs "1.0e+1" as the result. Typically, this small number wouldn't be represented using e-notation. If you want to have the most appropriate form of the number, the `toPrecision()` method should be used instead.

The `toPrecision()` method returns either the fixed or the exponential representation of a number, depending on which makes the most sense. This method takes one argument, which is the total number of digits to use to represent the number (not including exponents). Here's an example:



```
var num = 99;
alert(num.toPrecision(1));      //"1e+2"
alert(num.toPrecision(2));      //"99"
alert(num.toPrecision(3));      //"99.0"
```

[NumberTypeExample01.htm](#)

In this example, the first task is to represent the number 99 with a single digit, which results in "1e+2", otherwise known as 100. Because 99 cannot accurately be represented by just one digit, the method rounded up to 100, which can be represented using just one digit. Representing 99 with two digits yields "99" and with three digits returns "99.0". The `toPrecision()` method

essentially determines whether to call `toFixed()` or `toExponential()` based on the numeric value you're working with; all three methods round up or down to accurately represent a number with the correct number of decimal places.



The `toPrecision()` method can represent numbers with 1 through 21 decimal places. Some browsers may support larger ranges, but this is the typically implemented range.

Similar to the `Boolean` object, the `Number` object gives important functionality to numeric values but really should not be instantiated directly because of the same potential problems. The `typeof` and `instanceof` operators work differently when dealing with primitive numbers versus reference numbers, as shown in the following examples:

```
var numberObject = new Number(10);
var numberValue = 10;
alert(typeof numberObject);    // "object"
alert(typeof numberValue);    // "number"
alert(numberObject instanceof Number); // true
alert(numberValue instanceof Number); // false
```

Primitive numbers always return "number" when `typeof` is called on them, whereas `Number` objects return "object". Similarly, a `Number` object is an instance of `Number`, but a primitive number is not.

The String Type

The `String` type is the object representation for strings and is created using the `String` constructor as follows:



Available for
download on
[Wrox.com](#)

```
var stringObject = new String("hello world");
```

[StringTypeExample01.htm](#)

The methods of a `String` object are available on all string primitives. All three of the inherited methods — `valueOf()`, `toLocaleString()`, and `toString()` — return the object's primitive string value.

Each instance of `String` contains a single property, `length`, which indicates the number of characters in the string. Consider the following example:

```
var stringValue = "hello world";
alert(stringValue.length);    // "11"
```

This example outputs "11", the number of characters in "hello world". Note that even if the string contains a double-byte character (as opposed to an ASCII character, which uses just one byte), each character is still counted as one.

The `String` type has a large number of methods to aid in the dissection and manipulation of strings in ECMAScript.

Character Methods

Two methods access specific characters in the string: `charAt()` and `charCodeAt()`. These methods each accept a single argument, which is the character's zero-based position. The `charAt()` method simply returns the character in the given position as a single-character string. (There is no character type in ECMAScript.) For example:

```
var stringValue = "hello world";
alert(stringValue.charAt(1)); // "e"
```

The character in position 1 of "hello world" is "e", so calling `charAt(1)` returns "e". If you want the character's character code instead of the actual character, then calling `charCodeAt()` is the appropriate choice, as in the following example:

```
var stringValue = "hello world";
alert(stringValue.charCodeAt(1)); // outputs "101"
```

This example outputs "101", which is the character code for the lowercase "e" character.

ECMAScript 5 defines another way to access an individual character. Supporting browsers allow you to use bracket notation with a numeric index to access a specific character in the string, as in this example:

```
var stringValue = "hello world";
alert(stringValue[1]); // "e"
```

Individual character access using bracket notation is supported in Internet Explorer 8 and all current versions of Firefox, Safari, Chrome, and Opera. If this syntax is used in Internet Explorer 7 or earlier, the result is undefined (though not the special value `undefined`).

String-Manipulation Methods

Several methods manipulate the values of strings. The first of these methods is `concat()`, which is used to concatenate one or more strings to another, returning the concatenated string as the result. Consider the following example:

```
var stringValue = "hello ";
var result = stringValue.concat("world");
alert(result); // "hello world"
alert(stringValue); // "hello"
```

The result of calling the `concat()` method on `stringValue` in this example is "hello world" — the value of `stringValue` remains unchanged. The `concat()` method accepts any number of arguments, so it can create a string from any number of other strings, as shown here:

```
var stringValue = "hello ";
var result = stringValue.concat("world", "!");
```

```
alert(result);           // "hello world!"
alert(stringValue);    // "hello"
```

This modified example concatenates "world" and "!" to the end of "hello ". Although the `concat()` method is provided for string concatenation, the addition operator (+) is used more often and, in most cases, actually performs better than the `concat()` method even when concatenating multiple strings.

ECMAScript provides three methods for creating string values from a substring: `slice()`, `substr()`, and `substring()`. All three methods return a substring of the string they act on, and all accept either one or two arguments. The first argument is the position where capture of the substring begins; the second argument, if used, indicates where the operation should stop. For `slice()` and `substring()`, this second argument is the position before which capture is stopped (all characters up to this point are included except the character at that point). For `substr()`, the second argument is the number of characters to return. If the second argument is omitted in any case, it is assumed that the ending position is the length of the string. Just as with the `concat()` method, `slice()`, `substr()`, and `substring()` do not alter the value of the string itself — they simply return a primitive string value as the result, leaving the original unchanged. Consider this example:



Available for
download on
Wrox.com

```
var stringValue = "hello world";
alert(stringValue.slice(3));      // "lo world"
alert(stringValue.substring(3));   // "lo world"
alert(stringValue.substr(3));     // "lo world"
alert(stringValue.slice(3, 7));   // "lo w"
alert(stringValue.substring(3,7)); // "lo w"
alert(stringValue.substr(3, 7));  // "lo worl"
```

StringTypeManipulationMethodsExample01.htm

In this example, `slice()`, `substr()`, and `substring()` are used in the same manner and, in most cases, return the same value. When given just one argument, 3, all three methods return "lo world", because the second "l" in "hello" is in position 3. When given two arguments, 3 and 7, `slice()` and `substring()` return "lo w" (the "o" in "world" is in position 7, so it is not included), while `substr()` returns "lo worl", because the second argument specifies the number of characters to return.

There are different behaviors for these methods when an argument is a negative number. For the `slice()` method, a negative argument is treated as the length of the string plus the negative argument.

For the `substr()` method, a negative first argument is treated as the length of the string plus the number, whereas a negative second number is converted to 0. For the `substring()` method, all negative numbers are converted to 0. Consider this example:

```
var stringValue = "hello world";
alert(stringValue.slice(-3));      // "rld"
alert(stringValue.substring(-3));   // "hello world"
alert(stringValue.substr(-3));     // "rld"
```

```
alert(stringValue.slice(3, -4));      //"lo w"
alert(stringValue.substring(3, -4));   //"hel"
alert(stringValue.substr(3, -4));     //"" (empty string)
```

StringTypeManipulationMethodsExample01.htm

This example clearly indicates the differences between three methods. When `slice()` and `substr()` are called with a single negative argument, they act the same. This occurs because `-3` is translated into `7` (the length plus the argument), effectively making the calls `slice(7)` and `substr(7)`. The `substring()` method, on the other hand, returns the entire string, because `-3` is translated to `0`.



Because of a deviation in the Internet Explorer implementation of JavaScript, passing in a negative number to substr() results in the original string being returned. Internet Explorer 9 fixes this issue.

When the second argument is negative, the three methods act differently from one another. The `slice()` method translates the second argument to `7`, making the call equivalent to `slice(3, 7)` and so returning `"lo w"`. For the `substring()` method, the second argument gets translated to `0`, making the call equivalent to `substring(3, 0)`, which is actually equivalent to `substring(0, 3)`, because this method expects that the smaller number is the starting position and the larger one is the ending position. For the `substr()` method, the second argument is also converted to `0`, which means there should be zero characters in the returned string, leading to the return value of an empty string.

String Location Methods

There are two methods for locating substrings within another string: `indexOf()` and `lastIndexOf()`. Both methods search a string for a given substring and return the position (or `-1` if the substring isn't found). The difference between the two is that the `indexOf()` method begins looking for the substring at the beginning of the string, whereas the `lastIndexOf()` method begins looking from the end of the string. Consider this example:



```
var stringValue = "hello world";
alert(stringValue.indexOf("o"));           //4
alert(stringValue.lastIndexOf("o"));       //7
```

StringTypeLocationMethodsExample01.htm

Here, the first occurrence of the string `"o"` is at position `4`, which is the `"o"` in `"hello"`. The last occurrence of the string `"o"` is in the word `"world"`, at position `7`. If there is only one occurrence of `"o"` in the string, then `indexOf()` and `lastIndexOf()` return the same position.

Each method accepts an optional second argument that indicates the position to start searching from within the string. This means that the `indexOf()` method will start searching from that position and go toward the end of the string, ignoring everything before the start position, whereas

`lastIndexOf()` starts searching from the given position and continues searching toward the beginning of the string, ignoring everything between the given position and the end of the string. Here's an example:

```
var stringValue = "hello world";
alert(stringValue.indexOf("o", 6));           //7
alert(stringValue.lastIndexOf("o", 6));        //4
```

When the second argument of 6 is passed into each method, the results are the opposite from the previous example. This time, `indexOf()` returns 7 because it starts searching the string from position 6 (the letter "w") and continues to position 7, where "o" is found. The `lastIndexOf()` method returns 4 because the search starts from position 6 and continues back toward the beginning of the string, where it encounters the "o" in "hello". Using this second argument allows you to locate all instances of a substring by looping callings to `indexOf()` or `lastIndexOf()`, as in the following example:



Available for
download on
Wrox.com

```
var stringValue = "Lorem ipsum dolor sit amet, consectetur adipisicing elit";
var positions = new Array();
var pos = stringValue.indexOf("e");

while(pos > -1){
    positions.push(pos);
    pos = stringValue.indexOf("e", pos + 1);
}

alert(positions);      //"3,24,32,35,52"
```

StringTypeLocationMethodsExample02.htm

This example works through a string by constantly increasing the position at which `indexOf()` should begin. It begins by getting the initial position of "e" in the string and then enters a loop that continually passes in the last position plus one to `indexOf()`, ensuring that the search continues after the last substring instance. Each position is stored in the `positions` array so the data can be used later.

The `trim()` Method

ECMAScript 5 introduces a `trim()` method on all strings. The `trim()` method creates a copy of the string, removes all leading and trailing white space, and then returns the result. For example:

```
var stringValue = "    hello world    ";
var trimmedStringValue = stringValue.trim();
alert(stringValue);           //"    hello world    "
alert(trimmedStringValue);    //"hello world"
```

Note that since `trim()` returns a copy of a string, the original string remains intact with leading and trailing white space in place. This method has been implemented in Internet Explorer 9+, Firefox 3.5+, Safari 5+, Opera 10.5+, and Chrome. Firefox 3.5+, Safari 5+, and Chrome 8+ also

support two nonstandard `trimLeft()` and `trimRight()` methods that remove white space only from the beginning or end of the string, respectively.

String Case Methods

The next set of methods involves case conversion. Four methods perform case conversion: `toLowerCase()`, `toLocaleLowerCase()`, `toUpperCase()`, and `toLocaleUpperCase()`. The `toLowerCase()` and `toUpperCase()` methods are the original methods, modeled after the same methods in `java.lang.String`. The `toLocaleLowerCase()` and `toLocaleUpperCase()` methods are intended to be implemented based on a particular locale. In many locales, the locale-specific methods are identical to the generic ones; however, a few languages (such as Turkish) apply special rules to Unicode case conversion, and this necessitates using the locale-specific methods for proper conversion. Here are some examples:



```
var stringValue = "hello world";
alert(stringValue.toLocaleUpperCase());    // "HELLO WORLD"
alert(stringValue.toUpperCase());           // "HELLO WORLD"
alert(stringValue.toLocaleLowerCase());     // "hello world"
alert(stringValue.toLowerCase());           // "hello world"
```

StringTypeCaseMethodExample01.htm

This code outputs "HELLO WORLD" for both `toLocaleUpperCase()` and `toUpperCase()`, just as "hello world" is output for both `toLocaleLowerCase()` and `toLowerCase()`. Generally speaking, if you do not know the language in which the code will be running, it is safer to use the locale-specific methods.

String Pattern-Matching Methods

The `String` type has several methods designed to pattern-match within the string. The first of these methods is `match()` and is essentially the same as calling a `RegExp` object's `exec()` method. The `match()` method accepts a single argument, which is either a regular-expression string or a `RegExp` object. Consider this example:

```
var text = "cat, bat, sat, fat";
var pattern = /.at/;

//same as pattern.exec(text)
var matches = text.match(pattern);
alert(matches.index);          //0
alert(matches[0]);            //"cat"
alert(pattern.lastIndex);      //0
```

StringTypePatternMatchingExample01.htm

The array returned from `match()` is the same array that is returned when the `RegExp` object's `exec()` method is called with the string as an argument: the first item is the string that matches the entire pattern, and each other item (if applicable) represents capturing groups in the expression.

Another method for finding patterns is `search()`. The only argument for this method is the same as the argument for `match()`: a regular expression specified by either a string or a `RegExp` object. The `search()` method returns the index of the first pattern occurrence in the string or `-1` if it's not found. `search()` always begins looking for the pattern at the beginning of the string. Consider this example:



Available for download on
Wrox.com

```
var text = "cat, bat, sat, fat";
var pos = text.search(/at/);
alert(pos); //1
```

[StringTypePatternMatchingExample01.htm](#)

Here, `search(/at/)` returns `1`, which is the first position of "at" in the string.

To simplify replacing substrings, ECMAScript provides the `replace()` method. This method accepts two arguments. The first argument can be a `RegExp` object or a string (the string is not converted to a regular expression), and the second argument can be a string or a function. If the first argument is a string, then only the first occurrence of the substring will be replaced. The only way to replace all instances of a substring is to provide a regular expression with the global flag specified, as in this example:

```
var text = "cat, bat, sat, fat";
var result = text.replace("at", "ond");
alert(result); //"cond, bat, sat, fat"

result = text.replace(/at/g, "ond");
alert(result); //"cond, bond, sond, fond"
```

[StringTypePatternMatchingExample01.htm](#)

In this example, the string "at" is first passed into `replace()` with a replacement text of "ond". The result of the operation is that "cat" is changed to "cond", but the rest of the string remains intact. By changing the first argument to a regular expression with the global flag set, each instance of "at" is replaced with "ond".

When the second argument is a string, there are several special character sequences that can be used to insert values from the regular-expression operations. ECMA-262 specifies the following table of values.

SEQUENCE	REPLACEMENT TEXT
\$\$	\$
\$&	The substring matching the entire pattern. Same as <code>RegExp.lastMatch</code> .
\$'	The part of the string occurring before the matched substring. Same as <code>RegExp.rightContext</code> .
\$`	The part of the string occurring after the matched substring. Same as <code>RegExp.leftContext</code> .

continues

(continued)

SEQUENCE	REPLACEMENT TEXT
\$n	The <i>n</i> th capture, where <i>n</i> is a value 0–9. For instance, \$1 is the first capture, \$2 is the second, etc. If there is no capture then the empty string is used.
\$nn	The <i>nn</i> th capture, where <i>nn</i> is a value 01–99. For instance, \$01 is the first capture, \$02 is the second, etc. If there is no capture then the empty string is used.

Using these special sequences allows replacement using information about the last match, such as in this example:



```
var text = "cat, bat, sat, fat";
result = text.replace(/(.at)/g, "word ($1)");
alert(result); //word (cat), word (bat), word (sat), word (fat)
```

StringTypePatternMatchingExample01.htm

Here, each word ending with "at" is replaced with "word" followed in parentheses by what it replaces by using the \$1 sequence.

The second argument of `replace()` may also be a function. When there is a single match, the function gets passed three arguments: the string match, the position of the match within the string, and the whole string. When there are multiple capturing groups, each matched string is passed in as an argument, with the last two arguments being the position of the pattern match in the string and the original string. The function should return a string indicating what the match should be replaced with. Using a function as the second argument allows more granular control over replacement text, as in this example:

```
function htmlEscape(text){
    return text.replace(/[<>=&]/g, function(match, pos, originalText) {
        switch(match) {
            case "<":
                return "&lt;";
            case ">":
                return "&gt;";
            case "&":
                return "&amp;";
            case "\\":
                return "&quot;";
        }
    });
}

alert(htmlEscape("<p class=\"greeting\">Hello world!</p>"));
//&lt;p class="greeting"&gt;Hello world!&lt;/p&gt";
```

StringTypePatternMatchingExample01.htm

Here, the function `htmlEscape()` is defined to escape four characters for insertion into HTML: the less-than, greater-than, ampersand, and double-quote characters all must be escaped. The easiest way to accomplish this is to have a regular expression to look for those characters and then define a function that returns the specific HTML entities for each matched character.

The last string method for dealing with patterns is `split()`, which separates the string into an array of substrings based on a separator. The separator may be a string or a `RegExp` object. (The string is not considered a regular expression for this method.) An optional second argument, the array limit, ensures that the returned array will be no larger than a certain size. Consider this example:



Available for
download on
Wrox.com

```
var colorText = "red,blue,green,yellow";
var colors1 = colorText.split(",");      //["red", "blue", "green", "yellow"]
var colors2 = colorText.split(", ", 2);   //["red", "blue"]
var colors3 = colorText.split(/[^,\s]+/); //["", "", "", "", "", ""]
```

[StringTypePatternMatchingExample01.htm](#)

In this example, the string `colorText` is a comma-separated string of colors. The call to `split(",")` retrieves an array of those colors, splitting the string on the comma character. To truncate the results to only two items, a second argument of 2 is specified. Last, using a regular expression, it's possible to get an array of the comma characters. Note that in this last call to `split()`, the returned array has an empty string before and after the commas. This happens because the separator specified by the regular expression appears at the beginning of the string (the substring "red") and at the end (the substring "yellow").

Browsers vary in their exact support for regular expressions in the `split()` method. While simple patterns typically work the same way, patterns where no match is found and patterns with capturing groups can behave wildly different across browsers. Some of the notable differences are as follows:

- Internet Explorer through version 8 ignores capturing groups. ECMA-262 indicates that the groups should be spliced into the result array. Internet Explorer 9 correctly includes the capturing groups in the results.
- Firefox through version 3.6 includes empty strings in the results array when a capturing group has no match; ECMA-262 specifies that capturing groups without a match should be represented as `undefined` in the results array.

There are other, subtle differences when using capturing groups in regular expressions. When using such regular expressions, make sure to test thoroughly across a host of browsers.



For a larger discussion about the cross-browser issues with `split()` and capturing groups in regular expressions, please see “JavaScript split bugs: Fixed!” by Steven Levithan at <http://blog.stevenlevithan.com/archives/cross-browser-split>.

The `localeCompare()` Method

The last method is `localeCompare()`, which compares one string to another and returns one of three values as follows:

- If the string should come alphabetically before the string argument, a negative number is returned. (Most often this is `-1`, but it is up to each implementation as to the actual value.)
- If the string is equal to the string argument, `0` is returned.
- If the string should come alphabetically after the string argument, a positive number is returned. (Most often this is `1`, but once again, this is implementation-specific.)

Here's an example:



```
var stringValue = "yellow";
alert(stringValue.localeCompare("brick")); //1
alert(stringValue.localeCompare("yellow")); //0
alert(stringValue.localeCompare("zoo")); // -1
```

[StringTypeLocaleCompareExample01.htm](#)

In this code, the string `"yellow"` is compared to three different values: `"brick"`, `"yellow"`, and `"zoo"`. Because `"brick"` comes alphabetically before `"yellow"`, `localeCompare()` returns `1`; `"yellow"` is equal to `"yellow"`, so `localeCompare()` returns `0` for that line; and `"zoo"` comes after `"yellow"`, so `localeCompare()` returns `-1` for that line. Once again, because the values are implementation-specific, it is best to use `localeCompare()` as shown in this example:

```
function determineOrder(value) {
    var result = stringValue.localeCompare(value);
    if (result < 0) {
        alert("The string 'yellow' comes before the string '" + value + "' .");
    } else if (result > 0) {
        alert("The string 'yellow' comes after the string '" + value + "' .");
    } else {
        alert("The string 'yellow' is equal to the string '" + value + "' .");
    }
}

determineOrder("brick");
determineOrder("yellow");
determineOrder("zoo");
```

[StringTypeLocaleCompareExample01.htm](#)

By using this sort of construct, you can be sure that the code works correctly in all implementations.

The unique part of `localeCompare()` is that an implementation's locale (country and language) indicates exactly how this method operates. In the United States, where English is the standard language for ECMAScript implementations, `localeCompare()` is case-sensitive, determining that uppercase letters come alphabetically after lowercase letters. However, this may not be the case in other locales.

The fromCharCode() Method

There is one method on the `String` constructor: `fromCharCode()`. This method's job is to take one or more character codes and convert them into a string. Essentially, this is the reverse operation from the `charCodeAt()` instance method. Consider this example:



Available for download on
Wrox.com

```
alert(String.fromCharCode(104, 101, 108, 108, 111)); // "hello"
```

[StringTypeFromCharCodeExample01.htm](#)

In this code, `fromCharCode()` is called on a series of character codes from the letters in the word "hello".

HTML Methods

The web browser vendors recognized a need early on to format HTML dynamically using JavaScript. As a result, they extended the specification to include several methods specifically designed to aid in common HTML formatting tasks. The following table enumerates the HTML methods. However, be aware that typically these methods aren't used, because they tend to create nonsemantic markup.

METHOD	OUTPUT
<code>anchor(name)</code>	<code>string</code>
<code>big()</code>	<code><big>string</big></code>
<code>bold()</code>	<code>string</code>
<code>fixed()</code>	<code><tt>string</tt></code>
<code>fontcolor(color)</code>	<code>string</code>
<code>fontsize(size)</code>	<code>string</code>
<code>italics()</code>	<code><i>string</i></code>
<code>link(url)</code>	<code>string</code>
<code>small()</code>	<code><small>string</small></code>
<code>strike()</code>	<code><strike>string</strike></code>
<code>sub()</code>	<code><sub>string</sub></code>
<code>sup()</code>	<code><sup>string</sup></code>

SINGLETON BUILT-IN OBJECTS

ECMA-262 defines a built-in object as “any object supplied by an ECMA-Script implementation, independent of the host environment, which is present at the start of the execution of an ECMA-Script program.” This means the developer does not need to explicitly instantiate a built-in object; it is

already instantiated. You have already learned about most of the built-in objects, such as `Object`, `Array`, and `String`. There are two singleton built-in objects defined by ECMA-262: `Global` and `Math`.

The Global Object

The `Global` object is the most unique in ECMAScript, because it isn't explicitly accessible. ECMA-262 specifies the `Global` object as a sort of catchall for properties and methods that don't otherwise have an owning object. In truth, there is no such thing as a global variable or global function; all variables and functions defined globally become properties of the `Global` object. Functions covered earlier in this book, such as `isNaN()`, `isFinite()`, `parseInt()`, and `parseFloat()` are actually methods of the `Global` object. In addition to these, there are several other methods available on the `Global` object.

URI-Encoding Methods

The `encodeURI()` and `encodeURIComponent()` methods are used to encode URIs (Uniform Resource Identifiers) to be passed to the browser. To be valid, a URI cannot contain certain characters, such as spaces. The URI-encoding methods encode the URIs so that a browser can still accept and understand them, replacing all invalid characters with a special UTF-8 encoding.

The `encodeURI()` method is designed to work on an entire URI (for instance, `www.wrox.com/illegal value.htm`), whereas `encodeURIComponent()` is designed to work solely on a segment of a URI (such as `illegal value.htm` from the previous URI). The main difference between the two methods is that `encodeURI()` does not encode special characters that are part of a URI, such as the colon, forward slash, question mark, and pound sign, whereas `encodeURIComponent()` encodes every nonstandard character it finds. Consider this example:



Available for download on Wrox.com

```
var uri = "http://www.wrox.com/illegal value.htm#start";
// "http://www.wrox.com/illegal%20value.htm#start"
alert(encodeURI(uri));

// "http%3A%2Fwww.wrox.com%2Fillegal%20value.htm%23start"
alert(encodeURIComponent(uri));
```

GlobalObjectURIEncodingExample01.htm

Here, using `encodeURI()` left the value completely intact except for the space, which was replaced with `%20`. The `encodeURIComponent()` method replaced all nonalphanumeric characters with their encoded equivalents. This is why `encodeURI()` can be used on full URIs, whereas `encodeURIComponent()` can be used only on strings that are appended to the end of an existing URI.



Generally speaking, you'll use `encodeURIComponent()` much more frequently than `encodeURI()`, because it's more common to encode query string arguments separately from the base URI.

The two counterparts to `encodeURI()` and `encodeURIComponent()` are `decodeURI()` and `decodeURIComponent()`. The `decodeURI()` method decodes only characters that would have been replaced by using `encodeURI()`. For instance, `%20` is replaced with a space, but `%23` is not replaced because it represents a pound sign (#), which `encodeURI()` does not replace. Likewise, `decodeURIComponent()` decodes all characters encoded by `encodeURIComponent()`, essentially meaning it decodes all special values. Consider this example:



Available for download on
Wrox.com

```
var uri = "http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.htm%23start";
//http%3A%2F%2Fwww.wrox.com%2Fillegal value.htm%23start
alert(decodeURI(uri));

//http://www.wrox.com/illegal value.htm#start
alert(decodeURIComponent(uri));
```

GlobalObjectURIDecodingExample01.htm

Here, the `uri` variable contains a string that is encoded using `encodeURIComponent()`. The first value output is the result of `decodeURI()`, which replaced only the `%20` with a space. The second value is the output of `decodeURIComponent()`, which replaces all the special characters and outputs a string that has no escaping in it. (This string is not a valid URI.)



The URI methods `encodeURI()`, `encodeURIComponent()`, `decodeURI()`, and `decodeURIComponent()` replace the `escape()` and `unescape()` methods, which are deprecated in the ECMA-262 third edition. The URI methods are always preferable, because they encode all Unicode characters, whereas the original methods encode only ASCII characters correctly. Avoid using `escape()` and `unescape()` in production code.

The `eval()` Method

The final method is perhaps the most powerful in the entire ECMAScript language: the `eval()` method. This method works like an entire ECMAScript interpreter and accepts one argument, a string of ECMAScript (or JavaScript) to execute. Here's an example:

```
eval("alert('hi')");
```

This line is functionally equivalent to the following:

```
alert("hi");
```

When the interpreter finds an `eval()` call, it interprets the argument into actual ECMAScript statements and then inserts it into place. Code executed by `eval()` is considered to be part of the execution context in which the call is made, and the executed code has the same scope chain as that

context. This means variables that are defined in the containing context can be referenced inside an `eval()` call, such as in this example:

```
var msg = "hello world";
eval("alert(msg)"); // "hello world"
```

Here, the variable `msg` is defined outside the context of the `eval()` call, yet the call to `alert()` still displays the text "hello world", because the second line is replaced with a real line of code. Likewise, you can define a function or variables inside an `eval()` call that can be referenced by the code outside, as follows:

```
eval("function sayHi() { alert('hi'); }");
sayHi();
```

Here, the `sayHi()` function is defined inside an `eval()` call. Because that call is replaced with the actual function, it is possible to call `sayHi()` on the following line. This works the same for variables:

```
eval("var msg = 'hello world';");
alert(msg); // "hello world"
```

Any variables or functions created inside of `eval()` will not be hoisted, as they are contained within a string when the code is being parsed. They are created only at the time of `eval()` execution.

In strict mode, variables and functions created inside of `eval()` are not accessible outside, so these last two examples would cause errors. Also, in strict mode, assigning a value to `eval` causes an error:

```
"use strict";
eval = "hi"; //causes error
```



The capability to interpret strings of code is very powerful but also very dangerous. Use extreme caution with `eval()`, especially when passing user-entered data into it. A mischievous user could insert values that might compromise your site or application security. (This is called code injection.)

Global Object Properties

The `Global` object has a number of properties, some of which have already been mentioned in this book. The special values of `undefined`, `NaN`, and `Infinity` are all properties of the `Global` object. Additionally, all native reference type constructors, such as `Object` and `Function`, are properties of the `Global` object. The following table lists all of the properties.

PROPERTY	DESCRIPTION
<code>undefined</code>	The special value <code>undefined</code>
<code>NaN</code>	The special value <code>NaN</code>
<code>Infinity</code>	The special value <code>Infinity</code>

PROPERTY	DESCRIPTION
Object	Constructor for Object
Array	Constructor for Array
Function	Constructor for Function
Boolean	Constructor for Boolean
String	Constructor for String
Number	Constructor for Number
Date	Constructor for Date
RegExp	Constructor for RegExp
Error	Constructor for Error
EvalError	Constructor for EvalError
RangeError	Constructor for RangeError
ReferenceError	Constructor for ReferenceError
SyntaxError	Constructor for SyntaxError
TypeError	Constructor for TypeError
URIError	Constructor for URIError

In ECMAScript 5, it's explicitly disallowed to assign values to `undefined`, `Nan`, and `Infinity`. Doing so causes an error even in nonstrict mode.

The Window Object

Though ECMA-262 doesn't indicate a way to access the Global object directly, web browsers implement it such that the `window` is the Global object's delegate. Therefore, all variables and functions declared in the global scope become properties on `window`. Consider this example:



Available for download on
Wrox.com

```
var color = "red";
function sayColor(){
    alert(window.color);
}
window.sayColor(); // "red"
```

[GlobalObjectWindowExample01.htm](#)

Here, a global variable named `color` and a global function named `sayColor()` are defined. Inside `sayColor()`, the `color` variable is accessed via `window.color` to show that the global variable became a property of `window`. The function is then called directly off of the `window` object as `window.sayColor()`, which pops up the alert.



The window object does much more in JavaScript than just implement the ECMAScript Global object. Details of the window object and the Browser Object Model are discussed in Chapter 8.

Another way to retrieve the `Global` object is to use the following code:

```
var global = function(){
    return this;
}();
```

This code creates an immediately-invoked function expression that returns the value of `this`. As mentioned previously, the `this` value is equivalent to the `Global` object when a function is executed with no explicit `this` value specified (either by being an object method or via `call()`/`apply()`). Thus, calling a function that simply returns `this` is a consistent way to retrieve the `Global` object in any execution environment. Function expressions are discussed further in Chapter 7.

The Math Object

ECMAScript provides the `Math` object as a common location for mathematical formulas and information. The computations available on the `Math` object execute faster than if you were to write the computations in JavaScript directly. There are a number of properties and methods to help these computations.

Math Object Properties

The `Math` object has several properties, consisting mostly of special values in the world of mathematics. The following table describes these properties.

PROPERTY	DESCRIPTION
<code>Math.E</code>	The value of e , the base of the natural logarithms
<code>Math.LN10</code>	The natural logarithm of 10
<code>Math.LN2</code>	The natural logarithm of 2
<code>Math.LOG2E</code>	The base 2 logarithm of e
<code>Math.LOG10E</code>	The base 10 logarithm of e

PROPERTY	DESCRIPTION
Math.PI	The value of π
Math.SQRT1_2	The square root of $\frac{1}{2}$
Math.SQRT2	The square root of 2

Although the meanings and uses of these values are outside the scope of this book, they are available if and when you need them.

The min() and max() Methods

The `Math` object also contains many methods aimed at performing both simple and complex mathematical calculations.

The `min()` and `max()` methods determine which number is the smallest or largest in a group of numbers. These methods accept any number of parameters, as shown in the following example:



```
var max = Math.max(3, 54, 32, 16);
alert(max);      //54

var min = Math.min(3, 54, 32, 16);
alert(min);      //3
```

[MathObjectMinMaxExample01.htm](#)

Out of the numbers 3, 54, 32, and 16, `Math.max()` returns the number 54, whereas `Math.min()` returns the number 3. These methods are useful for avoiding extra loops and `if` statements to determine the maximum value out of a group of numbers.

To find the maximum or the minimum value in an array, you can use the `apply()` method as follows:

```
var values = [1, 2, 3, 4, 5, 6, 7, 8];
var max = Math.max.apply(Math, values);
```

The key to this technique is to pass in the `Math` object as the first argument of `apply()` so that the `this` value is set appropriately. Then you can pass an array in as the second argument.

Rounding Methods

The next group of methods has to do with rounding decimal values into integers. Three methods — `Math.ceil()`, `Math.floor()`, and `Math.round()` — handle rounding in different ways as described here:

- The `Math.ceil()` method represents the ceiling function, which always rounds numbers up to the nearest integer value.
- The `Math.floor()` method represents the floor function, which always rounds numbers down to the nearest integer value.

- The `Math.round()` method represents a standard round function, which rounds up if the number is at least halfway to the next integer value (0.5 or higher) and rounds down if not. This is the way you were taught to round in elementary school.

The following example illustrates how these methods work:



```
alert(Math.ceil(25.9));      //26
alert(Math.ceil(25.5));      //26
alert(Math.ceil(25.1));      //26

alert(Math.round(25.9));     //26
alert(Math.round(25.5));     //26
alert(Math.round(25.1));     //25

alert(Math.floor(25.9));     //25
alert(Math.floor(25.5));     //25
alert(Math.floor(25.1));     //25
```

MathObjectRoundingExample01.htm

For all values between 25 and 26 (exclusive), `Math.ceil()` always returns 26, because it will always round up. The `Math.round()` method returns 26 only if the number is 25.5 or greater; otherwise it returns 25. Last, `Math.floor()` returns 25 for all numbers between 25 and 26 (exclusive).

The `random()` Method

The `Math.random()` method returns a random number between the 0 and the 1, not including either 0 or 1. This is a favorite tool of web sites that are trying to display random quotes or random facts upon entry of a web site. You can use `Math.random()` to select numbers within a certain integer range by using the following formula:

```
number = Math.floor(Math.random() * total_number_of_choices + first_possible_value)
```

The `Math.floor()` method is used here because `Math.random()` always returns a decimal value, meaning that multiplying it by a number and adding another still yields a decimal value. So, if you wanted to select a number between 1 and 10, the code would look like this:

```
var num = Math.floor(Math.random() * 10 + 1);
```

MathObjectRandomExample01.htm

You see 10 possible values (1 through 10), with the first possible value being 1. If you want to select a number between 2 and 10, then the code would look like this:

```
var num = Math.floor(Math.random() * 9 + 2);
```

MathObjectRandomExample02.htm

There are only nine numbers when counting from 2 to 10, so the total number of choices is nine, with the first possible value being 2. Many times, it's just easier to use a function that handles the calculation of the total number of choices and the first possible value, as in this example:

```
function selectFrom(lowerValue, upperValue) {
    var choices = upperValue - lowerValue + 1;
    return Math.floor(Math.random() * choices + lowerValue);
}

var num = selectFrom(2,10);
alert(num); //number between 2 and 10, inclusive
```

[MathObjectRandomExample03.htm](#)

Here, the function `selectFrom()` accepts two arguments: the lowest value that should be returned and the highest value that should be returned. The number of choices is calculated by subtracting the two values and adding one and then applying the previous formula to those numbers. So it's possible to select a number between 2 and 10 (inclusive) by calling `selectFrom(2,10)`. Using the function, it's easy to select a random item from an array, as shown here:

```
var colors = ["red", "green", "blue", "yellow", "black", "purple", "brown"];
var color = colors[selectFrom(0, colors.length-1)];
```

[MathObjectRandomExample03.htm](#)

In this example, the second argument to `selectFrom()` is the length of the array minus 1, which is the last position in an array.

Other Methods

The `Math` object has a lot of methods related to various simple and higher-level mathematical operations. It's beyond the scope of this book to discuss the ins and outs of each or in what situations they may be used, but the following table enumerates the remaining methods of the `Math` object.

METHOD	DESCRIPTION
<code>Math.abs (num)</code>	Returns the absolute value of <code>(num)</code>
<code>Math.exp (num)</code>	Returns <code>Math.E</code> raised to the power of <code>(num)</code>
<code>Math.log (num)</code>	Returns the natural logarithm of <code>(num)</code>
<code>Math.pow (num, power)</code>	Returns <code>num</code> raised to the power of <code>power</code>
<code>Math.sqrt (num)</code>	Returns the square root of <code>(num)</code>
<code>Math.acos (x)</code>	Returns the arc cosine of <code>x</code>

continues

(continued)

METHOD	DESCRIPTION
Math.asin(x)	Returns the arc sine of x
Math.atan(x)	Returns the arc tangent of x
Math.atan2(y, x)	Returns the arc tangent of y/x
Math.cos(x)	Returns the cosine of x
Math.sin(x)	Returns the sine of x
Math.tan(x)	Returns the tangent of x

Even though these methods are defined by ECMA-262, the results are implementation-dependent for those dealing with sines, cosines, and tangents, because you can calculate each value in many different ways. Consequently, the precision of the results may vary from one implementation to another.

SUMMARY

Objects in JavaScript are called reference values, and several built-in reference types can be used to create specific types of objects, as follows:

- Reference types are similar to classes in traditional object-oriented programming but are implemented differently.
- The `Object` type is the base from which all other reference types inherit basic behavior.
- The `Array` type represents an ordered list of values and provides functionality for manipulating and converting the values.
- The `Date` type provides information about dates and times, including the current date and time and calculations.
- The `RegExp` type is an interface for regular-expression support in ECMAScript, providing most basic and some advanced regular-expression functionality.

One of the unique aspects of JavaScript is that functions are actually instances of the `Function` type, meaning functions are objects. Because functions are objects, functions have methods that can be used to augment how they behave.

Because of the existence of primitive wrapper types, primitive values in JavaScript can be accessed as if they were objects. There are three primitive wrapper types: `Boolean`, `Number`, and `String`. They all have the following characteristics:

- Each of the wrapper types maps to the primitive type of the same name.

- When a primitive value is accessed in read mode, a primitive wrapper object is instantiated so that it can be used to manipulate the data.
- As soon as a statement involving a primitive value is executed, the wrapper object is destroyed.

There are also two built-in objects that exist at the beginning of code execution: `Global` and `Math`. The `Global` object isn't accessible in most ECMAScript implementations; however, web browsers implement it as the `window` object. The `Global` object contains all global variables and functions as properties. The `Math` object contains properties and methods to aid in complex mathematical calculations.

6

Object-Oriented Programming

WHAT'S IN THIS CHAPTER?

- ▶ Understanding object properties
- ▶ Understanding and creating objects
- ▶ Understanding inheritance

Object-oriented (OO) languages typically are identified through their use of classes to create multiple objects that have the same properties and methods. As mentioned previously, ECMAScript has no concept of classes, and therefore objects are different than in class-based languages.

ECMA-262 defines an object as an “unordered collection of properties each of which contains a primitive value, object, or function.” Strictly speaking, this means that an object is an array of values in no particular order. Each property or method is identified by a name that is mapped to a value. For this reason (and others yet to be discussed), it helps to think of ECMAScript objects as hash tables: nothing more than a grouping of name-value pairs where the value may be data or a function.

Each object is created based on a reference type, either one of the native types discussed in the previous chapter, or a developer-defined type.

UNDERSTANDING OBJECTS

As mentioned in the previous chapter, the simplest way to create a custom object is to create a new instance of `Object` and add properties and methods to it, as in this example:



Available for
download on
Wrox.com

```
var person = new Object();
person.name = "Nicholas";
person.age = 29;
```

```

person.job = "Software Engineer";

person.sayName = function(){
    alert(this.name);
};

```

CreatingObjectsExample01.htm

This example creates an object called `person` that has three properties (`name`, `age`, and `job`) and one method (`sayName()`). The `sayName()` method displays the value of `this.name`, which resolves to `person.name`. Early JavaScript developers used this pattern frequently to create new objects. A few years later, object literals became the preferred pattern for creating such objects. The previous example can be rewritten using object literal notation as follows:

```

var person = {
    name: "Nicholas",
    age: 29,
    job: "Software Engineer",

    sayName: function(){
        alert(this.name);
    }
};

```

The `person` object in this example is equivalent to the `person` object in the prior example, with all the same properties and methods. These properties are all created with certain characteristics that define their behavior in JavaScript.

Types of Properties

ECMA-262 fifth edition describes characteristics of properties through the use of internal-only attributes. These attributes are defined by the specification for implementation in JavaScript engines, and as such, these attributes are not directly accessible in JavaScript. To indicate that an attribute is internal, surround the attribute name with two pairs of square brackets, such as `[[Enumerable]]`. Although ECMA-262 third edition had different definitions, this book refers only to the fifth edition descriptions.

There are two types of properties: data properties and accessor properties.

Data Properties

Data properties contain a single location for a data value. Values are read from and written to this location. Data properties have four attributes describing their behavior:

- `[[Configurable]]` — Indicates if the property may be redefined by removing the property via `delete`, changing the property's attributes, or changing the property into an accessor property. By default, this is true for all properties defined directly on an object, as in the previous example.

- `[[Enumerable]]` — Indicates if the property will be returned in a `for-in` loop. By default, this is true for all properties defined directly on an object, as in the previous example.
- `[[Writable]]` — Indicates if the property's value can be changed. By default, this is true for all properties defined directly on an object, as in the previous example.
- `[[Value]]` — Contains the actual data value for the property. This is the location from which the property's value is read and the location to which new values are saved. The default value for this attribute is `undefined`.

When a property is explicitly added to an object as in the previous examples, `[[Configurable]]`, `[[Enumerable]]`, and `[[Writable]]` are all set to true while the `[[Value]]` attribute is set to the assigned value. For example:

```
var person = {
    name: "Nicholas"
};
```

Here, the property called `name` is created and a value of "Nicholas" is assigned. That means `[[Value]]` is set to "Nicholas", and any changes to that value are stored in this location.

To change any of the default property attributes, you must use the ECMAScript 5 `Object.defineProperty()` method. This method accepts three arguments: the object on which the property should be added or modified, the name of the property, and a descriptor object. The properties on the descriptor object match the attribute names: `configurable`, `enumerable`, `writable`, and `value`. You can set one or all of these values to change the corresponding attribute values. For example:



Available for
download on
[Wrox.com](#)

```
var person = {};
Object.defineProperty(person, "name", {
    writable: false,
    value: "Nicholas"
});

alert(person.name);      // "Nicholas"
person.name = "Greg";
alert(person.name);      // "Nicholas"
```

[DataPropertiesExample01.htm](#)

This example creates a property called `name` with a value of "Nicholas" that is read-only. The value of this property can't be changed, and any attempts to assign a new value are ignored in nonstrict mode. In strict mode, an error is thrown when an attempt is made to change the value of a read-only property.

Similar rules apply to creating a nonconfigurable property. For example:

```
var person = {};
Object.defineProperty(person, "name", {
    configurable: false,
    value: "Nicholas"
```

```

});  
  

alert(person.name);      // "Nicholas"  

delete person.name;  

alert(person.name);      // "Nicholas"

```

[DataPropertiesExample02.htm](#)

Here, setting `configurable` to `false` means that the property cannot be removed from the object. Calling `delete` on the property has no effect in nonstrict mode and throws an error in strict mode. Additionally, once a property has been defined as nonconfigurable, it cannot become configurable again. Any attempt to call `Object.defineProperty()` and change any attribute other than `writable` causes an error:



Available for
download on
Wrox.com

```

var person = {};  

Object.defineProperty(person, "name", {  

    configurable: false,  

    value: "Nicholas"  

});  
  

//throws an error  

Object.defineProperty(person, "name", {  

    configurable: true,  

    value: "Nicholas"  

});

```

[DataPropertiesExample03.htm](#)

So although you can call `Object.defineProperty()` multiple times for the same property, there are limits once `configurable` has been set to `false`.

When you are using `Object.defineProperty()`, the values for `configurable`, `enumerable`, and `writable` default to `false` unless otherwise specified. In most cases, you likely won't need the powerful options provided by `Object.defineProperty()`, but it's important to understand the concepts to have a good understanding of JavaScript objects.



Internet Explorer 8 was the first version to implement `Object.defineProperty()`. Unfortunately, the implementation is extremely limited. This method can be used only on DOM objects and can create only accessor properties. It's recommended you avoid using `Object.defineProperty()` in Internet Explorer 8 because of its incomplete implementation.

Accessor Properties

Accessor properties do not contain a data value. Instead, they contain a combination of a getter function and a setter function (though both are not necessary). When an accessor property is read from, the getter function is called, and it's the function's responsibility to return a valid value; when

an accessor property is written to, a function is called with the new value, and that function must decide how to react to the data. Accessor properties have four attributes:

- `[Configurable]` — Indicates if the property may be redefined by removing the property via `delete`, changing the property's attributes, or changing the property into a data property. By default, this is true for all properties defined directly on an object.
- `[Enumerable]` — Indicates if the property will be returned in a `for-in` loop. By default, this is true for all properties defined directly on an object.
- `[Get]` — The function to call when the property is read from. The default value is `undefined`.
- `[Set]` — The function to call when the property is written to. The default value is `undefined`.

It is not possible to define an accessor property explicitly; you must use `Object.defineProperty()`. Here's a simple example:



Available for
download on
Wrox.com

```
var book = {
    _year: 2004,
    edition: 1
};

Object.defineProperty(book, "year", {
    get: function() {
        return this._year;
    },
    set: function(newValue) {

        if (newValue > 2004) {
            this._year = newValue;
            this.edition += newValue - 2004;
        }
    }
});

book.year = 2005;
alert(book.edition); //2
```

AccessPropertiesExample01.htm

In this code, an object `book` is created with two default properties: `_year` and `edition`. The underscore on `_year` is a common notation to indicate that a property is not intended to be accessed from outside of the object's methods. The `year` property is defined to be an accessor property where the getter function simply returns the value of `_year` and the setter does some calculation to determine the correct `edition`. So changing the `year` property to 2005 results in both `_year` and `edition` changing to 2. This is a typical use case for accessor properties, when setting a property value results in some other changes to occur.

It's not necessary to assign both a getter and a setter. Assigning just a getter means that the property cannot be written to and attempts to do so will be ignored. In strict mode, trying to write to a

property with only a getter throws an error. Likewise, a property with only a setter cannot be read and will return the value `undefined` in nonstrict mode, while doing so throws an error in strict mode.

Prior to the ECMAScript 5 method, which is available in Internet Explorer 9+ (Internet Explorer 8 had a partial implementation), Firefox 4+, Safari 5+, Opera 12+, and Chrome, two nonstandard methods were used to create accessor properties: `__defineGetter__()` and `__defineSetter__()`. These were first developed by Firefox and later copied by Safari 3, Chrome 1, and Opera 9.5. The previous example can be rewritten using these legacy methods as follows:



```
var book = {
    _year: 2004,
    edition: 1
};

//legacy accessor support
book.__defineGetter__("year", function(){
    return this._year;
});

book.__defineSetter__("year", function(newValue){
    if (newValue > 2004) {
        this._year = newValue;
        this.edition += newValue - 2004;
    }
});

book.year = 2005;
alert(book.edition); //2
```

AccessorPropertiesExample02.htm

There is no way to modify `[[Configurable]]` or `[[Enumerable]]` in browsers that don't support `Object.defineProperty()`.

Defining Multiple Properties

Since there's a high likelihood that you'll need to define more than one property on an object, ECMAScript 5 provides the `Object.defineProperties()` method. This method allows you to define multiple properties using descriptors at once. There are two arguments: the object on which to add or modify the properties and an object whose property names correspond to the properties' names to add or modify. For example:

```
var book = {};

Object.defineProperties(book, {
    _year: {
        value: 2004
    },
    edition: {
        value: 1
```

```

        },
        year: {
            get: function(){
                return this._year;
            },
            set: function(newValue){
                if (newValue > 2004) {
                    this._year = newValue;
                    this.edition += newValue - 2004;
                }
            }
        }
    );
}

```

[MultiplePropertiesExample01.htm](#)

This code defines two data properties, `_year` and `edition`, and an accessor property called `year` on the `book` object. The resulting object is identical to the example in the previous section. The only difference is that all of these properties are created at the same time.

The `Object.defineProperties()` method is supported in Internet Explorer 9+, Firefox 4+, Safari 5+, Opera 12+, and Chrome.

Reading Property Attributes

It's also possible to retrieve the property descriptor for a given property by using the ECMAScript 5 `Object.getOwnPropertyDescriptor()` method. This method accepts two arguments: the object on which the property resides and the name of the property whose descriptor should be retrieved. The return value is an object with properties for `configurable`, `enumerable`, `get`, and `set` for accessor properties or `configurable`, `enumerable`, `writable`, and `value` for data properties. Example:



Available for
download on
Wrox.com

```

var book = {};
Object.defineProperties(book, {
    _year: {
        value: 2004
    },
    edition: {
        value: 1
    },
    year: {
        get: function(){
            return this._year;
        },
        set: function(newValue){
            if (newValue > 2004) {
                this._year = newValue;
            }
        }
    }
});

```

```

        this.edition += newValue - 2004;
    }
}
}
});

var descriptor = Object.getOwnPropertyDescriptor(book, "__year");
alert(descriptor.value);           //2004
alert(descriptor.configurable);   //false
alert(typeof descriptor.get);     //"undefined"

var descriptor = Object.getOwnPropertyDescriptor(book, "year");
alert(descriptor.value);          //undefined
alert(descriptor.enumerable);    //false
alert(typeof descriptor.get);    //"function"

```

GetPropertyDescriptorExample01.htm

For the data property `_year`, `value` is equal to the original value, `configurable` is `false`, and `get` is `undefined`. For the accessor property `year`, `value` is `undefined`, `enumerable` is `false`, and `get` is a pointer to the specified getter function.

The `Object.getOwnPropertyDescriptor()` method can be used on any object in JavaScript, including DOM and BOM objects. This method is supported in Internet Explorer 9+, Firefox 4+, Safari 5+, Opera 12+, and Chrome.

OBJECT CREATION

Although using the `Object` constructor or an object literal are convenient ways to create single objects, there is an obvious downside: creating multiple objects with the same interface requires a lot of code duplication. To solve this problem, developers began using a variation of the factory pattern.

The Factory Pattern

The factory pattern is a well-known design pattern used in software engineering to abstract away the process of creating specific objects. (Other design patterns and their implementation in JavaScript are discussed later in the book.) With no way to define classes in ECMAScript, developers created functions to encapsulate the creation of objects with specific interfaces, such as in this example:



```

function createPerson(name, age, job){
    var o = new Object();
    o.name = name;
    o.age = age;
    o.job = job;
    o.sayName = function(){
        alert(this.name);
    };
    return o;
}

```

```

}

var person1 = createPerson("Nicholas", 29, "Software Engineer");
var person2 = createPerson("Greg", 27, "Doctor");

```

[FactoryPatternExample01.htm](#)

Here, the function `createPerson()` accepts arguments with which to build an object with all of the necessary information to represent a `Person` object. The function can be called any number of times with different arguments and will still return an object that has three properties and one method. Though this solved the problem of creating multiple similar objects, the factory pattern didn't address the issue of object identification (what type of object an object is). As JavaScript continued to evolve, a new pattern emerged.

The Constructor Pattern

As mentioned in previous chapters, constructors in ECMAScript are used to create specific types of objects. There are native constructors, such as `Object` and `Array`, which are available automatically in the execution environment at runtime. It is also possible to define custom constructors that define properties and methods for your own type of object. For instance, the previous example can be rewritten using the constructor pattern as the following:



```

function Person(name, age, job){
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName = function(){
        alert(this.name);
    };
}

var person1 = new Person("Nicholas", 29, "Software Engineer");
var person2 = new Person("Greg", 27, "Doctor");

```

[ConstructorPatternExample01.htm](#)

In this example, the `Person()` function takes the place of the factory `createPerson()` function. Note that the code inside `Person()` is the same as the code inside `createPerson()`, with the following exceptions:

- There is no object being created explicitly.
- The properties and method are assigned directly onto the `this` object.
- There is no `return` statement.

Also note the name of the function is `Person` with an uppercase `P`. By convention, constructor functions always begin with an uppercase letter, whereas nonconstructor functions begin with a

lowercase letter. This convention is borrowed from other OO languages and helps to distinguish function use in ECMAScript, since constructors are simply functions that create objects.

To create a new instance of `Person`, use the `new` operator. Calling a constructor in this manner essentially causes the following four steps to be taken:

1. Create a new object.
2. Assign the `this` value of the constructor to the new object (so `this` points to the new object).
3. Execute the code inside the constructor (adds properties to the new object).
4. Return the new object.

At the end of the preceding example, `person1` and `person2` are each filled with a different instance of `Person`. Each of these objects has a `constructor` property that points back to `Person`, as follows:

```
alert(person1.constructor == Person); //true
alert(person2.constructor == Person); //true
```

The `constructor` property was originally intended for use in identifying the object type. However, the `instanceof` operator is considered to be a safer way of determining type. Each of the objects in this example is considered to be both an instance of `Object` and an instance of `Person`, as indicated by using the `instanceof` operator like this:

```
alert(person1 instanceof Object); //true
alert(person1 instanceof Person); //true
alert(person2 instanceof Object); //true
alert(person2 instanceof Person); //true
```

Defining your own constructors ensures that instances can be identified as a particular type later on, which is a great advantage over the factory pattern. In this example, `person1` and `person2` are considered to be instances of `Object`, because all custom objects inherit from `Object` (the specifics of this are discussed later).



Constructors defined in this manner are defined on the Global object (the window object in web browsers). The Browser Object Model (BOM) is discussed further in Chapter 8.

Constructors as Functions

The only difference between constructor functions and other functions is the way in which they are called. Constructors are, after all, just functions; there's no special syntax to define a constructor that automatically makes it behave as such. Any function that is called with the `new` operator acts as a constructor, whereas any function called without it acts just as you would expect a normal function call to act. For instance, the `Person()` function from the previous example may be called in any of the following ways:



```
//use as a constructor
var person = new Person("Nicholas", 29, "Software Engineer");
person.sayName(); //Nicholas

//call as a function
Person("Greg", 27, "Doctor"); //adds to window
window.sayName(); //Greg

//call in the scope of another object
var o = new Object();
Person.call(o, "Kristen", 25, "Nurse");
o.sayName(); //Kristen
```

ConstructorPatternExample02.htm

The first part of this example shows the typical use of a constructor, to create a new object via the `new` operator. The second part shows what happens when the `Person()` function is called without the `new` operator: the properties and methods get added to the `window` object. Remember that the `this` object always points to the `Global` object (`window` in web browsers) when a function is called without an explicitly set `this` value (by being an object method or through `call()`/`apply()`). So after the function is called, the `sayName()` method can be called on the `window` object, and it will return `"Greg"`. The `Person()` function can also be called within the scope of a particular object using `call()` (or `apply()`). In this case, it's called with a `this` value of the object `o`, which then gets assigned all of the properties and the `sayName()` method.

Problems with Constructors

Though the constructor paradigm is useful, it is not without its faults. The major downside to constructors is that methods are created once for each instance. So, in the previous example, both `person1` and `person2` have a method called `sayName()`, but those methods are not the same instance of `Function`. Remember, functions are objects in ECMAScript, so every time a function is defined, it's actually an object being instantiated. Logically, the constructor actually looks like this:

```
function Person(name, age, job){
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName = new Function("alert(this.name)"); //logical equivalent
}
```

Thinking about the constructor in this manner makes it clear that each instance of `Person` gets its own instance of `Function` that happens to display the `name` property. To be clear, creating a function in this manner is different with regard to scope chains and identifier resolution, but the mechanics of creating a new instance of `Function` remain the same. So, functions of the same name on different instances are not equivalent, as the following code proves:

```
alert(person1.sayName == person2.sayName); //false
```

It doesn't make sense to have two instances of `Function` that do the same thing, especially when the `this` object makes it possible to avoid binding functions to particular objects until runtime.



Available for
download on
Wrox.com

```
function Person(name, age, job) {
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName = sayName;
}

function sayName() {
    alert(this.name);
}

var person1 = new Person("Nicholas", 29, "Software Engineer");
var person2 = new Person("Greg", 27, "Doctor");
```

ConstructorPatternExample03.htm

In this example, the `sayName()` function is defined outside the constructor. Inside the constructor, the `sayName` property is set equal to the global `sayName()` function. Since the `sayName` property now contains just a pointer to a function, both `person1` and `person2` end up sharing the `sayName()` function that is defined in the global scope. This solves the problem of having duplicate functions that do the same thing but also creates some clutter in the global scope by introducing a function that can realistically be used only in relation to an object. If the object needed multiple methods, that would mean multiple global functions, and all of a sudden the custom reference type definition is no longer nicely grouped in the code. These problems are addressed by using the prototype pattern.

The Prototype Pattern

Each function is created with a `prototype` property, which is an object containing properties and methods that should be available to instances of a particular reference type. This object is literally a prototype for the object to be created once the constructor is called. The benefit of using the prototype is that all of its properties and methods are shared among object instances. Instead of assigning object information in the constructor, they can be assigned directly to the prototype, as in this example:

```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var person1 = new Person();
person1.sayName(); // "Nicholas"

var person2 = new Person();
```

```

person2.sayName() ;      // "Nicholas"
alert(person1.sayName == person2.sayName) ;  // true

```

[PrototypePatternExample01.htm](#)

Here, the properties and the `sayName()` method are added directly to the `prototype` property of `Person`, leaving the constructor empty. However, it's still possible to call the constructor to create a new object and have the properties and methods present. Unlike the constructor pattern, the properties and methods are all shared among instances, so `person1` and `person2` are both accessing the same set of properties and the same `sayName()` function. To understand how this works, you must understand the nature of prototypes in ECMAScript.

How Prototypes Work

Whenever a function is created, its `prototype` property is also created according to a specific set of rules. By default, all prototypes automatically get a property called `constructor` that points back to the function on which it is a property. In the previous example, for instance, `Person.prototype.constructor` points to `Person`. Then, depending on the constructor, other properties and methods may be added to the prototype.

When defining a custom constructor, the prototype gets the `constructor` property only by default; all other methods are inherited from `Object`. Each time the constructor is called to create a new instance, that instance has an internal pointer to the constructor's prototype. In ECMA-262 fifth edition, this is called `[[Prototype]]`. There is no standard way to access `[[Prototype]]` from script, but Firefox, Safari, and Chrome all support a property on every object called `__proto__`; in other implementations, this property is completely hidden from script. The important thing to understand is that a direct link exists between the instance and the constructor's prototype but not between the instance and the constructor.

Consider the previous example using the `Person` constructor and `Person.prototype`. The relationship between the objects in the example is shown in Figure 6-1.

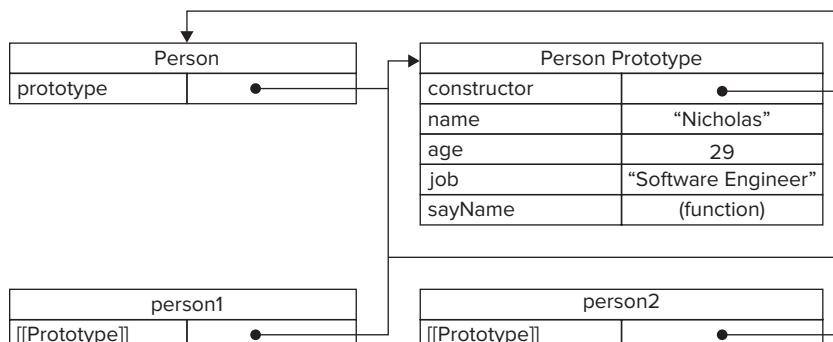


FIGURE 6-1

Figure 6-1 shows the relationship between the `Person` constructor, the `Person`'s prototype, and the two instances of `Person` that exist. Note that `Person.prototype` points to the prototype object but `Person.prototype.constructor` points back to `Person`. The prototype contains the `constructor` property and the other properties that were added. Each instance of `Person`, `person1`, and `person2` has internal properties that point back to `Person.prototype` only; each has no direct relationship with the constructor. Also note that even though neither of these instances have properties or methods, `person1.sayName()` works. This is due to the lookup procedure for object properties.

Even though `[[Prototype]]` is not accessible in all implementations, the `isPrototypeOf()` method can be used to determine if this relationship exists between objects. Essentially, `isPrototypeOf()` returns `true` if `[[Prototype]]` points to the prototype on which the method is being called, as shown here:

```
alert(Person.prototype.isPrototypeOf(person1)); //true
alert(Person.prototype.isPrototypeOf(person2)); //true
```

In this code, the prototype's `isPrototypeOf()` method is called on both `person1` and `person2`. Since both instances have a link to `Person.prototype`, it returns `true`.

ECMAScript 5 adds a new method called `Object.getPrototypeOf()`, which returns the value of `[[Prototype]]` in all supporting implementations. For example:

```
alert(Object.getPrototypeOf(person1) == Person.prototype); //true
alert(Object.getPrototypeOf(person1).name); // "Nicholas"
```

The first line of this code simply confirms that the object returned from `Object.getPrototypeOf()` is actually the prototype of the object. The second line retrieves the value of the `name` property on the prototype, which is "Nicholas". Using `Object.getPrototypeOf()`, you are able to retrieve an object's prototype easily, which becomes important once you want to implement inheritance using the prototype (discussed later in this chapter). This method is supported in Internet Explorer 9+, Firefox 3.5+, Safari 5+, Opera 12+, and Chrome.

Whenever a property is accessed for reading on an object, a search is started to find a property with that name. The search begins on the object instance itself. If a property with the given name is found on the instance, then that value is returned; if the property is not found, then the search continues up the pointer to the prototype, and the prototype is searched for a property with the same name. If the property is found on the prototype, then that value is returned. So, when `person1.sayName()` is called, a two-step process happens. First, the JavaScript engine asks, "Does the instance `person1` have a property called `sayName`?" The answer is no, so it continues the search and asks, "Does the `person1` prototype have a property called `sayName`?" The answer is yes, so the function stored on the prototype is accessed. When `person2.sayName()` is called, the same search executes, ending with the same result. This is how prototypes are used to share properties and methods among multiple object instances.



The constructor property mentioned earlier exists only on the prototype and is accessible from object instances.

Although it's possible to read values on the prototype from object instances, it is not possible to overwrite them. If you add a property to an instance that has the same name as a property on the prototype, you create the property on the instance, which then masks the property on the prototype. Here's an example:



Available for
download on
Wrox.com

```
function Person() {
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var person1 = new Person();
var person2 = new Person();

person1.name = "Greg";
alert(person1.name);    // "Greg" - from instance
alert(person2.name);    // "Nicholas" - from prototype
```

[PrototypePatternExample02.htm](#)

In this example, the `name` property of `person1` is shadowed by a new value. Both `person1.name` and `person2.name` still function appropriately, returning "Greg" (from the object instance) and "Nicholas" (from the prototype), respectively. When `person1.name` was accessed in the `alert()`, its value was read, so the search began for a property called `name` on the instance. Since the property exists, it is used without searching the prototype. When `person2.name` is accessed the same way, the search doesn't find the property on the instance, so it continues to search on the prototype where the `name` property is found.

Once a property is added to the object instance, it *shadows* any properties of the same name on the prototype, which means that it blocks access to the property on the prototype without altering it. Even setting the property to `null` only sets the property on the instance and doesn't restore the link to the prototype. The `delete` operator, however, completely removes the instance property and allows the prototype property to be accessed again as follows:

```
function Person() {
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var person1 = new Person();
var person2 = new Person();

person1.name = "Greg";
```

```

alert(person1.name);    // "Greg" - from instance
alert(person2.name);    // "Nicholas" - from prototype

delete person1.name;
alert(person1.name);    // "Nicholas" - from the prototype

```

PrototypePatternExample03.htm

In this modified example, `delete` is called on `person1.name`, which previously had been shadowed with the value "Greg". This restores the link to the prototype's `name` property, so the next time `person1.name` is accessed, it's the prototype property's value that is returned.

The `hasOwnProperty()` method determines if a property exists on the instance or on the prototype. This method, which is inherited from `Object`, returns `true` only if a property of the given name exists on the object instance, as in this example:

```

function Person() {
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var person1 = new Person();
var person2 = new Person();

alert(person1.hasOwnProperty("name")); //false

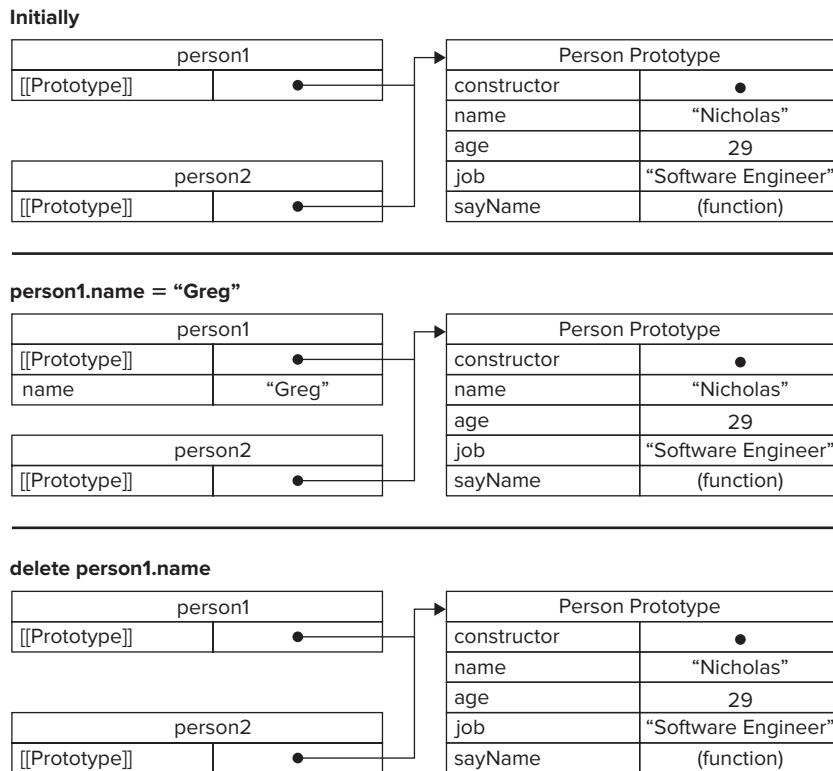
person1.name = "Greg";
alert(person1.name);    // "Greg" - from instance
alert(person1.hasOwnProperty("name")); //true

alert(person2.name);    // "Nicholas" - from prototype
alert(person2.hasOwnProperty("name")); //false

delete person1.name;
alert(person1.name);    // "Nicholas" - from the prototype
alert(person1.hasOwnProperty("name")); //false

```

By injecting calls to `hasOwnProperty()` in this example, it becomes clear when the instance's property is being accessed and when the prototype's property is being accessed. Calling `person1.hasOwnProperty("name")` returns `true` only after `name` has been overwritten on `person1`, indicating that it now has an instance property instead of a prototype property. Figure 6-2 illustrates the various steps being taken in this example. (For simplicity, the relationship to the `Person` constructor has been omitted.)

**FIGURE 6-2**

The ECMAScript 5 `Object.getOwnPropertyDescriptor()` method works only on instance properties; to retrieve the descriptor of a prototype property, you must call `Object.getOwnPropertyDescriptor()` on the prototype object directly.

Prototypes and the `in` Operator

There are two ways to use the `in` operator: on its own or as a `for-in` loop. When used on its own, the `in` operator returns `true` when a property of the given name is accessible by the object, which is to say that the property may exist on the instance or on the prototype. Consider the following example:



```
function Person() {
}
Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
```

Available for download on
Wrox.com

```

Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var person1 = new Person();
var person2 = new Person();

alert(person1.hasOwnProperty("name")); //false
alert("name" in person1); //true

person1.name = "Greg";
alert(person1.name); // "Greg" - from instance
alert(person1.hasOwnProperty("name")); //true
alert("name" in person1); //true

alert(person2.name); // "Nicholas" - from prototype
alert(person2.hasOwnProperty("name")); //false
alert("name" in person2); //true

delete person1.name;
alert(person1.name); // "Nicholas" - from the prototype
alert(person1.hasOwnProperty("name")); //false
alert("name" in person1); //true

```

PrototypePatternExample04.htm

Throughout the execution of this code, the property `name` is available on each object either directly or from the prototype. Therefore, calling `"name"` in `person1` always returns `true`, regardless of whether the property exists on the instance. It's possible to determine if the property of an object exists on the prototype by combining a call to `hasOwnProperty()` with the `in` operator like this:



Available for
download on
Wrox.com

```

function hasPrototypeProperty(object, name){
    return !object.hasOwnProperty(name) && (name in object);
}

```

Since the `in` operator always returns `true` so long as the property is accessible by the object, and `hasOwnProperty()` returns `true` only if the property exists on the instance, a prototype property can be determined if the `in` operator returns `true` but `hasOwnProperty()` returns `false`. Consider the following example:

```

function Person() {
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
}

```

```

};

var person = new Person();
alert(hasPrototypeProperty(person, "name")); //true

person.name = "Greg";
alert(hasPrototypeProperty(person, "name")); //false

```

[PrototypePatternExample05.htm](#)

In this code, the name property first exists on the prototype, so `hasPrototypeProperty()` returns true. Once the name property is overwritten, it exists on the instance, so `hasPrototypeProperty()` returns false. Even though the name property still exists on the prototype, it is no longer used because the instance property now exists.

When using a `for-in` loop, all properties that are accessible by the object and can be enumerated will be returned, which includes properties both on the instance and on the prototype.

Instance properties that shadow a non-enumerable prototype property (a property that has `[[Enumerable]]` set to false) will be returned in the `for-in` loop, since all developer-defined properties are enumerable by rule, except in Internet Explorer 8 and earlier.

The old Internet Explorer implementation has a bug where properties that shadow non-enumerable properties will not show up in a `for-in` loop. Here's an example:



```

var o = {
    toString : function(){
        return "My Object";
    }
};

for (var prop in o){
    if (prop == "toString"){
        alert("Found toString"); //won't display in Internet Explorer
    }
}

```

[PrototypePatternExample06.htm](#)

When this code is run, a single alert should be displayed indicating that the `toString()` method was found. The object `o` has an instance property called `toString()` that shadows the prototype's `toString()` method (which is not enumerable). In Internet Explorer, this alert is never displayed because it skips over the property, honoring the `[[Enumerable]]` attribute that was set on the prototype's `toString()` method. This same bug affects all properties and methods that aren't enumerable by default: `hasOwnProperty()`, `propertyIsEnumerable()`, `toLocaleString()`, `toString()`, and `valueOf()`. ECMAScript 5 sets `[[Enumerable]]` to false on the constructor and prototype properties, but this is inconsistent across implementations.

To retrieve a list of all enumerable instance properties on an object, you can use the ECMAScript 5 `Object.keys()` method, which accepts an object as its argument and returns an array of strings containing the names of all enumerable properties. For example:



```
function Person() {
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var keys = Object.keys(Person.prototype);
alert(keys);          // "name,age,job,sayName"

var p1 = new Person();
p1.name = "Rob";
p1.age = 31;
var p1keys = Object.keys(p1);
alert(p1keys);        // "name,age"
```

ObjectKeysExample01.htm

Here, the `keys` variable is filled with an array containing "name", "age", "job", and "sayName". This is the order in which they would normally appear using `for-in`. When called on an instance of `Person`, `Object.keys()` returns an array of name and age, the only two instance properties.

If you'd like a list of all instance properties, whether enumerable or not, you can use `Object.getOwnPropertyNames()` in the same way:

```
var keys = Object.getOwnPropertyNames(Person.prototype);
alert(keys);      // "constructor,name,age,job,sayName"
```

ObjectPropertyNamesExample01.htm

Note the inclusion of the non-enumerable `constructor` property in the list of results. Both `Object.keys()` and `Object.getOwnPropertyNames()` may be suitable replacements for using `for-in`. These methods are supported in Internet Explorer 9+, Firefox 4+, Safari 5+, Opera 12+, and Chrome.

Alternate Prototype Syntax

You may have noticed in the previous example that `Person.prototype` had to be typed out for each property and method. To limit this redundancy and to better visually encapsulate functionality on the prototype, it has become more common to simply overwrite the prototype with an object literal that contains all of the properties and methods, as in this example:



Available for
download on
Wrox.com

```
function Person() {
}

Person.prototype = {
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    sayName : function () {
        alert(this.name);
    }
};
```

[PrototypePatternExample07.htm](#)

In this rewritten example, the `Person.prototype` property is set equal to a new object created with an object literal. The end result is the same, with one exception: the `constructor` property no longer points to `Person`. When a function is created, its `prototype` object is created and the `constructor` is automatically assigned. Essentially, this syntax overwrites the default `prototype` object completely, meaning that the `constructor` property is equal to that of a completely new object (the `Object` constructor) instead of the function itself. Although the `instanceof` operator still works reliably, you cannot rely on the `constructor` to indicate the type of object, as this example shows:

```
var friend = new Person();
alert(friend instanceof Object);      //true
alert(friend instanceof Person);     //true
alert(friend.constructor == Person); //false
alert(friend.constructor == Object); //true
```

[PrototypePatternExample07.htm](#)

Here, `instanceof` still returns `true` for both `Object` and `Person`, but the `constructor` property is now equal to `Object` instead of `Person`. If the `constructor`'s value is important, you can set it specifically back to the appropriate value, as shown here:

```
function Person() {
}

Person.prototype = {
    constructor: Person,
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    sayName : function () {
        alert(this.name);
    }
};
```

[PrototypePatternExample08.htm](#)

This code specifically includes a `constructor` property and sets it equal to `Person`, ensuring that the property contains the appropriate value.

Keep in mind that restoring the constructor in this manner creates a property with `[[Enumerable]]` set to `true`. Native `constructor` properties are not enumerable by default, so if you're using an ECMAScript 5-compliant JavaScript engine, you may wish to use `Object.defineProperty()` instead:

```
function Person() {
}

Person.prototype = {
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    sayName : function () {
        alert(this.name);
    }
};

//ECMAScript 5 only - restore the constructor
Object.defineProperty(Person.prototype, "constructor", {
    enumerable: false,
    value: Person
});
```

Dynamic Nature of Prototypes

Since the process of looking up values on a prototype is a search, changes made to the prototype at any point are immediately reflected on instances, even the instances that existed before the change was made. Here's an example:



Available for
download on
[Wrox.com](#)

```
var friend= new Person();

Person.prototype.sayHi = function(){
    alert("hi");
};

friend.sayHi(); // "hi" - works!
```

[PrototypePatternExample09.htm](#)

In this code, an instance of `Person` is created and stored in `friend`. The next statement adds a method called `sayHi()` to `Person.prototype`. Even though the `friend` instance was created prior to this change, it still has access to the new method. This happens because of the loose link between the instance and the prototype. When `friend.sayHi()` is called, the instance is first searched for a property named `sayHi`; when it's not found, the search continues to the prototype. Since the link between the instance and the prototype is simply a pointer, not a copy, the search finds the new `sayHi` property on the prototype and returns the function stored there.

Although properties and methods may be added to the prototype at any time, and they are reflected instantly by all object instances, you cannot overwrite the entire prototype and expect the same behavior.

The `[[Prototype]]` pointer is assigned when the constructor is called, so changing the prototype to a different object severs the tie between the constructor and the original prototype. Remember: the instance has a pointer to only the prototype, not to the constructor. Consider the following:



Available for download on Wrox.com

```

function Person() {
}

var friend = new Person();

Person.prototype = {
    constructor: Person,
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    sayName : function () {
        alert(this.name);
    }
};

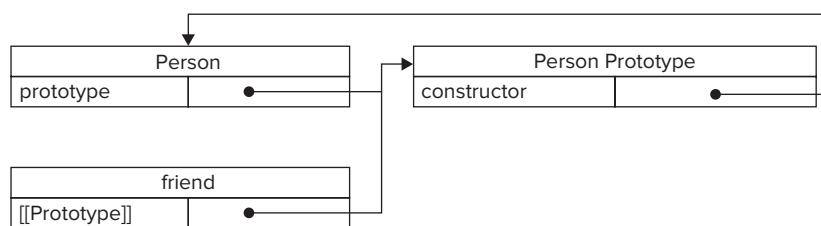
friend.sayName();      //error

```

[PrototypePatternExample10.htm](#)

In this example, a new instance of `Person` is created before the prototype object is overwritten. When `friend.sayName()` is called, it causes an error, because the prototype that `friend` points to doesn't contain a property of that name. Figure 6-3 illustrates why this happens.

Before prototype assignment



After prototype assignment

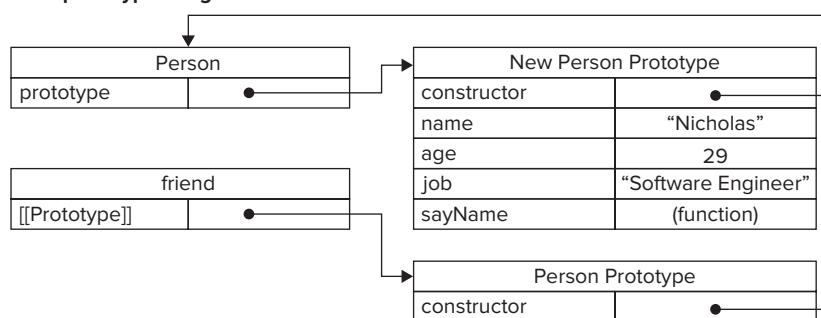


FIGURE 6-3

Overwriting the prototype on the constructor means that new instances will reference the new prototype while any previously existing object instances still reference the old prototype.

Native Object Prototypes

The prototype pattern is important not just for defining custom types but also because it is the pattern used to implement all of the native reference types. Each of these (including `Object`, `Array`, `String`, and so on) has its methods defined on the constructor's prototype. For instance, the `sort()` method can be found on `Array.prototype`, and `substring()` can be found on `String.prototype`, as shown here:



```
alert(typeof Array.prototype.sort);      // "function"
alert(typeof String.prototype.substring); // "function"
```

Through native object prototypes, it's possible to get references to all of the default methods and to define new methods. Native object prototypes can be modified just like custom object prototypes, so methods can be added at any time. For example, the following code adds a method called `startsWith()` to the `String` primitive wrapper:

```
String.prototype.startsWith = function (text) {
    return this.indexOf(text) == 0;
};

var msg = "Hello world!";
alert(msg.startsWith("Hello")); //true
```

[PrototypePatternExample11.htm](#)

The `startsWith()` method in this example returns `true` if some given text occurs at the beginning of a string. The method is assigned to `String.prototype`, making it available to all strings in the environment. Since `msg` is a string, the `String` primitive wrapper is created behind the scenes, making `startsWith()` accessible.



Although possible, it is not recommended to modify native object prototypes in a production environment. This can often cause confusion and create possible name collisions if a method that didn't exist natively in one browser is implemented natively in another. It's also possible to accidentally overwrite native methods.

Problems with Prototypes

The prototype pattern isn't without its faults. For one, it negates the ability to pass initialization arguments into the constructor, meaning that all instances get the same property values by default. Although this is an inconvenience, it isn't the biggest problem with prototypes. The main problem comes with their shared nature.

All properties on the prototype are shared among instances, which is ideal for functions. Properties that contain primitive values also tend to work well, as shown in the previous example, where it's possible to hide the `prototype` property by assigning a property of the same name to the instance. The real problem occurs when a property contains a reference value. Consider the following example:



Available for
download on
Wrox.com

```
function Person() {
}

Person.prototype = {
    constructor: Person,
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    friends : ["Shelby", "Court"],
    sayName : function () {
        alert(this.name);
    }
};

var person1 = new Person();
var person2 = new Person();

person1.friends.push("Van");

alert(person1.friends);      // "Shelby,Court,Van"
alert(person2.friends);      // "Shelby,Court,Van"
alert(person1.friends === person2.friends); // true
```

[PrototypePatternExample12.htm](http://www.wowebook.com/PrototypePatternExample12.htm)

Here, the `Person.prototype` object has a property called `friends` that contains an array of strings. Two instances of `Person` are then created. The `person1.friends` array is altered by adding another string. Because the `friends` array exists on `Person.prototype`, not on `person1`, the changes made are also reflected on `person2.friends` (which points to the same array). If the intention is to have an array shared by all instances, then this outcome is okay. Typically, though, instances want to have their own copies of all properties. This is why the prototype pattern is rarely used on its own.

Combination Constructor/Prototype Pattern

The most common way of defining custom types is to combine the constructor and prototype patterns. The constructor pattern defines instance properties, whereas the prototype pattern defines methods and shared properties. With this approach, each instance ends up with its own copy of the instance properties, but they all share references to methods, conserving memory. This pattern allows arguments to be passed into the constructor as well, effectively combining the best parts of each pattern. The previous example can now be rewritten as follows:

```
function Person(name, age, job) {
    this.name = name;
    this.age = age;
    this.job = job;
```

```

        this.friends = ["Shelby", "Court"];
    }

Person.prototype = {
    constructor: Person,
    sayName : function () {
        alert(this.name);
    }
};

var person1 = new Person("Nicholas", 29, "Software Engineer");
var person2 = new Person("Greg", 27, "Doctor");

person1.friends.push("Van");

alert(person1.friends);      // "Shelby,Court,Van"
alert(person2.friends);      // "Shelby,Court"
alert(person1.friends === person2.friends); //false
alert(person1.sayName === person2.sayName); //true

```

[HybridPatternExample01.htm](#)

Note that the instance properties are now defined solely in the constructor, and the shared property `constructor` and the method `sayName()` are defined on the prototype. When `person1.friends` is augmented by adding a new string, `person2.friends` is not affected, because they each have separate arrays.

The hybrid constructor/prototype pattern is the most widely used and accepted practice for defining custom reference types in ECMAScript. Generally speaking, this is the default pattern to use for defining reference types.

Dynamic Prototype Pattern

Developers coming from other OO languages may find the visual separation between the constructor and the prototype confusing. The dynamic prototype pattern seeks to solve this problem by encapsulating all of the information within the constructor while maintaining the benefits of using both a constructor and a prototype by initializing the prototype inside the constructor, but only if it is needed. You can determine if the prototype needs to be initialized by checking for the existence of a method that should be available. Consider this example:



Available for
download on
[Wrox.com](#)

```

function Person(name, age, job) {

    //properties
    this.name = name;
    this.age = age;
    this.job = job;

    //methods
    if (typeof this.sayName != "function"){

        Person.prototype.sayName = function(){

```

```

        alert(this.name);
    };

}

var friend = new Person("Nicholas", 29, "Software Engineer");
friend.sayName();

```

[DynamicPrototypeExample01.htm](#)

The highlighted section of code inside the constructor adds the `sayName()` method if it doesn't already exist. This block of code is executed only the first time the constructor is called. After that, the prototype has been initialized and doesn't need any further modification. Remember that changes to the prototype are reflected immediately in all instances, so this approach works perfectly. The `if` statement may check for any property or method that will be present once initialized — there's no need for multiple `if` statements to check each property or method; any one will do. This pattern preserves the use of `instanceof` in determining what type of object was created.



You cannot overwrite the prototype using an object literal when using the dynamic prototype pattern. As described previously, overwriting a prototype when an instance already exists effectively cuts off that instance from the new prototype.

Parasitic Constructor Pattern

The parasitic constructor pattern is typically a fallback when the other patterns fail. The basic idea of this pattern is to create a constructor that simply wraps the creation and return of another object while looking like a typical constructor. Here's an example:



Available for download on
Wrox.com

```

function Person(name, age, job){
    var o = new Object();
    o.name = name;
    o.age = age;
    o.job = job;
    o.sayName = function(){
        alert(this.name);
    };
    return o;
}

var friend = new Person("Nicholas", 29, "Software Engineer");
friend.sayName(); //Nicholas

```

[HybridFactoryPatternExample01.htm](#)

In this example, the `Person` constructor creates a new object, initializes it with properties and methods, and then returns the object. This is exactly the same as the factory pattern except that the function is called as a constructor, using the `new` operator. When a constructor doesn't return a value, it returns the new object instance by default. Adding a `return` statement at the end of a constructor allows you to override the value that is returned when the constructor is called.

This pattern allows you to create constructors for objects that may not be possible otherwise. For example, you may want to create a special array that has an extra method. Since you don't have direct access to the `Array` constructor, this pattern works:



```
function SpecialArray() {
    //create the array
    var values = new Array();

    //add the values
    values.push.apply(values, arguments);

    //assign the method
    values.toPipedString = function(){
        return this.join("|");
    };

    //return it
    return values;
}

var colors = new SpecialArray("red", "blue", "green");
alert(colors.toPipedString()); // "red|blue|green"
```

[HybridFactoryPatternExample02.htm](#)

In this example, a constructor called `SpecialArray` is created. In the constructor, a new array is created and initialized using the `push()` method (which has all of the constructor arguments passed in). Then a method called `toPipedString()` is added to the instance, which simply outputs the array values as a pipe-delimited list. The last step is to return the array as the function value. Once that is complete, the `SpecialArray` constructor can be called, passing in the initial values for the array, and `toPipedString()` can be called.

A few important things to note about this pattern: there is no relationship between the returned object and the constructor or the constructor's prototype; the object exists just as if it were created outside of a constructor. Therefore, you cannot rely on the `instanceof` operator to indicate the object type. Because of these issues, this pattern should not be used when other patterns work.

Durable Constructor Pattern

Douglas Crockford coined the term *durable objects* in JavaScript to refer to objects that have no public properties and whose methods don't reference the `this` object. Durable objects are best used

in secure environments (those that forbid the use of `this` and `new`) or to protect data from the rest of the application (as in mashups). A *durable constructor* is a constructor that follows a pattern similar to the parasitic constructor pattern, with two differences: instance methods on the created object don't refer to `this`, and the constructor is never called using the `new` operator. The `Person` constructor from the previous section can be rewritten as a durable constructor like this:

```
function Person(name, age, job){

    //create the object to return
    var o = new Object();

    //optional: define private variables/functions here

    //attach methods
    o.sayName = function(){
        alert(name);
    };

    //return the object
    return o;
}
```

Note that there is no way to access the value of `name` from the returned object. The `sayName()` method has access to it, but nothing else does. The `Person` durable constructor is used as follows:

```
var friend = Person("Nicholas", 29, "Software Engineer");
friend.sayName(); // "Nicholas"
```

The `person` variable is a durable object, and there is no way to access any of its data members without calling a method. Even if some other code adds methods or data members to the object, there is no way to access the original data that was passed into the constructor. Such security makes the durable constructor pattern useful when dealing with secure execution environments such as those provided by ADsafe (www.adsafe.org) or Caja (<http://code.google.com/p/google-caja/>).



As with the parasitic constructor pattern, there is no relationship between the constructor and the object instance, so `instanceof` will not work.

INHERITANCE

The concept most often discussed in relation to OO programming is inheritance. Many OO languages support two types of inheritance: interface inheritance, where only the method signatures are inherited, and implementation inheritance, where actual methods are inherited. Interface inheritance is not possible in ECMAScript, because, as mentioned previously, functions do not have

signatures. Implementation inheritance is the only type of inheritance supported by ECMAScript, and this is done primarily through the use of prototype chaining.

Prototype Chaining

ECMA-262 describes *prototype chaining* as the primary method of inheritance in ECMAScript. The basic idea is to use the concept of prototypes to inherit properties and methods between two reference types. Recall the relationship between constructors, prototypes, and instances: each constructor has a prototype object that points back to the constructor, and instances have an internal pointer to the prototype. What if the prototype were actually an instance of another type? That would mean the prototype itself would have a pointer to a different prototype that, in turn, would have a pointer to another constructor. If that prototype were also an instance of another type, then the pattern would continue, forming a chain between instances and prototypes. This is the basic idea behind prototype chaining.

Implementing prototype chaining involves the following code pattern:



```

function SuperType() {
    this.property = true;
}

SuperType.prototype.getSuperValue = function(){
    return this.property;
};

function SubType() {
    this.subproperty = false;
}

//inherit from SuperType
SubType.prototype = new SuperType();

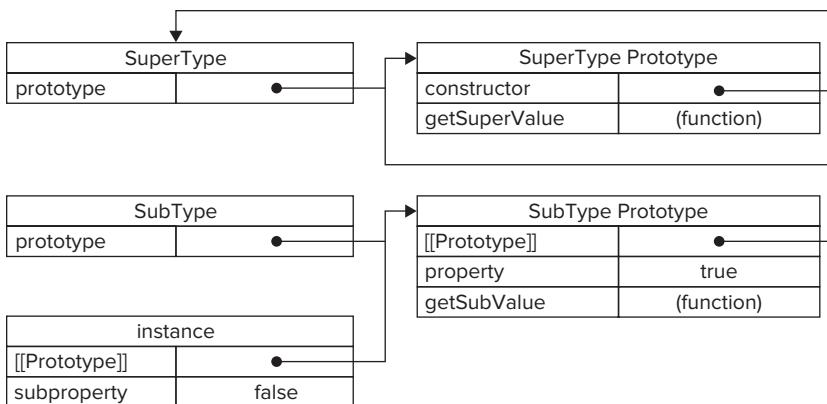
SubType.prototype.getSubValue = function (){
    return this.subproperty;
};

var instance = new SubType();
alert(instance.getSuperValue()); //true

```

[PrototypeChainingExample01.htm](#)

This code defines two types: `SuperType` and `SubType`. Each type has a single property and a single method. The main difference between the two is that `SubType` inherits from `SuperType` by creating a new instance of `SuperType` and assigning it to `SubType.prototype`. This overwrites the original prototype and replaces it with a new object, which means that all properties and methods that typically exist on an instance of `SuperType` now also exist on `SubType.prototype`. After the inheritance takes place, a method is assigned to `SubType.prototype`, adding a new method on top of what was inherited from `SuperType`. The relationship between the instance and both constructors and prototypes is displayed in Figure 6-4.

**FIGURE 6-4**

Instead of using the default prototype of `SubType`, a new prototype is assigned. That new prototype happens to be an instance of `SuperType`, so it not only gets the properties and methods of a `SuperType` instance but also points back to the `SuperType`'s prototype. So `instance` points to `SubType.prototype`, and `SubType.prototype` points to `SuperType.prototype`. Note that the `getSuperValue()` method remains on the `SuperType.prototype` object, but `property` ends up on `SubType.prototype`. That's because `getSuperValue()` is a prototype method, and `property` is an instance property. `SubType.prototype` is now an instance of `SuperType`, so `property` is stored there. Also note that `instance.constructor` points to `SuperType`, because the `constructor` property on the `SubType.prototype` was overwritten.

Prototype chaining extends to the prototype search mechanism described earlier. As you may recall, when a property is accessed in read mode on an instance, the property is first searched for on the instance. If the property is not found, then the search continues to the prototype. When inheritance has been implemented via prototype chaining, that search can continue up the prototype chain. In the previous example, for instance, a call to `instance.getSuperValue()` results in a three-step search: 1) the instance, 2) `SubType.prototype`, and 3) `SuperType.prototype`, where the method is found. The search for properties and methods always continues until the end of the prototype chain is reached.

Default Prototypes

In reality, there is another step in the prototype chain. All reference types inherit from `Object` by default, which is accomplished through prototype chaining. The default prototype for any function is an instance of `Object`, meaning that its internal prototype pointer points to `Object.prototype`. This is how custom types inherit all of the default methods such as `toString()` and `valueOf()`. So the previous example has an extra layer of inheritance. Figure 6-5 shows the complete prototype chain.

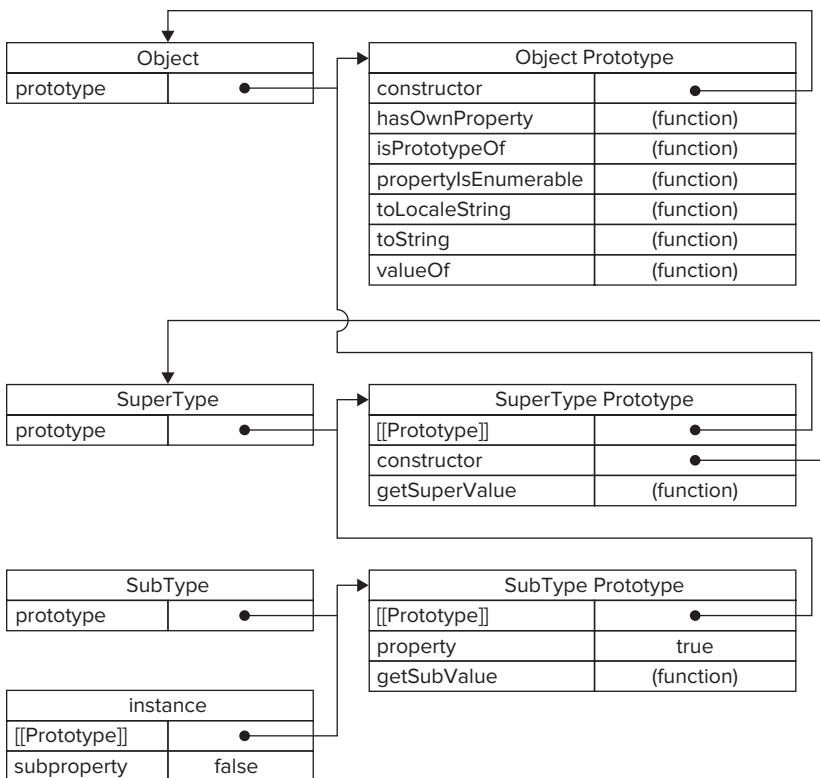


FIGURE 6-5

SubType inherits from SuperType, and SuperType inherits from Object. When instance .toString() is called, the method being called actually exists on Object.prototype.

Prototype and Instance Relationships

The relationship between prototypes and instances is discernible in two ways. The first way is to use the instanceof operator, which returns true whenever an instance is used with a constructor that appears in its prototype chain, as in this example:

```


Available for download on Wrox.com
alert(instance instanceof Object);      //true
alert(instance instanceof SuperType);   //true
alert(instance instanceof SubType);     //true
  
```

[PrototypeChainingExample01.htm](#)

Here, the instance object is technically an instance of Object, SuperType, and SubType because of the prototype chain relationship. The result is that instanceof returns true for all of these constructors.

The second way to determine this relationship is to use the `isPrototypeOf()` method. Each prototype in the chain has access to this method, which returns `true` for an instance in the chain, as in this example:

```
alert(Object.prototype.isPrototypeOf(instance)); //true
alert(SuperType.prototype.isPrototypeOf(instance)); //true
alert(SubType.prototype.isPrototypeOf(instance)); //true
```

[PrototypeChainingExample01.htm](#)

Working with Methods

Often a subtype will need to either override a supertype method or introduce new methods that don't exist on the supertype. To accomplish this, the methods must be added to the prototype after the prototype has been assigned. Consider this example:



Available for
download on
Wrox.com

```
function SuperType() {
    this.property = true;
}

SuperType.prototype.getSuperValue = function(){
    return this.property;
};

function SubType() {
    this.subproperty = false;
}

//inherit from SuperType
SubType.prototype = new SuperType();

//new method
SubType.prototype.getSubValue = function (){
    return this.subproperty;
};

//override existing method
SubType.prototype.getSuperValue = function (){
    return false;
};

var instance = new SubType();
alert(instance.getSuperValue()); //false
```

[PrototypeChainingExample02.htm](#)

In this code, the highlighted area shows two methods. The first is `getSubValue()`, which is a new method on the `SubType`. The second is `getSuperValue()`, which already exists in the prototype chain but is being shadowed here. When `getSuperValue()` is called on an instance of `SubType`, it will call this one, but instances of `SuperType` will still call the original. The important thing to note

is that both of the methods are defined after the prototype has been assigned as an instance of `SuperType`.

Another important thing to understand is that the object literal approach to creating prototype methods cannot be used with prototype chaining, because you end up overwriting the chain. Here's an example:



```
function SuperType() {
    this.property = true;
}

SuperType.prototype.getSuperValue = function(){
    return this.property;
};

function SubType() {
    this.subproperty = false;
}

//inherit from SuperType
SubType.prototype = new SuperType();

//try to add new methods - this nullifies the previous line
SubType.prototype = {
    getSubValue : function (){
        return this.subproperty;
    },
    someOtherMethod : function (){
        return false;
    }
};

var instance = new SubType();
alert(instance.getSuperValue()); //error!
```

[PrototypeChainingExample03.htm](#)

In this code, the prototype is reassigned to be an object literal after it was already assigned to be an instance of `SuperType`. The prototype now contains a new instance of `Object` instead of an instance of `SuperType`, so the prototype chain has been broken — there is no relationship between `SubType` and `SuperType`.

Problems with Prototype Chaining

Even though prototype chaining is a powerful tool for inheritance, it is not without its issues. The major issue revolves around prototypes that contain reference values. Recall from earlier that prototype properties containing reference values are shared with all instances; this is why properties are typically defined within the constructor instead of on the prototype. When implementing inheritance using prototypes, the prototype actually becomes an instance of another type, meaning that what once were instance properties are now prototype properties. The issue is highlighted by the following example:



```

function SuperType() {
    this.colors = ["red", "blue", "green"];
}

function SubType() {
}

//inherit from SuperType
SubType.prototype = new SuperType();

var instance1 = new SubType();
instance1.colors.push("black");
alert(instance1.colors);      // "red,blue,green,black"

var instance2 = new SubType();
alert(instance2.colors);      // "red,blue,green,black"

```

[PrototypeChainingExample04.htm](#)

In this example, the `SuperType` constructor defines a property `colors` that contains an array (a reference value). Each instance of `SuperType` has its own `colors` property containing its own array. When `SubType` inherits from `SuperType` via prototype chaining, `SubType.prototype` becomes an instance of `SuperType` and so it gets its own `colors` property, which is akin to specifically creating `SubType.prototype.colors`. The end result: all instances of `SubType` share a `colors` property. This is indicated as the changes made to `instance1.colors` are reflected on `instance2.colors`.

A second issue with prototype chaining is that you cannot pass arguments into the supertype constructor when the subtype instance is being created. In fact, there is no way to pass arguments into the supertype constructor without affecting all of the object instances. Because of this and the aforementioned issue with reference values on the prototype, prototype chaining is rarely used alone.

Constructor Stealing

In an attempt to solve the inheritance problem with reference values on prototypes, developers began using a technique called *constructor stealing* (also sometimes called object masquerading or classical inheritance). The basic idea is quite simple: call the supertype constructor from within the subtype constructor. Keeping in mind that functions are simply objects that execute code in a particular context, the `apply()` and `call()` methods can be used to execute a constructor on the newly created object, as in this example:

```

function SuperType() {
    this.colors = ["red", "blue", "green"];
}

function SubType() {
    //inherit from SuperType
    SuperType.call(this);
}

var instance1 = new SubType();
instance1.colors.push("black");

```

```

alert(instance1.colors);      // "red,blue,green,black"

var instance2 = new SubType();
alert(instance2.colors);      // "red,blue,green"

```

[ConstructorStealingExample01.htm](#)

The highlighted lines in this example show the single call that is used in constructor stealing. By using the `call()` method (or alternately, `apply()`), the `SuperType` constructor is called in the context of the newly created instance of `SubType`. Doing this effectively runs all of the object-initialization code in the `SuperType()` function on the new `SubType` object. The result is that each instance has its own copy of the `colors` property.

Passing Arguments

One advantage that constructor stealing offers over prototype chaining is the ability to pass arguments into the supertype constructor from within the subtype constructor. Consider the following:



```

function SuperType(name) {
    this.name = name;
}

function SubType() {
    //inherit from SuperType passing in an argument
    SuperType.call(this, "Nicholas");

    //instance property
    this.age = 29;
}

var instance = new SubType();
alert(instance.name);      //"Nicholas";
alert(instance.age);       //29

```

[ConstructorStealingExample02.htm](#)

In this code, the `SuperType` constructor accepts a single argument, `name`, which is simply assigned to a property. A value can be passed into the `SuperType` constructor when called from within the `SubType` constructor, effectively setting the `name` property for the `SubType` instance. To ensure that the `SuperType` constructor doesn't overwrite those properties, you can define additional properties on the subtype after the call to the supertype constructor.

Problems with Constructor Stealing

The downside to using constructor stealing exclusively is that it introduces the same problems as the constructor pattern for custom types: methods must be defined inside the constructor, so there's no function reuse. Furthermore, methods defined on the supertype's prototype are not accessible on the subtype, so all types can use only the constructor pattern. Because of these issues, constructor stealing is rarely used on its own.

Combination Inheritance

Combination inheritance (sometimes also called pseudoclassical inheritance) combines prototype chaining and constructor stealing to get the best of each approach. The basic idea is to use prototype chaining to inherit properties and methods on the prototype and to use constructor stealing to inherit instance properties. This allows function reuse by defining methods on the prototype and allows each instance to have its own properties. Consider the following:



Available for
download on
Wrox.com

```
function SuperType(name) {
    this.name = name;
    this.colors = ["red", "blue", "green"];
}

SuperType.prototype.sayName = function(){
    alert(this.name);
};

function SubType(name, age) {

    //inherit properties
    SuperType.call(this, name);

    this.age = age;
}

//inherit methods
SubType.prototype = new SuperType();

SubType.prototype.sayAge = function(){
    alert(this.age);
};

var instance1 = new SubType("Nicholas", 29);
instance1.colors.push("black");
alert(instance1.colors); // "red,blue,green,black"
instance1.sayName(); // "Nicholas";
instance1.sayAge(); // 29

var instance2 = new SubType("Greg", 27);
alert(instance2.colors); // "red,blue,green"
instance2.sayName(); // "Greg";
instance2.sayAge(); // 27
```

[CombinationInheritanceExample01.htm](#)

In this example, the `SuperType` constructor defines two properties, `name` and `colors`, and the `SuperType` prototype has a single method called `sayName()`. The `SubType` constructor calls the `SuperType` constructor, passing in the `name` argument, and defines its own property called `age`. Additionally, the `SubType` prototype is assigned to be an instance of `SuperType`, and then a new method called `sayAge()` is defined. With this code, it's then possible to create two separate instances of `SubType` that have their own properties, including the `colors` property, but all use the same methods.

Addressing the downsides of both prototype chaining and constructor stealing, combination inheritance is the most frequently used inheritance pattern in JavaScript. It also preserves the behavior of `instanceof` and `isPrototypeOf()` for identifying the composition of objects.

Prototypal Inheritance

In 2006, Douglas Crockford wrote an article titled “Prototypal Inheritance in JavaScript” in which he introduced a method of inheritance that didn’t involve the use of strictly defined constructors. His premise was that prototypes allow you to create new objects based on existing objects without the need for defining custom types. The function he introduced to this end is as follows:



```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}
```

The `object()` function creates a temporary constructor, assigns a given object as the constructor’s prototype, and returns a new instance of the temporary type. Essentially, `object()` performs a shadow copy of any object that is passed into it. Consider the following:

```
var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

var anotherPerson = object(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");

var yetAnotherPerson = object(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");

alert(person.friends); // "Shelby,Court,Van,Rob,Barbie"
```

[PrototypalInheritanceExample01.htm](#)

This is the way Crockford advocates using prototypal inheritance: you have an object that you want to use as the base of another object. That object should be passed into `object()`, and the resulting object should be modified accordingly. In this example, the `person` object contains information that should be available on another object, so it is passed into the `object()` function, which returns a new object. The new object has `person` as its prototype, meaning that it has both a primitive value property and a reference value property on its prototype. This also means that `person.friends` is shared not only by `person` but also with `anotherPerson` and `yetAnotherPerson`. Effectively, this code has created two clones of `person`.

ECMAScript 5 formalized the concept of prototypal inheritance by adding the `Object.create()` method. This method accepts two arguments, an object to use as the prototype for a new object and

an optional object defining additional properties to apply to the new object. When used with one argument, `Object.create()` behaves the same as the `object()` method:



```
var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

var anotherPerson = Object.create(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");

var yetAnotherPerson = Object.create(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");

alert(person.friends); // "Shelby,Court,Van,Rob,Barbie"
```

[PrototypalInheritanceExample02.htm](#)

The second argument for `Object.create()` is in the same format as the second argument for `Object.defineProperties()`: each additional property to define is specified along with its descriptor. Any properties specified in this manner will shadow properties of the same name on the prototype object. For example:

```
var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

var anotherPerson = Object.create(person, {
    name: {
        value: "Greg"
    }
});

alert(anotherPerson.name); // "Greg"
```

[PrototypalInheritanceExample03.htm](#)

The `Object.create()` method is supported in Internet Explorer 9+, Firefox 4+, Safari 5+, Opera 12+, and Chrome.

Prototypal inheritance is useful when there is no need for the overhead of creating separate constructors, but you still need an object to behave similarly to another. Keep in mind that properties containing reference values will always share those values, similar to using the prototype pattern.

Parasitic Inheritance

Closely related to prototypal inheritance is the concept of *parasitic inheritance*, another pattern popularized by Crockford. The idea behind parasitic inheritance is similar to that of the parasitic constructor and factory patterns: create a function that does the inheritance, augments the object

in some way, and then returns the object as if it did all the work. The basic parasitic inheritance pattern looks like this:

```
function createAnother(original){
    var clone = object(original);           //create a new object by calling a function
    clone.sayHi = function(){               //augment the object in some way
        alert("hi");
    };
    return clone;                          //return the object
}
```

In this code, the `createAnother()` function accepts a single argument, which is the object to base a new object on. This object, `original`, is passed into the `object()` function, and the result is assigned to `clone`. Next, the `clone` object is changed to have a new method called `sayHi()`. The last step is to return the object. The `createAnother()` function can be used in the following way:

```
var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

var anotherPerson = createAnother(person);
anotherPerson.sayHi(); // "hi"
```

The code in this example returns a new object based on `person`. The `anotherPerson` object has all of the properties and methods of `person` but adds a new method called `sayHi()`.

Parasitic inheritance is another pattern to use when you are concerned primarily with objects and not with custom types and constructors. The `object()` method is not required for parasitic inheritance; any function that returns a new object fits the pattern.



Keep in mind that adding functions to objects using parasitic inheritance leads to inefficiencies related to function reuse, similar to the constructor pattern.

Parasitic Combination Inheritance

Combination inheritance is the most often-used pattern for inheritance in JavaScript, though it is not without its inefficiencies. The most inefficient part of the pattern is that the supertype constructor is always called twice: once to create the subtype's prototype, and once inside the subtype constructor. Essentially, the subtype prototype ends up with all of the instance properties of a supertype object, only to have it overwritten when the subtype constructor executes. Consider the combination inheritance example again:

```
function SuperType(name) {
    this.name = name;
    this.colors = ["red", "blue", "green"];
}

SuperType.prototype.sayName = function() {
    alert(this.name);
```

```

};

function SubType(name, age) {
    SuperType.call(this, name);           //second call to SuperType()

    this.age = age;
}

SubType.prototype = new SuperType();      //first call to SuperType()
SubType.prototype.constructor = SubType;
SubType.prototype.sayAge = function() {
    alert(this.age);
};

```

The highlighted lines of code indicate when `SuperType` constructor is executed. When this code is executed, `SubType.prototype` ends up with two properties: `name` and `colors`. These are instance properties for `SuperType`, but they are now on the `SubType`'s prototype. When the `SubType` constructor is called, the `SuperType` constructor is also called, which creates instance properties `name` and `colors` on the new object that mask the properties on the prototype. Figure 6-6 illustrates this process.

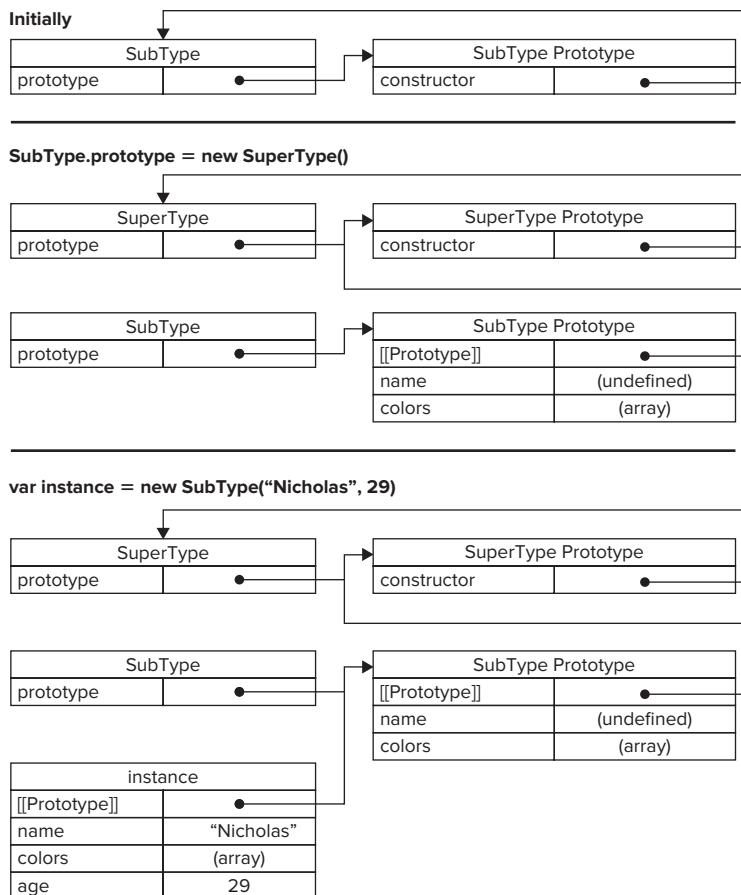


FIGURE 6-6

As you can see, there are two sets of `name` and `colors` properties: one on the instance and one on the `SubType` prototype. This is the result of calling the `SuperType` constructor twice. Fortunately, there is a way around this.

Parasitic combination inheritance uses constructor stealing to inherit properties but uses a hybrid form of prototype chaining to inherit methods. The basic idea is this: instead of calling the supertype constructor to assign the subtype's prototype, all you need is a copy of the supertype's prototype. Essentially, use parasitic inheritance to inherit from the supertype's prototype and then assign the result to the subtype's prototype. The basic pattern for parasitic combination inheritance is as follows:

```
function inheritPrototype(subType, superType) {
    var prototype = object(superType.prototype);           //create object
    prototype.constructor = subType;                      //augment object
    subType.prototype = prototype;                        //assign object
}
```

The `inheritPrototype()` function implements very basic parasitic combination inheritance. This function accepts two arguments: the subtype constructor and the supertype constructor. Inside the function, the first step is to create a clone of the supertype's prototype. Next, the `constructor` property is assigned onto `prototype` to account for losing the default `constructor` property when the prototype is overwritten. Finally, the subtype's prototype is assigned to the newly created object. A call to `inheritPrototype()` can replace the subtype prototype assignment in the previous example, as shown here:



```
function SuperType(name) {
    this.name = name;
    this.colors = ["red", "blue", "green"];
}

SuperType.prototype.sayName = function() {
    alert(this.name);
};

function SubType(name, age) {
    SuperType.call(this, name);

    this.age = age;
}

inheritPrototype(SubType, SuperType);

SubType.prototype.sayAge = function() {
    alert(this.age);
};
```

<ParasiticCombinationInheritanceExample01.htm>

This example is more efficient in that the `SuperType` constructor is being called only one time, avoiding having unnecessary and unused properties on `SubType.prototype`. Furthermore, the

prototype chain is kept intact, so both `instanceof` and `isPrototypeOf()` behave as they would normally. Parasitic combination inheritance is considered the most optimal inheritance paradigm for reference types.



The Yahoo! User Interface (YUI) library was the first to include parasitic combination inheritance in a widely distributed JavaScript library via the `Y.extend()` method. For more information on YUI, visit <http://yuilibrary.com/>.

SUMMARY

ECMAScript supports object-oriented (OO) programming without the use of classes or interfaces. Objects are created and augmented at any point during code execution, making objects into dynamic rather than strictly defined entities. In place of classes, the following patterns are used for the creation of objects:

- The factory pattern uses a simple function that creates an object, assigns properties and methods, and then returns the object. This pattern fell out of favor when the constructor pattern emerged.
- Using the constructor pattern, it's possible to define custom reference types that can be created using the `new` operator in the same way as built-in object instances are created. The constructor pattern does have a downside, however, in that none of its members are reused, including functions. Since functions can be written in a loosely typed manner, there's no reason they cannot be shared by multiple object instances.
- The prototype pattern takes this into account, using the constructor's `prototype` property to assign properties and methods that should be shared. The combination constructor/prototype pattern uses the constructor to define instance properties and the prototype pattern to define shared properties and methods.

Inheritance in JavaScript is implemented primarily using the concept of prototype chaining. Prototype chaining involves assigning a constructor's `prototype` to be an instance of another type. In doing so, the subtype assumes all of the properties and methods of the supertype in a manner similar to class-based inheritance. The problem with prototype chaining is that all of the inherited properties and methods are shared among object instances, making it ill-suited for use on its own. The constructor stealing pattern avoids these issues, calling the supertype's constructor from inside of the subtype's constructor. This allows each instance to have its own properties but forces the types to be defined using only the constructor pattern. The most popular pattern of inheritance is combination inheritance, which uses prototype chaining to inherit shared properties and methods and uses constructor stealing to inherit instance properties.

There are also the following alternate inheritance patterns:

- Prototypal inheritance implements inheritance without the need for predefined constructors, essentially performing a shallow clone operation on a given object. The result of the operation then may be augmented further.
- Closely related is parasitic inheritance, which is a pattern for creating an object based on another object or some information, augmenting it, and returning it. This pattern has also been repurposed for use with combination inheritance to remove the inefficiencies related to the number of times the supertype constructor is called.
- Parasitic combination inheritance is considered the most efficient way to implement type-based inheritance.

7

Function Expressions

WHAT'S IN THIS CHAPTER?

- Function expression characteristics
- Recursion with functions
- Private variables using closures

One of the more powerful, and often confusing, parts of JavaScript is function expressions. As mentioned in Chapter 5, there are two ways to define a function: by function declaration and by function expression. The first, function declaration, has the following form:

```
function functionName(arg0, arg1, arg2) {  
    //function body  
}
```

The name of the function follows the `function` keyword, and this is how the function's name is assigned. Firefox, Safari, Chrome, and Opera all feature a nonstandard `name` property on functions exposing the assigned name. This value is always equivalent to the identifier that immediately follows the `function` keyword:



Available for
download on
Wrox.com

```
//works only in Firefox, Safari, Chrome, and Opera  
alert(functionName.name);      //functionName"
```

FunctionNameExample01.htm



```
sayHi();
function sayHi(){
    alert("Hi!");
}
```

FunctionDeclarationHoisting01.htm

This example doesn't throw an error because the function declaration is read first before the code begins to execute.

The second way to create a function is by using a function expression. Function expressions have several forms. The most common is as follows:

```
var functionName = function(arg0, arg1, arg2){
    //function body
};
```

This pattern of function expression looks like a normal variable assignment. A function is created and assigned to the variable `functionName`. The created function is considered to be an *anonymous function*, because it has no identifier after the `function` keyword. (Anonymous functions are also sometimes called *lambda functions*.) This means the `name` property is the empty string.

Function expressions act like other expressions and, therefore, must be assigned before usage. The following causes an error:

```
sayHi();      //error - function doesn't exist yet
var sayHi = function(){
    alert("Hi!");
};
```

Understanding function hoisting is key to understanding the differences between function declarations and function expressions. For instance, the result of the following code may be surprising:

```
//never do this!
if(condition){
    function sayHi(){
        alert("Hi!");
    }
} else {
    function sayHi(){
        alert("Yo!");
    }
}
```

FunctionDeclarationsErrorExample01.htm

The code seems to indicate that if `condition` is `true`, use one definition for `sayHi()`; otherwise, use a different definition. In fact, this is not valid syntax in ECMAScript, so JavaScript engines try to error correct into an appropriate state. The problem is that browsers don't consistently error correct in this case. Most browsers return the second declaration regardless of `condition`; Firefox returns the first when `condition` is `true`. This pattern is dangerous and should not be used. It is perfectly fine, however, to use function expressions in this way:

```
//this is okay
var sayHi;

if(condition){
    sayHi = function(){
        alert("Hi!");
    };
} else {
    sayHi = function(){
        alert("Yo!");
    };
}
```

This example behaves the way you would expect, assigning the correct function expression to the variable `sayHi` based on `condition`.

The ability to create functions for assignment to variables also allows you to return functions as the value of other functions. Recall the following `createComparisonFunction()` example from Chapter 5:

```
function createComparisonFunction(propertyName) {

    return function(object1, object2){
        var value1 = object1[propertyName];
        var value2 = object2[propertyName];

        if (value1 < value2){
            return -1;
        } else if (value1 > value2){
            return 1;
        } else {
            return 0;
        }
    };
}
```

`createComparisonFunction()` returns an anonymous function. The returned function will, presumably, be either assigned to a variable or otherwise called, but within `createComparisonFunction()` it is anonymous. Any time a function is being used as a value, it is a function expression. However, these are not the only uses for function expressions, as the rest of this chapter will show.

RECURSION

A *recursive function* typically is formed when a function calls itself by name, as in the following example:



```
function factorial(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * factorial(num-1);
    }
}
```

RecursionExample01.htm

This is the classic recursive factorial function. Although this works initially, it's possible to prevent it from functioning by running the following code immediately after it:

```
var anotherFactorial = factorial;
factorial = null;
alert(anotherFactorial(4)); //error!
```

RecursionExample01.htm

Here, the `factorial()` function is stored in a variable called `anotherFactorial`. The `factorial` variable is then set to `null`, so only one reference to the original function remains. When `anotherFactorial()` is called, it will cause an error, because it will try to execute `factorial()`, which is no longer a function. Using `argumentscallee` can alleviate this problem.

Recall that `argumentscallee` is a pointer to the function being executed and, as such, can be used to call the function recursively, as shown here:

```
function factorial(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * argumentscallee(num-1);
    }
}
```

RecursionExample02.htm

Changing the highlighted line to use `argumentscallee` instead of the function name ensures that this function will work regardless of how it is accessed. It's advisable to always use `argumentscallee` of the function name whenever you're writing recursive functions.

The value of `argumentscallee` is not accessible to a script running in strict mode and will cause an error when attempts are made to read it. Instead, you can use *named function expressions* to achieve the same result. For example:

```
var factorial = (function f(num) {
    if (num <= 1) {
        return 1;
    } else {
```

```

        return num * f(num-1);
    }
}) ;

```

In this code, a named function expression `f()` is created and assigned to the variable `factorial`. The name `f` remains the same even if the function is assigned to another variable, so the recursive call will always execute correctly. This pattern works in both nonstrict mode and strict mode.

CLOSURES

The terms *anonymous functions* and *closures* are often incorrectly used interchangeably. *Closures* are functions that have access to variables from another function's scope. This is often accomplished by creating a function inside a function, as in the following highlighted lines from the previous `createComparisonFunction()` example:

```

function createComparisonFunction(propertyName) {

    return function(object1, object2){
        var value1 = object1[propertyName];
        var value2 = object2[propertyName];

        if (value1 < value2){
            return -1;
        } else if (value1 > value2){
            return 1;
        } else {
            return 0;
        }
    };
}

```

The highlighted lines in this example are part of the inner function (an anonymous function) that is accessing a variable (`propertyName`) from the outer function. Even after the inner function has been returned and is being used elsewhere, it has access to that variable. This occurs because the inner function's scope chain includes the scope of `createComparisonFunction()`. To understand why this is possible, consider what happens when a function is first called.

Chapter 4 introduced the concept of a scope chain. The details of how scope chains are created and used are important for a good understanding of closures. When a function is called, an execution context is created, and its scope chain is created. The activation object for the function is initialized with values for arguments and any named arguments. The outer function's activation object is the second object in the scope chain. This process continues for all containing functions until the scope chain terminates with the global execution context.

As the function executes, variables are looked up in the scope chain for the reading and writing of values. Consider the following:

```

function compare(value1, value2){
    if (value1 < value2){
        return -1;
    } else if (value1 > value2){

```

```

        return 1;
    } else {
        return 0;
    }
}

var result = compare(5, 10);

```

This code defines a function named `compare()` that is called in the global execution context. When `compare()` is called for the first time, a new activation object is created that contains `arguments`, `value1`, and `value2`. The global execution context's variable object is next in the `compare()` execution context's scope chain, which contains `this`, `result`, and `compare`. Figure 7-1 illustrates this relationship.

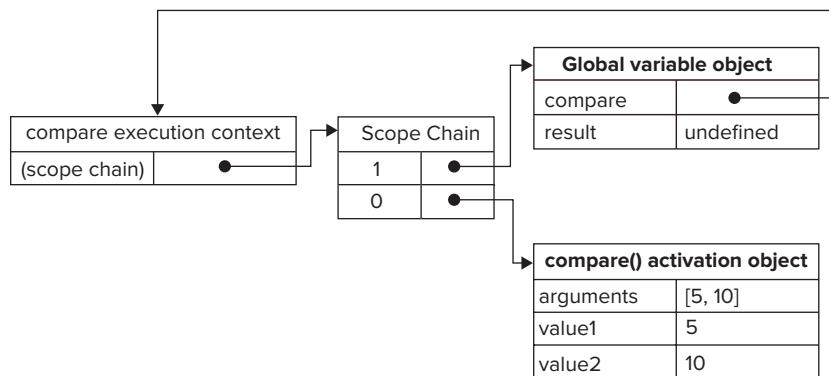


FIGURE 7-1

Behind the scenes, an object represents the variables in each execution context. The global context's variable object always exists, whereas local context variable objects, such as the one for `compare()`, exist only while the function is being executed. When `compare()` is defined, its scope chain is created, preloaded with the global variable object, and saved to the internal `[[Scope]]` property. When the function is called, an execution context is created and its scope chain is built up by copying the objects in the function's `[[Scope]]` property. After that, an activation object (which also acts as a variable object) is created and pushed to the front of the context's scope chain. In this example, that means the `compare()` function's execution context has two variable objects in its scope chain: the local activation object and the global variable object. Note that the scope chain is essentially a list of pointers to variable objects and does not physically contain the objects.

Whenever a variable is accessed inside a function, the scope chain is searched for a variable with the given name. Once the function has completed, the local activation object is destroyed, leaving only the global scope in memory. Closures, however, behave differently.

A function that is defined inside another function adds the containing function's activation object into its scope chain. So, in `createComparisonFunction()`, the anonymous function's scope chain actually contains a reference to the activation object for `createComparisonFunction()`. Figure 7-2 illustrates this relationship when the following code is executed:

```
var compare = createComparisonFunction("name");
var result = compare({ name: "Nicholas" }, { name: "Greg" });
```

When the anonymous function is returned from `createComparisonFunction()`, its scope chain has been initialized to contain the activation object from `createComparisonFunction()` and the global variable object. This gives the anonymous function access to all of the variables from `createComparisonFunction()`. Another interesting side effect is that the activation object from `createComparisonFunction()` cannot be destroyed once the function finishes executing, because a reference still exists in the anonymous function's scope chain. After `createComparisonFunction()` completes, the scope chain for its execution context is destroyed, but its activation object will remain in memory until the anonymous function is destroyed, as in the following:

```
//create function
var compareNames = createComparisonFunction("name");

//call function
var result = compareNames({ name: "Nicholas" }, { name: "Greg" });

//dereference function - memory can now be reclaimed
compareNames = null;
```

Here, the comparison function is created and stored in the variable `compareNames`. Setting `compareNames` equal to `null` dereferences the function and allows the garbage collection routine to clean it up. The scope chain will then be destroyed and, all of the scopes (except the global scope) can be destroyed safely. Figure 7-2 shows the scope-chain relationships that occur when `compareNames()` is called in this example.

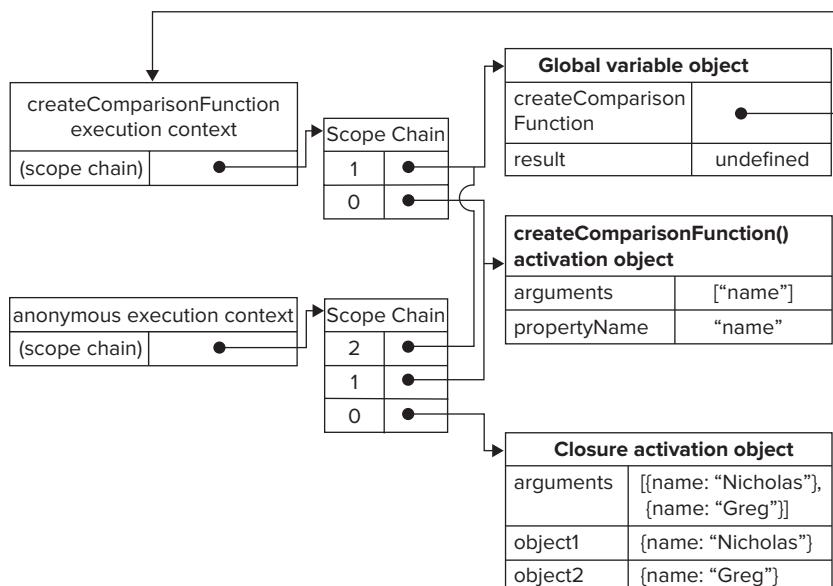


FIGURE 7-2



Since closures carry with them the containing function's scope, they take up more memory than other functions. Overuse of closures can lead to excess memory consumption, so it's recommended you use them only when absolutely necessary. Optimizing JavaScript engines, such as V8, make attempts to reclaim memory that is trapped because of closures, but it's still recommended to be careful when using closures.

Closures and Variables

There is one notable side effect of this scope-chain configuration. The closure always gets the last value of any variable from the containing function. Remember that the closure stores a reference to the entire variable object, not just to a particular variable. This issue is illustrated clearly in the following example:



Available for download on
Wrox.com

```
function createFunctions(){
    var result = new Array();

    for (var i=0; i < 10; i++){
        result[i] = function(){
            return i;
        };
    }

    return result;
}
```

ClosureExample01.htm

This function returns an array of functions. It seems that each function should just return the value of its index, so the function in position 0 returns 0, the function in position 1 returns 1, and so on. In reality, every function returns 10. Since each function has the `createFunctions()` activation object in its scope chain, they are all referring to the same variable, `i`. When `createFunctions()` finishes running, the value of `i` is 10, and since every function references the same variable object in which `i` exists, the value of `i` inside each function is 10. You can, however, force the closures to act appropriately by creating another anonymous function, as follows:

```
function createFunctions(){
    var result = new Array();

    for (var i=0; i < 10; i++){
        result[i] = function(num){
            return function(){
                return num;
            };
        }(i);
    }
}
```

```

        }
        return result;
    }
}

```

[ClosureExample02.htm](#)

With this version of `createFunctions()`, each function returns a different number. Instead of assigning a closure directly into the array, an anonymous function is defined and called immediately. The anonymous function has one argument, `num`, which is the number that the result function should return. The variable `i` is passed in as an argument to the anonymous function. Since function arguments are passed by value, the current value of `i` is copied into the argument `num`. Inside the anonymous function, a closure that accesses `num` is created and returned. Now each function in the `result` array has its own copy of `num` and thus can return separate numbers.

The this Object

Using the `this` object inside closures introduces some complex behaviors. The `this` object is bound at runtime based on the context in which a function is executed: when used inside global functions, `this` is equal to `window` in nonstrict mode and `undefined` in strict mode, whereas `this` is equal to the object when called as an object method. Anonymous functions are not bound to an object in this context, meaning the `this` object points to `window` unless executing in strict mode (where `this` is `undefined`). Because of the way closures are written, however, this fact is not always obvious.

Consider the following:



Available for
download on
[Wrox.com](#)

```

var name = "The Window";
var object = {
    name : "My Object",
    getNameFunc : function(){
        return function(){
            return this.name;
        };
    }
};

alert(object.getNameFunc()()); // "The Window" (in non-strict mode)

```

[ThisObjectExample01.htm](#)

Here, a global variable called `name` is created along with an object that also contains a property called `name`. The object contains a method, `getNameFunc()`, that returns an anonymous function, which returns `this.name`. Since `getNameFunc()` returns a function, calling `object.getNameFunc()()` immediately calls the function that is returned, which returns a string. In this case, however, it returns "The Window", which is the value of the global `name` variable. Why didn't the anonymous function pick up the containing scope's `this` object?

Remember that each function automatically gets two special variables as soon as the function is called: `this` and `arguments`. An inner function can never access these variables directly from an outer function. It is possible to allow a closure access to a different `this` object by storing it in another variable that the closure can access, as in this example:



Available for download on Wrox.com

```

var name = "The Window";

var object = {
    name : "My Object",

    getNameFunc : function(){
        var that = this;
        return function(){
            return that.name;
        };
    }
};

alert(object.getNameFunc()()); // "My Object"

```

ThisObjectExample02.htm

The two highlighted lines show the difference between this example and the previous one. Before defining the anonymous function, a variable named `that` is assigned equal to the `this` object. When the closure is defined, it has access to `that`, since it is a uniquely named variable in the containing function. Even after the function is returned, `that` is still bound to `object`, so calling `object.getNameFunc()()` returns "My Object".



Both `this` and `arguments` behave in this way. If you want access to a containing scope's `arguments` object, you'll need to save a reference into another variable that the closure can access.

There are a few special cases where the value of `this` may not end up as the value you expect. Consider the following modification to the previous example:

```

var name = "The Window";

var object = {
    name : "My Object",

    getName: function(){
        return this.name;
    }
};

```

The `getName()` method simply returns the value of `this.name`. Here are various ways to call `object.getName()` and the results:

```
object.getName();      // "My Object"
(object.getName)();   // "My Object"
(object.getName = object.getName)(); // "The Window" in non-strict mode
```

[ThisObjectExample03.htm](#)

The first line calls `object.getName()` in the way you normally would and so returns "My Object", as `this.name` is the same as `object.name`. The second line places parentheses around `object.getName` before calling it. While this might seem to be a reference just to the function, the `this` value is maintained, because `object.getName` and `(object.getName)` are defined to be equivalent. The third line performs an assignment and then calls the result. Because the value of this assignment expression is the function itself, the `this` value is not maintained, and so "The Window" is returned.

It's unlikely that you'll intentionally use the patterns in lines two or three, but it is helpful to know that the value of `this` can change in unexpected ways when syntax is changed slightly.

Memory Leaks

The way closures work causes particular problems in Internet Explorer prior to version 9 because of the different garbage-collection routines used for JScript objects versus COM objects (discussed in Chapter 4). Storing a scope in which an HTML element is stored effectively ensures that the element cannot be destroyed. Consider the following:

```
function assignHandler(){
    var element = document.getElementById("someElement");
    element.onclick = function(){
        alert(element.id);
    };
}
```

This code creates a closure as an event handler on `element`, which in turn creates a circular reference (events are discussed in Chapter 13). The anonymous function keeps a reference to the `assignHandler()` function's activation object, which prevents the reference count for `element` from being decremented. As long as the anonymous function exists, the reference count for `element` will be at least 1, which means the memory will never be reclaimed. This situation can be remedied by changing the code slightly, as shown here:

```
function assignHandler(){
    var element = document.getElementById("someElement");
    var id = element.id;

    element.onclick = function(){
        alert(id);
    };

    element = null;
}
```

In this version of the code, a copy of `element`'s ID is stored in a variable that is used in the closure, eliminating the circular reference. That step alone is not enough, however, to prevent the memory problem. Remember: the closure has a reference to the containing function's entire activation object, which contains `element`. Even if the closure doesn't reference `element` directly, a reference is still stored in the containing function's activation object. It is necessary, therefore, to set the `element` variable equal to `null`. This dereferences the COM object and decrements its reference count, ensuring that the memory can be reclaimed when appropriate.

MIMICKING BLOCK SCOPE

As mentioned previously, JavaScript has no concept of block-level scoping, meaning variables defined inside of block statements are actually created in the containing function, not within the statement. Consider the following:



```
function outputNumbers(count) {
    for (var i=0; i < count; i++) {
        alert(i);
    }

    alert(i); //count
}
```

BlockScopeExample01.htm

In this function, a `for` loop is defined and the variable `i` is initialized to be equal to 0. For languages such as Java and C++, the variable `i` would be defined only in the block statement representing the `for` loop, so the variable would be destroyed as soon as the loop completed. However, in JavaScript the variable `i` is defined as part of the `outputNumbers()` activation object, meaning it is accessible inside the function from that point on. Even the following errant redeclaration of the variable won't wipe out its value:

```
function outputNumbers(count) {
    for (var i=0; i < count; i++) {
        alert(i);
    }

    var i; //variable redeclared
    alert(i); //count
}
```

BlockScopeExample02.htm

JavaScript will never tell you if you've declared the same variable more than once; it simply ignores all subsequent declarations (though it will honor initializations). Anonymous functions can be used to mimic block scoping and avoid such problems.

The basic syntax of an anonymous function used as a block scope (often called a *private scope*) is as follows:

```
(function(){
    //block code here
})();
```

This syntax defines an anonymous function that is called immediately and is also sometimes called an *immediately invoked function*. What looks like a function declaration is enclosed in parentheses to indicate that it's actually a function expression. This function is then called via the second set of parentheses at the end. If this syntax is confusing, consider the following example:

```
var count = 5;
outputNumbers(count);
```

In this example, a variable `count` is initialized with the value of `5`. Of course, the variable is unnecessary since the value is being passed directly into a function. To make the code more concise, the value `5` can replace the variable `count` when calling the function as follows:

```
outputNumbers(5);
```

This works the same as the previous example because a variable is just a representation of another value, so the variable can be replaced with the actual value, and the code works fine. Now consider the following:

```
var someFunction = function(){
    //block code here
};
someFunction();
```

In this example, a function is defined and then called immediately. An anonymous function is created and assigned to the variable `someFunction`. The function is then called by placing parentheses after the function name, becoming `someFunction()`. Remember in the previous example that the variable `count` could be replaced with its actual value; the same thing can be done here. However, the following won't work:

```
function(){
    //block code here
}(); //error!
```

This code causes a syntax error, because JavaScript sees the `function` keyword as the beginning of a function declaration, and function declarations cannot be followed by parentheses. Function *expressions*, however, *can* be followed by parentheses. To turn the function declaration into a function expression, you need only surround it with parentheses like this:

```
(function(){
    //block code here
})();
```

Available for
download on
Wrox.com

```
function outputNumbers(count){
    (function () {
        for (var i=0; i < count; i++){
            alert(i);
        }
    })();
    alert(i); //causes an error
}
```

[BlockScopeExample03.htm](#)

In this rewritten version of the `outputNumbers()` function, a private scope is inserted around the `for` loop. Any variables defined within the anonymous function are destroyed as soon as it completes execution, so the variable `i` is used in the loop and then destroyed. The `count` variable is accessible inside the private scope because the anonymous function is a closure, with full access to the containing scope's variables.

This technique is often used in the global scope outside of functions to limit the number of variables and functions added to the global scope. Typically you want to avoid adding variables and functions to the global scope, especially in large applications with multiple developers, to avoid naming collisions. Private scopes allow every developer to use his or her own variables without worrying about polluting the global scope. Consider this example:

```
(function(){

    var now = new Date();
    if (now.getMonth() == 0 && now.getDate() == 1){
        alert("Happy new year!");
    }
})();
```

Placing this code in the global scope provides functionality for determining if the day is January 1 and, if so, displaying a message to the user. The variable `now` becomes a variable that is local to the anonymous function instead of being created in the global scope.



This pattern limits the closure memory problem, because there is no reference to the anonymous function. Therefore the scope chain can be destroyed immediately after the function has completed.

PRIVATE VARIABLES

Strictly speaking, JavaScript has no concept of private members; all object properties are public. There is, however, a concept of *private variables*. Any variable defined inside a function is considered private since it is inaccessible outside that function. This includes function arguments, local variables, and functions defined inside other functions. Consider the following:

```
function add(num1, num2) {
    var sum = num1 + num2;
    return sum;
}
```

In this function, there are three private variables: `num1`, `num2`, and `sum`. These variables are accessible inside the function but can't be accessed outside it. If a closure were to be created inside this function, it would have access to these variables through its scope chain. Using this knowledge, you can create public methods that have access to private variables.

A *privileged method* is a public method that has access to private variables and/or private functions. There are two ways to create privileged methods on objects. The first is to do so inside a constructor, as in this example:

```
function MyObject() {

    //private variables and functions
    var privateVariable = 10;

    function privateFunction(){
        return false;
    }

    //privileged methods
    this.publicMethod = function (){
        privateVariable++;
        return privateFunction();
    };
}
```

This pattern defines all private variables and functions inside the constructor. Then privileged methods can be created to access those private members. This works because the privileged methods, when defined in the constructor, become closures with full access to all variables and functions defined inside the constructor's scope. In this example, the variable `privateVariable` and the function `privateFunction()` are accessed only by `publicMethod()`. Once an instance of `MyObject` is created, there is no way to access `privateVariable` and `privateFunction()` directly; you can do so only by way of `publicMethod()`.

You can define private and privileged members to hide data that should not be changed directly, as in this example:



Available for
download on
Wrox.com

```
function Person(name) {
    this.getName = function() {
        return name;
    };

    this.setName = function (value) {
        name = value;
    };
}

var person = new Person("Nicholas");
alert(person.getName()); // "Nicholas"
person.setName("Greg");
alert(person.getName()); // "Greg"
```

PrivilegedMethodExample01.htm

The constructor in this code defines two privileged methods: `getName()` and `setName()`. Each method is accessible outside the constructor and accesses the private `name` variable. Outside the `Person` constructor, there is no way to access `name`. Since both methods are defined inside the constructor, they are closures and have access to `name` through the scope chain. The private variable `name` is unique to each instance of `Person` since the methods are being re-created each time the constructor is called. One downside, however, is that you must use the constructor pattern to accomplish this result. As discussed in Chapter 6, the constructor pattern is flawed in that new methods are created for each instance. Using static private variables to achieve privileged methods avoids this problem.

Static Private Variables

Privileged methods can also be created by using a private scope to define the private variables or functions. The pattern is as follows:

```
(function(){

    //private variables and functions
    var privateVariable = 10;

    function privateFunction(){
        return false;
    }

    //constructor
    MyObject = function(){
    };

    //public and privileged methods
```

```

MyObject.prototype.publicMethod = function(){
    privateVariable++;
    return privateFunction();
};

})();

```

In this pattern, a private scope is created to enclose the constructor and its methods. The private variables and functions are defined first, followed by the constructor and the public methods. Public methods are defined on the prototype, as in the typical prototype pattern. Note that this pattern defines the constructor not by using a function declaration but instead by using a function expression. Function declarations always create local functions, which is undesirable in this case. For this same reason, the `var` keyword is not used with `MyObject`. Remember: initializing an undeclared variable always creates a global variable, so `MyObject` becomes global and available outside the private scope. Also keep in mind that assigning to an undeclared variable in strict mode causes an error.

The main difference between this pattern and the previous one is that private variables and functions are shared among instances. Since the privileged method is defined on the prototype, all instances use that same function. The privileged method, being a closure, always holds a reference to the containing scope. Consider the following:



Available for
download on
Wrox.com

```

(function(){

    var name = "";

    Person = function(value){
        name = value;
    };

    Person.prototype.getName = function(){
        return name;
    };

    Person.prototype.setName = function (value){
        name = value;
    };
})();

var person1 = new Person("Nicholas");
alert(person1.getName());    //"Nicholas"
person1.setName("Greg");
alert(person1.getName());    //"Greg"

var person2 = new Person("Michael");
alert(person1.getName());    //"Michael"
alert(person2.getName());    //"Michael"

```

[PrivilegedMethodExample02.htm](#)

The `Person` constructor in this example has access to the private variable `name`, as do the `getName()` and `setName()` methods. Using this pattern, the `name` variable becomes static and will be used among all instances. This means calling `setName()` on one instance affects all other instances. Calling `setName()` or creating a new `Person` instance sets the `name` variable to a new value. This causes all instances to return the same value.

Creating static private variables in this way allows for better code reuse through prototypes, although each instance doesn't have its own private variable. Ultimately, the decision to use instance or static private variables needs to be based on your individual requirements.



The farther up the scope chain a variable lookup is, the slower the lookup becomes because of the use of closures and private variables.

The Module Pattern

The previous patterns create private variables and privileged methods for custom types. The module pattern, as described by Douglas Crockford, does the same for singletons. Singletons are objects of which there will only ever be one instance. Traditionally, singletons are created in JavaScript using object literal notation, as shown in the following example:

```
var singleton = {
  name : value,
  method : function () {
    //method code here
  }
};
```

The module pattern augments the basic singleton to allow for private variables and privileged methods, taking the following format:

```
var singleton = function(){
  //private variables and functions
  var privateVariable = 10;

  function privateFunction(){
    return false;
  }

  //privileged/public methods and properties
  return {
    publicProperty: true,
    publicMethod : function(){}
  };
};
```

```

        privateVariable++;
        return privateFunction();
    }

};

}();

```

The module pattern uses an anonymous function that returns an object. Inside the anonymous function, the private variables and functions are defined first. After that, an object literal is returned as the function value. That object literal contains only properties and methods that should be public. Since the object is defined inside the anonymous function, all of the public methods have access to the private variables and functions. Essentially, the object literal defines the public interface for the singleton. This can be useful when the singleton requires some sort of initialization and access to private variables, as in this example:



Available for
download on
Wrox.com

```

var application = function(){

    //private variables and functions
    var components = new Array();

    //initialization
    components.push(new BaseComponent());

    //public interface
    return {
        getComponentCount : function(){
            return components.length;
        },
        registerComponent : function(component){
            if (typeof component == "object"){
                components.push(component);
            }
        }
    };
}();

```

[ModulePatternExample01.htm](#)

In web applications, it's quite common to have a singleton that manages application-level information. This simple example creates an application object that manages components. When the object is first created, the private components array is created and a new instance of BaseComponent is added to its list. (The code for BaseComponent is not important; it is used only to show initialization in the example.) The getComponentCount() and registerComponent() methods are privileged methods with access to the components array. The former simply returns the number of registered components, and the latter registers a new component.

The module pattern is useful for cases like this, when a single object must be created and initialized with some data and expose public methods that have access to private data. Every singleton created in this manner is an instance of Object, since ultimately an object literal

represents it. This is inconsequential, because singletons are typically accessed globally instead of passed as arguments into a function, which negates the need to use the `instanceof` operator to determine the object type.

The Module-Augmentation Pattern

Another take on the module pattern calls for the augmentation of the object before returning it. This pattern is useful when the singleton object needs to be an instance of a particular type but must be augmented with additional properties and/or methods. Consider the following example:

```
var singleton = function(){

    //private variables and functions
    var privateVariable = 10;

    function privateFunction(){
        return false;
    }

    //create object
    var object = new CustomType();

    //add privileged/public properties and methods
    object.publicProperty = true;

    object.publicMethod = function(){
        privateVariable++;
        return privateFunction();
    };

    //return the object
    return object;
}();
```

If the `application` object in the module pattern example had to be an instance of `BaseComponent`, the following code could be used:



Available for
download on
Wrox.com

```
var application = function(){

    //private variables and functions
    var components = new Array();

    //initialization
    components.push(new BaseComponent());

    //create a local copy of application
    var app = new BaseComponent();

    //public interface
    app.getComponentCount = function(){
        return components.length;
    };
}();
```

```

};

app.registerComponent = function(component){
    if (typeof component == "object"){
        components.push(component);
    }
};

//return it
return app;
}();

```

ModuleAugmentationPatternExample01.htm

In this rewritten version of the application singleton, the private variables are defined first, as in the previous example. The main difference is the creation of a variable named `app` that is a new instance of `BaseComponent`. This is the local version of what will become the `application` object. Public methods are then added onto the `app` object to access the private variables. The last step is to return the `app` object, which assigns it to `application`.

SUMMARY

Function expressions are useful tools in JavaScript programming. They allow truly dynamic programming where functions need not be named. These anonymous functions, also called lambda functions, are a powerful way to use JavaScript functions. The following is a summary of function expressions:

- Function expressions are different from function declarations. Function declarations require names, while function expressions do not. A function expression without a name is also called an anonymous function.
- With no definitive way to reference a function, recursive functions become more complicated.
- Recursive functions running in nonstrict mode may use `arguments.callee` to call themselves recursively instead of using the function name, which may change.

Closures are created when functions are defined inside other functions, allowing the closure access to all of the variables inside of the containing function, as follows:

- Behind the scenes, the closure's scope chain contains a variable object for itself, the containing function, and the global context.
- Typically a function's scope and all of its variables are destroyed when the function has finished executing.
- When a closure is returned from that function, its scope remains in memory until the closure no longer exists.

Using closures, it's possible to mimic block scoping in JavaScript, which doesn't exist natively, as follows:

- A function can be created and called immediately, executing the code within it but never leaving a reference to the function.
- This results in all of the variables inside the function being destroyed unless they are specifically set to a variable in the containing scope.

Closures can also be used to create private variables in objects, as follows:

- Even though JavaScript doesn't have a formal concept of private object properties, closures can be used to implement public methods that have access to variables defined within the containing scope.
- Public methods that have access to private variables are called privileged methods.
- Privileged methods can be implemented on custom types using the constructor or prototype patterns and on singletons by using the module or module-augmentation patterns.

Function expressions and closures are extremely powerful in JavaScript and can be used to accomplish many things. Keep in mind that closures maintain extra scopes in memory, so overusing them may result in increased memory consumption.

8

The Browser Object Model

WHAT'S IN THIS CHAPTER?

- Understanding the window object, the core of the BOM
- Controlling windows, frames, and pop-ups
- Page information from the location object
- Using the navigator object to learn about the browser

Though ECMAScript describes it as the core of JavaScript, the Browser Object Model (BOM) is really the core of using JavaScript on the Web. The BOM provides objects that expose browser functionality independent of any web page content. For years, a lack of any real specification made the BOM both interesting and problematic, because browser vendors were free to augment it as they saw fit. The commonalities between browsers became de facto standards that have survived browser development mostly for the purpose of interoperability. Part of the HTML5 specification now covers the major aspects of the BOM, as the W3C seeks to standardize one of the most fundamental parts of JavaScript in the browser.

THE WINDOW OBJECT

At the core of the BOM is the `window` object, which represents an instance of the browser. The `window` object serves a dual purpose in browsers, acting as the JavaScript interface to the browser window and the ECMAScript `Global` object. This means that every object, variable, and function defined in a web page uses `window` as its `Global` object and has access to methods like `parseInt()`.

The Global Scope

Since the `window` object doubles as the ECMAScript `Global` object, all variables and functions declared globally become properties of the `window` object. Consider this example:

```
var age = 29;
function sayAge(){
    alert(this.age);
}

alert(window.age);      //29
sayAge();              //29
window.sayAge();        //29
```

Here, a variable named `age` and a function named `sayAge()` are defined in the global scope, which automatically places them on the `window` object. Thus, the variable `age` is also accessible as `window.age`, and the function `sayAge()` is also accessible via `window.sayAge()`. Since `sayAge()` exists in the global scope, `this.age` maps to `window.age`, and the correct result is displayed.

Despite global variables becoming properties of the `window` object, there is a slight difference between defining a global variable and defining a property directly on `window`: global variables cannot be removed using the `delete` operator, while properties defined directly on `window` can. For example:



Available for
download on
Wrox.com

```
var age = 29;
window.color = "red";

//throws an error in IE < 9, returns false in all other browsers
delete window.age;

//throws an error in IE < 9, returns true in all other browsers
delete window.color;    //returns true

alert(window.age);      //29
alert(window.color);    //undefined
```

[DeleteOperatorExample01.htm](#)

Properties of `window` that were added via `var` statements have their `[[Configurable]]` attribute set to `false` and so may not be removed via the `delete` operator. Internet Explorer 8 and earlier enforced this by throwing an error when the `delete` operator is used on `window` properties regardless of how they were originally created. Internet Explorer 9 and later do not throw an error.

Another thing to keep in mind: attempting to access an undeclared variable throws an error, but it is possible to check for the existence of a potentially undeclared variable by looking on the `window` object. For example:

```
//this throws an error because oldValue is undeclared
var newValue = oldValue;

//this doesn't throw an error, because it's a property lookup
//newValue is set to undefined
var newValue = window.oldValue;
```

Keeping this in mind, there are many objects in JavaScript that are considered to be global, such as `location` and `navigator` (both discussed later in the chapter), but are actually properties of the `window` object.



Internet Explorer for Windows Mobile doesn't allow direct creation of new properties or methods on the window object via `window.property = value`. All variables and functions declared globally, however, will still become members of `window`.

Window Relationships and Frames

If a page contains frames, each frame has its own `window` object and is stored in the `frames` collection. Within the `frames` collection, the `window` objects are indexed both by number (starting at 0, going from left to right, and then row by row) and by the name of the frame. Each `window` object has a `name` property containing the name of the frame. Consider the following:



Available for download on
Wrox.com

```
<html>
  <head>
    <title>Frameset Example</title>
  </head>
  <frameset rows="160,*">
    <frame src="frame.htm" name="topFrame">
    <frameset cols="50%,50%">
      <frame src="anotherframe.htm" name="leftFrame">
      <frame src="yetanotherframe.htm" name="rightFrame">
    </frameset>
  </frameset>
</html>
```

[FramesetExample01.htm](#)

This code creates a frameset with one frame across the top and two frames underneath. Here, the top frame can be referenced by `window.frames[0]` or `window.frames["topFrame"]`; however, you would probably use the `top` object instead of `window` to refer to these frames (making it `top.frames[0]`, for instance).

The `top` object always points to the very top (outermost) frame, which is the browser window itself. This ensures that you are pointing to the correct frame from which to access the others. Any code written within a frame that references the `window` object is pointing to that frame's unique instance rather than the topmost one. Figure 8-1 indicates the various ways that the frames in the previous example may be accessed from code that exists in the topmost window.

**FIGURE 8-1**

Another window object is called `parent`. The `parent` object always points to the current frame's immediate parent frame. In some cases, `parent` may be equal to `top`, and when there are no frames, `parent` is equal to `top` (and both are equal to `window`). Consider the following example:



```

<html>
  <head>
    <title>Frameset Example</title>
  </head>
  <frameset rows="100,*">
    <frame src="frame.htm" name="topFrame">
    <frameset cols="50%,50%">
      <frame src="anotherframe.htm" name="leftFrame">
      <frame src="anotherframeset.htm" name="rightFrame">
    </frameset>
  </frameset>
</html>
  
```

[frameset1.htm](#)

This frameset has a frame that contains another frameset, the code for which is as follows:

```

<html>
  <head>
    <title>Frameset Example</title>
  </head>
  <frameset cols="50%,50%">
    <frame src="red.htm" name="redFrame">
  
```

```
<frame src="blue.htm" name="blueFrame">
</frameset>
</html>
```

[anotherframeset.htm](#)

When the first frameset is loaded into the browser, it loads another frameset into `rightFrame`. If code is written inside `redFrame` (or `blueFrame`), the parent object points to `rightFrame`. If, however, the code is written in `topFrame`, then `parent` points to `top` because its immediate parent is the outermost frame. Figure 8-2 shows the values of the various window objects when this example is loaded into a web browser.

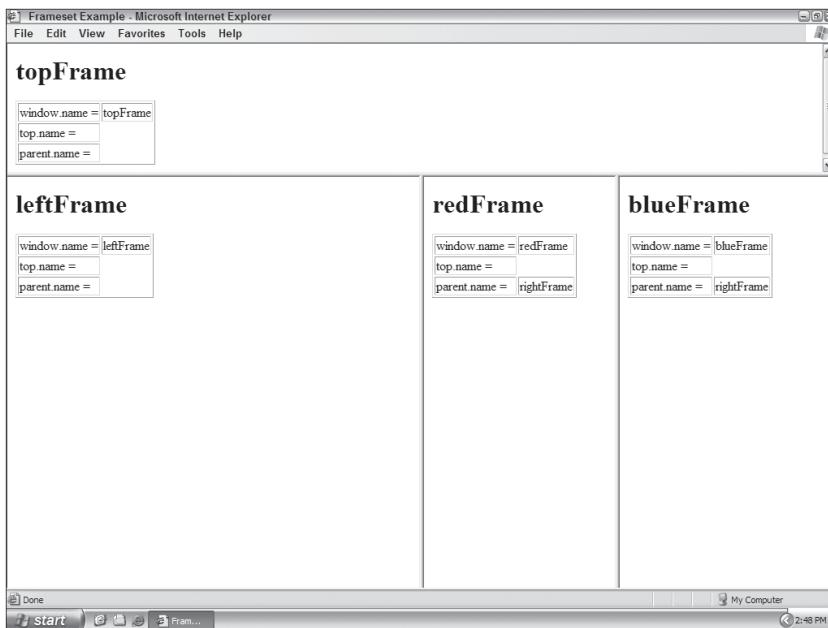


FIGURE 8-2

Note that the topmost window will never have a value set for `name` unless the window was opened using `window.open()`, as discussed later in this chapter.

There is one final `window` object, called `self`, which always points to `window`. The two can, in fact, be used interchangeably. Even though it has no separate value, `self` is included for consistency with the `top` and `parent` objects.

Each of these objects is actually a property of the `window` object, accessible via `window.parent`, `window.top`, and so on. This means it's possible to chain `window` objects together, such as `window.parent.parent.frames[0]`.



Whenever frames are used, multiple Global objects exist in the browser. Global variables defined in each frame are defined to be properties of that frame's window object. Since each window object contains the native type constructors, each frame has its own version of the constructors, which are not equal. For example, top.Object is not equal to top.frames[0].Object, which affects the use of instanceof when objects are passed across frames.

Window Position

The position of a window object may be determined and changed using various properties and methods. Internet Explorer, Safari, Opera, and Chrome all provide `screenLeft` and `screenTop` properties that indicate the window's location in relation to the left and top of the screen, respectively. Firefox provides this functionality through the `screenX` and `screenY` properties, which are also supported in Safari and Chrome. Opera supports `screenX` and `screenY`, but you should avoid using them in Opera, because they don't correspond to `screenLeft` and `screenTop`. The following code determines the left and top positions of the window across browsers:



Available for download on
Wrox.com

```
var leftPos = (typeof window.screenLeft == "number") ?
    window.screenLeft : window.screenX;
var topPos = (typeof window.screenTop == "number") ?
    window.screenTop : window.screenY;
```

WindowPositionExample01.htm

This example uses the ternary operator to determine if the `screenLeft` and `screenTop` properties exist. If they do (which is the case in Internet Explorer, Safari, Opera, and Chrome), they are used. If they don't exist (as in Firefox), `screenX` and `screenY` are used.

There are some quirks to using these values. In Internet Explorer, Opera, and Chrome, `screenLeft` and `screenTop` refer to the space from the left and top of the screen to the page view area represented by `window`. If the `window` object is the topmost object and the browser window is at the very top of the screen (with a y-coordinate of 0), the `screenTop` value will be the pixel height of the browser toolbars that appear above the page view area. Firefox and Safari treat these coordinates as being related to the entire browser window, so placing the window at y-coordinate 0 on the screen returns a top position of 0.

To further confuse things, Firefox, Safari, and Chrome always return the values of `top.screenX` and `top.screenY` for every frame on the page. Even if a page is offset by some margins, these same values are returned every time `screenX` and `screenY` are used in relation to a `window` object. Internet Explorer and Opera give accurate coordinates for the location of frames in relation to the screen edges.

The end result is that you cannot accurately determine the left and top coordinates of a browser window across all browsers. It is possible, however, to accurately move the window to a new position using the `moveTo()` and `moveBy()` methods. Each method accepts two arguments. `moveTo()` expects the x and y coordinates to move to, and `moveBy()` expects the number of pixels to move in each direction. Consider this example:

```
//move the window to the upper-left coordinate
window.moveTo(0,0);

//move the window down by 100 pixels
window.moveBy(0, 100);

//move the window to position (200, 300)
window.moveTo(200, 300);

//move the window left by 50 pixels
window.moveBy(-50, 0);
```

These methods may be disabled by the browser and are disabled by default for the main browser window in Internet Explorer 7+, Chrome, and Opera. None of these methods work for frames; they apply only to the topmost window object.

Window Size

Determining the size of a window cross-browser is not straightforward. Internet Explorer 9+, Firefox, Safari, Opera, and Chrome all provide four properties: innerWidth, innerHeight, outerWidth, and outerHeight. In Internet Explorer 9+, Safari, Firefox, and Chrome, outerWidth and outerHeight return the dimensions of the browser window itself (regardless of whether it's used on the topmost window object or on a frame). In Opera, these values are the size of the page viewport. The innerWidth and innerHeight properties indicate the size of the page viewport inside the browser window (minus borders and toolbars).

Internet Explorer 8 and earlier versions offer no way to get the current dimensions of the browser window; however, they do provide information about the viewable area of the page via the DOM.

The document.documentElement.clientWidth and document.documentElement.clientHeight properties provide the width and height of the page viewport in Internet Explorer, Firefox, Safari, Opera, and Chrome. In Internet Explorer 6, the browser must be in standards mode for these properties to be available; when in quirks mode, the information is available via document.body.clientWidth and document.body.clientHeight. When Chrome is in quirks mode, the values of clientWidth and clientHeight on document.documentElement and document.body both contain the viewport dimensions.

The end result is that there's no accurate way to determine the size of the browser window itself, but it is possible to get the dimensions of the page viewport, as shown in the following example:



Available for
download on
Wrox.com

```
var pageWidth = window.innerWidth,
    pageHeight = window.innerHeight;

if (typeof pageWidth != "number") {
    if (document.compatMode == "CSS1Compat") {
        pageWidth = document.documentElement.clientWidth;
        pageHeight = document.documentElement.clientHeight;
    } else {
        pageWidth = document.body.clientWidth;
        pageHeight = document.body.clientHeight;
    }
}
```

[WindowSizeExample01.htm](#)

In this code, `pageWidth` and `pageHeight` are assigned initial values of `window.innerWidth` and `window.innerHeight`, respectively. A check is then done to see if the value of `pageWidth` is a number; if not, then the code determines if the page is in standards mode by using `document.compatMode`. (This property is discussed fully in Chapter 11.) If it is, then `document.documentElement.clientWidth` and `document.documentElement.clientHeight` are used; otherwise, `document.body.clientWidth` and `document.body.clientHeight` are used.

For mobile devices, `window.innerWidth` and `window.innerHeight` are the dimensions of the visual viewport, which is the visible area of the page on the screen. Mobile Internet Explorer doesn't support these properties but provides the same information on `document.documentElement.clientWidth` and `document.documentElement.clientHeight`. These values change as you zoom in or out of a page.

In other mobile browsers, the measurements of `document.documentElement` provide measurements for the layout viewport, which are the actual dimensions of the rendered page (as opposed to the visual viewport, which is only a small portion of the entire page). Mobile Internet Explorer stores this information in `document.body.clientWidth` and `document.body.clientHeight`. These values do not change as you zoom in and out.

Because of these differences from desktop browsers, you may need to first determine if the user is on a mobile device before deciding which measurements to use and honor.



The topic of mobile viewports is a complex one with various exceptions and caveats. Peter-Paul Koch, a mobile development consultant, outlined all of his research at <http://quirksmode.org/mobile/viewports2.html>. This is recommended reading if you're developing for mobile devices.

The browser window can be resized using the `resizeTo()` and `resizeBy()` methods. Each method accepts two arguments: `resizeTo()` expects a new width and height, and `resizeBy()` expects the differences in each dimension. Here's an example:

```
//resize to 100 x 100
window.resizeTo(100, 100);

//resize to 200 x 150
window.resizeBy(100, 50);

//resize to 300 x 300
window.resizeTo(300, 300);
```

As with the window-movement methods, the resize methods may be disabled by the browser and are disabled by default in Internet Explorer 7+, Chrome, and Opera. Also like the movement methods, these methods apply only to the topmost window object.

Navigating and Opening Windows

The `window.open()` method can be used both to navigate to a particular URL and to open a new browser window. This method accepts four arguments: the URL to load, the window target, a string of features, and a Boolean value indicating that the new page should take the place of the currently loaded page in the browser history. Typically only the first three arguments are used; the last argument applies only when not opening a new window.

If the second argument passed to `window.open()` is the name of a window or frame that already exists, then the URL is loaded into the window or frame with that name. Here's an example:

```
//same as <a href="http://www.wrox.com" target="topFrame"></a>
window.open("http://www.wrox.com/", "topFrame");
```

This line of code acts as if the user clicked a link with the `href` attribute set to `"http://www.wrox.com"` and the `target` attribute set to `"topFrame"`. If there is a window or frame named `"topFrame"`, then the URL will be loaded there; otherwise, a new window is created and given the name `"topFrame"`. The second argument may also be any of the special window names: `_self`, `_parent`, `_top`, or `_blank`.

Popping Up Windows

When the second argument doesn't identify an existing window or frame, a new window or tab is created based on a string passed in as the third argument. If the third argument is missing, a new browser window (or tab, based on browser settings) is opened with all of the default browser window settings. (Toolbars, the location bar, and the status bar are all set based on the browser's default settings.) The third argument is ignored when not opening a new window.

The third argument is a comma-delimited string of settings indicating display information for the new window. The following table describes the various options.

SETTING	VALUE(S)	DESCRIPTION
<code>fullscreen</code>	<code>"yes"</code> or <code>"no"</code>	Indicates that the browser window should be maximized when created. Internet Explorer only.
<code>height</code>	Number	The initial height of the new window. This cannot be less than 100.
<code>left</code>	Number	The initial left coordinate of the new window. This cannot be a negative number.
<code>location</code>	<code>"yes"</code> or <code>"no"</code>	Indicates if the location bar should be displayed. The default varies based on the browser. When set to <code>"no"</code> , the location bar may be either hidden or disabled (browser-dependent).
<code>menubar</code>	<code>"yes"</code> or <code>"no"</code>	Indicates if the menu bar should be displayed. The default is <code>"no"</code> .

continues

(continued)

SETTING	VALUE(S)	DESCRIPTION
resizable	"yes" or "no"	Indicates if the new window can be resized by dragging its border. The default is "no".
scrollbars	"yes" or "no"	Indicates if the new window allows scrolling if the content cannot fit in the viewport. The default is "no".
status	"yes" or "no"	Indicates if the status bar should be displayed. The default varies based on the browser.
toolbar	"yes" or "no"	Indicates if the toolbar should be displayed. The default is "no".
top	Number	The initial top coordinate of the new window. This cannot be a negative number.
width	Number	The initial width of the new window. This cannot be less than 100.

Any or all of these settings may be specified as a comma-delimited set of name-value pairs. The name-value pairs are indicated by an equal sign. (No white space is allowed in the feature string.) Consider the following example:

```
window.open("http://www.wrox.com/", "wroxWindow",
            "height=400,width=400,top=10,left=10,resizable=yes");
```

This code opens a new resizable window that's 400 × 400 and positioned 10 pixels from the top and left of the screen.

The `window.open()` method returns a reference to the newly created window. This object is the same as any other `window` object except that you typically have more control over it. For instance, some browsers that don't allow you to resize or move the main browser window by default may allow you to resize or move windows that you've created using `window.open()`. This object can be used to manipulate the newly opened window in the same way as any other window, as shown in this example:

```
var wroxWin =window.open("http://www.wrox.com/", "wroxWindow",
                        "height=400,width=400,top=10,left=10,resizable=yes");

//resize it
wroxWin.resizeTo(500, 500);

//move it
wroxWin.moveTo(100, 100);
```

It's possible to close the newly opened window by calling the `close()` method as follows:

```
wroxWin.close();
```

This method works only for pop-up windows created by `window.open()`. It's not possible to close the main browser window without confirmation from the user. It is possible, however, for the pop-up window to close itself without user confirmation by calling `top.close()`. Once the window has been closed, the window reference still exists but cannot be used other than to check the `closed` property, as shown here:

```
wroxWin.close();
alert(wroxWin.closed); //true
```

The newly created `window` object has a reference back to the window that opened it via the `opener` property. This property is defined only on the topmost `window` object (`top`) of the pop-up window and is a pointer to the window or frame that called `window.open()`. For example:

```
var wroxWin =window.open("http://www.wrox.com/", "wroxWindow",
    "height=400,width=400,top=10,left=10,resizable=yes");

alert(wroxWin.opener == window); //true
```

Even though there is a pointer from the pop-up window back to the window that opened it, there is no reverse relationship. Windows do not keep track of the pop-ups that they spawn, so it's up to you to keep track if necessary.

Some browsers, such as Internet Explorer 8+ and Google Chrome, try to run each tab in the browser as a separate process. When one tab opens another, the `window` objects need to be able to communicate with one another, so the tabs cannot run in separate processes. Chrome allows you to indicate that the newly created tab should be run in a separate process by setting the `opener` property to `null`, as in the following example:

```
var wroxWin =window.open("http://www.wrox.com/", "wroxWindow",
    "height=400,width=400,top=10,left=10,resizable=yes");

wroxWin.opener = null;
```

Setting `opener` to `null` indicates to the browser that the newly created tab doesn't need to communicate with the tab that opened it, so it may be run in a separate process. Once this connection has been severed, there is no way to recover it.

Security Restrictions

Pop-up windows went through a period of overuse by advertisers online. Pop-ups were often disguised as system dialogs to get the user to click on an advertisement. Since these pop-up web pages were styled to look like system dialogs, it was unclear to the user whether the dialog was legitimate. To aid in this determination, browsers began putting limits on the configuration of pop-up windows.

Internet Explorer 6 on Windows XP Service Pack 2 implemented multiple security features on pop-up windows, including not allowing pop-up windows to be created or moved offscreen and ensuring that the status bar cannot be turned off. Beginning with Internet Explorer 7, the location bar cannot be turned off and pop-up windows can't be moved or resized by default. Firefox 1 turned off the ability to suppress the status bar, so all pop-up windows have to display the status bar regardless of the feature string passed into `window.open()`. Firefox 3 forces the location bar to always be

displayed on pop-up windows. Opera opens pop-up windows only within its main browser window but doesn't allow them to exist where they might be confused with system dialogs.

Additionally, browsers will allow the creation of pop-up windows only after a user action. A call to `window.open()` while a page is still being loaded, for instance, will not be executed and may cause an error to be displayed to the user. Pop-up windows may be opened based only on a click or a key press.

Chrome uses a different approach to handling pop-up windows that aren't initiated by the user. Instead of blocking them, the browser displays only the title bar of the pop-up window and places it in the lower-right corner of the browser window.



Internet Explorer lifts some restrictions on pop-up windows when displaying a web page stored on the computer's hard drive. The same code, when run on a server, will invoke the pop-up restrictions.

Pop-up Blockers

Most browsers have pop-up-blocking software built in, and for those that don't, utilities such as the Yahoo! Toolbar have built-in pop-up blockers. The result is that most unexpected pop-ups are blocked. When a pop-up is blocked, one of two things happens. If the browser's built-in pop-up blocker stopped the pop-up, then `window.open()` will most likely return `null`. In that case, you can tell if a pop-up was blocked by checking the return value, as shown in the following example:

```
var wroxWin = window.open("http://www.wrox.com", "_blank");
if (wroxWin == null){
    alert("The popup was blocked!");
}
```

When a browser add-on or other program blocks a pop-up, `window.open()` typically throws an error. So to accurately detect when a pop-up has been blocked, you must check the return value and wrap the call to `window.open()` in a `try-catch` block, as in this example:



```
var blocked = false;

try {
    var wroxWin = window.open("http://www.wrox.com", "_blank");
    if (wroxWin == null){
        blocked = true;
    }
} catch (ex){
    blocked = true;
}

if (blocked){
    alert("The popup was blocked!");
}
```

[PopupBlockerExample01.htm](#)

This code accurately detects if a pop-up blocker has blocked the call to `window.open()`, regardless of the method being used. Note that detecting if a pop-up was blocked does not stop the browser from displaying its own message about a pop-up being blocked.

Intervals and Timeouts

JavaScript execution in a browser is single-threaded, but does allow for the scheduling of code to run at specific points in time through the use of timeouts and intervals. Timeouts execute some code after a specified amount of time, whereas intervals execute code repeatedly, waiting a specific amount of time in between each execution.

You set a timeout using the `window.setTimeout()` method, which accepts two arguments: the code to execute and the number of time (in milliseconds) to wait before attempting to execute the code. The first argument can be either a string containing JavaScript code (as would be used with `eval()`) or a function. For example, both of the following display an alert after 1 second:



Available for
download on
Wrox.com

```
//avoid!
setTimeout("alert('Hello world!') ", 1000);

//preferred
setTimeout(function() {
    alert("Hello world!");
}, 1000);
```

[TimeoutExample01.htm](#)

Even though both of these statements work, it's considered poor practice to use a string as the first argument, because it brings with it performance penalties.

The second argument, the number of milliseconds to wait, is not necessarily when the specified code will execute. JavaScript is single-threaded and, as such, can execute only one piece of code at a time. To manage execution, there is a queue of JavaScript tasks to execute. The tasks are executed in the order in which they were added to the queue. The second argument of `setTimeout()` tells the JavaScript engine to add this task onto the queue after a set number of milliseconds. If the queue is empty, then that code is executed immediately; if the queue is not empty, the code must wait its turn.

When `setTimeout()` is called, it returns a numeric ID for the timeout. The timeout ID is a unique identifier for the scheduled code that can be used to cancel the timeout. To cancel a pending timeout, use the `clearTimeout()` method and pass in the timeout ID, as in the following example:

```
//set the timeout
var timeoutId = setTimeout(function() {
    alert("Hello world!");
}, 1000);

//nevermind - cancel it
clearTimeout(timeoutId);
```

[TimeoutExample02.htm](#)

As long as `clearTimeout()` is called before the specified amount of time has passed, a timeout can be canceled completely. Calling `clearTimeout()` after the code has been executed has no effect.



All code executed by a timeout runs in the global scope, so the value of this inside the function will always point to window when running in nonstrict mode and undefined when running in strict mode.

Intervals work in the same way as timeouts except that they execute the code repeatedly at specific time intervals until the interval is canceled or the page is unloaded. The `setInterval()` method lets you set up intervals, and it accepts the same arguments as `setTimeout()`: the code to execute (string or function) and the milliseconds to wait between executions. Here's an example:



```
//avoid!
setInterval("alert('Hello world!') ", 10000);

//preferred
setInterval(function() {
    alert("Hello world!");
}, 10000);
```

[IntervalExample01.htm](#)

The `setInterval()` method also returns an interval ID that can be used to cancel the interval at some point in the future. The `clearInterval()` method can be used with this ID to cancel all pending intervals. This ability is more important for intervals than timeouts since, if left unchecked, they continue to execute until the page is unloaded. Here is a common example of interval usage:

```
var num = 0;
var max = 10;
var intervalId = null;

function incrementNumber() {
    num++;

    //if the max has been reached, cancel all pending executions
    if (num == max) {
        clearInterval(intervalId);
        alert("Done");
    }
}

intervalId = setInterval(incrementNumber, 500);
```

[IntervalExample02.htm](#)

In this example, the variable num is incremented every half second until it finally reaches the maximum number, at which point the interval is canceled. This pattern can also be implemented using timeouts, as shown here:



Available for download on
Wrox.com

```
var num = 0;
var max = 10;

function incrementNumber() {
    num++;

    //if the max has not been reached, set another timeout
    if (num < max) {
        setTimeout(incrementNumber, 500);
    } else {
        alert("Done");
    }
}

setTimeout(incrementNumber, 500);
```

[TimeoutExample03.htm](#)

Note that when you're using timeouts, it is unnecessary to track the timeout ID, because the execution will stop on its own and continue only if another timeout is set. This pattern is considered a best practice for setting intervals without actually using intervals. True intervals are rarely used in production environments because the time between the end of one interval and the beginning of the next is not necessarily guaranteed, and some intervals may be skipped. Using timeouts, as in the preceding example, ensures that can't happen. Generally speaking, it's best to avoid intervals.

System Dialogs

The browser is capable of invoking system dialogs to display to the user through the `alert()`, `confirm()`, and `prompt()` methods. These dialogs are not related to the web page being displayed in the browser and do not contain HTML. Their appearance is determined by operating system and/or browser settings rather than CSS. Additionally, each of these dialogs is synchronous and modal, meaning code execution stops when a dialog is displayed, and resumes after it has been dismissed.

The `alert()` method has been used throughout this book. It simply accepts a string to display to the user. When `alert()` is called, a system message box displays the specified text to the user, followed by a single OK button. For example, `alert("Hello world!")` renders the dialog box shown in Figure 8-3 when used with Internet Explorer on Windows XP.

Alert dialogs are typically used when users must be made aware of something that they have no control over, such as an error. A user's only choice is to dismiss the dialog after reading the message.

The second type of dialog is invoked by calling `confirm()`. A confirm dialog looks similar to an alert dialog in that it displays a message to the user. The main difference between the two is the



FIGURE 8-3

presence of a Cancel button along with the OK button, which allows the user to indicate if a given action should be taken. For example, `confirm("Are you sure?")` displays the confirm dialog box shown in Figure 8-4.

To determine if the user clicked OK or Cancel, the `confirm()` method returns a Boolean value: `true` if OK was clicked, or `false` if Cancel was clicked or the dialog box was closed by clicking the X in the corner. Typical usage of a confirm dialog looks like this:

```
if (confirm("Are you sure?")) {
    alert("I'm so glad you're sure! ");
} else {
    alert("I'm sorry to hear you're not sure. ");
```

In this example, the confirm dialog is displayed to the user in the first line, which is a condition of the `if` statement. If the user clicks OK, an alert is displayed saying, “I'm so glad you're sure!” If, however, the Cancel button is clicked, an alert is displayed saying, “I'm sorry to hear you're not sure.” This type of pattern is often employed when the user tries to delete something, such as an e-mail message.

The final type of dialog is displayed by calling `prompt()`, which prompts the user for input. Along with OK and Cancel buttons, this dialog has a text box where the user may enter some data. The `prompt()` method accepts two arguments: the text to display to the user, and the default value for the text box (which can be an empty string). Calling `prompt("What's your name?", "Michael")` results in the dialog box shown in Figure 8-5.



FIGURE 8-4

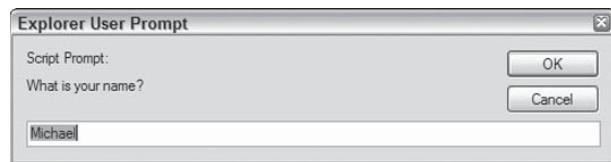


FIGURE 8-5

If the OK button is clicked, `prompt()` returns the value in the text box; if Cancel is clicked or the dialog is otherwise closed without clicking OK, the function returns `null`. Here's an example:

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

These system dialogs can be helpful for displaying information to the user and asking for confirmation of decisions. Since they require no HTML, CSS, or JavaScript to be loaded, they are fast and easy ways to enhance a web application.

Chrome and Opera introduced a special feature regarding these system dialogs. If the actively running script produces two or more system dialogs during its execution, each subsequent dialog after the first displays a check box that allows the user to disable any further dialogs until the page reloads (see Figure 8-6).

When the check box is checked and the dialog box is dismissed, all further system dialogs (alerts, confirms, and prompts) are blocked until the page is reloaded. Chrome gives the developer no indication as to whether the dialog was displayed. The dialog counter resets whenever the browser is idle, so if two separate user actions produce an alert, the check box will not be displayed in either; if a single user action produces two alerts in a row, the second will contain the check box. Since being first introduced, this feature has made it into Internet Explorer 9 and Firefox 4.

Two other types of dialogs can be displayed from JavaScript: `find` and `print`. Both of these dialogs are displayed asynchronously, returning control to the script immediately. The dialogs are the same as the ones the browser employs when the user selects either Find or Print from the browser's menu. These are displayed using the `find()` and `print()` methods on the `window` object as follows:

```
//display print dialog
window.print();

//display find dialog
window.find();
```

These two methods give no indication as to whether the user has done anything with the dialog, so it is difficult to make good use of them. Furthermore, since they are asynchronous, they don't contribute to Chrome's dialog counter and won't be affected by the user opting to disallow further dialogs.

THE LOCATION OBJECT

One of the most useful BOM objects is `location`, which provides information about the document that is currently loaded in the window, as well as general navigation functionality. The `location` object is unique in that it is a property of both `window` and `document`; both `window.location` and `document.location` point to the same object. Not only does `location` know about the currently loaded document, but it also parses the URL into discrete segments that can be accessed via a series of properties. These properties are enumerated in the following table (the `location` prefix is assumed).

PROPERTY NAME	EXAMPLE	DESCRIPTION
hash	"#contents"	The URL hash (the pound sign followed by zero or more characters), or an empty string if the URL doesn't have a hash.
host	"www.wrox.com:80"	The name of the server and port number if present.

continues



FIGURE 8-6

(continued)

PROPERTY NAME	EXAMPLE	DESCRIPTION
hostname	"www.wrox.com"	The name of the server without the port number.
href	"http://www.wrox.com"	The full URL of the currently loaded page. The <code>toString()</code> method of <code>location</code> returns this value.
pathname	"/WileyCDA/"	The directory and/or filename of the URL.
port	"8080"	The port of the request if specified in the URL. If a URL does not contain a port, then this property returns an empty string.
protocol	"http:"	The protocol used by the page. Typically "http:" or "https:".
search	"?q=javascript"	The query string of the URL. It returns a string beginning with a question mark.

Query String Arguments

Most of the information in `location` is easily accessible from these properties. The one part of the URL that isn't provided is an easy-to-use query string. Though `location.search` returns everything from the question mark until the end of the URL, there is no immediate access to query-string arguments on a one-by-one basis. The following function parses the query string and returns an object with entries for each argument:



```
function getQueryStringArgs() {
    //get query string without the initial ?
    var qs = (location.search.length > 0 ? location.search.substring(1) : ""),
        //object to hold data
        args = {},
        //get individual items
        items = qs.length ? qs.split("&") : [],
        item = null,
        name = null,
        value = null,
        //used in for loop
        i = 0,
        len = items.length;

    //assign each item onto the args object
    for (i=0; i < len; i++){
        item = items[i].split("=");
        name = item[0];
        value = item[1];
        if (args[name] === undefined) {
            args[name] = value;
        } else {
            args[name] += "," + value;
        }
    }
}
```

```

        name = decodeURIComponent(item[0]);
        value = decodeURIComponent(item[1]);

        if (name.length) {
            args[name] = value;
        }
    }

    return args;
}

```

LocationExample01.htm

The first step in this function is to strip off the question mark from the beginning of the query string. This happens only if `location.search` has one or more characters. The arguments will be stored on the `args` object, which is created using object-literal format. Next, the query string is split on the ampersand character, returning an array of strings in the format `name=value`. The `for` loop iterates over this array and then splits each item on the equal sign, returning an array where the first item is the name of the argument and the second item is the value. The `name` and `value` are each decoded using `decodeURIComponent()` (since query-string arguments are supposed to be encoded). Last, the `name` is assigned as a property on the `args` object and its value is set to `value`. This function is used as follows:

```

//assume query string of ?q=javascript&num=10

var args = getQueryStringArgs();

alert(args["q"]);      // "javascript"
alert(args["num"]);    // "10"

```

Each of the query-string arguments is now a property on the returned object, which provides fast access to each argument.

Manipulating the Location

The browser location can be changed in a number of ways using the `location` object. The first, and most common, way is to use the `assign()` method and pass in a URL, as in the following example:

```
location.assign("http://www.wrox.com");
```

This immediately starts the process of navigating to the new URL and makes an entry in the browser's history stack. If `location.href` or `window.location` is set to a URL, the `assign()` method is called with the value. For example, both of the following perform the same behavior as calling `assign()` explicitly:

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

Of these three approaches to changing the browser location, setting `location.href` is most often seen in code.

Changing various properties on the `location` object can also modify the currently loaded page. The `hash`, `search`, `hostname`, `pathname`, and `port` properties can be set with new values that alter the current URL, as in this example:

```
//assume starting at http://www.wrox.com/WileyCDA/
//changes URL to "http://www.wrox.com/WileyCDA/#section1"
location.hash = "#section1";

//changes URL to "http://www.wrox.com/WileyCDA/?q=javascript"
location.search = "?q=javascript";

//changes URL to "http://www.yahoo.com/WileyCDA/"
location.hostname = "www.yahoo.com";

//changes URL to "http://www.yahoo.com/mydir/"
location.pathname = "mydir";

//changes URL to "http://www.yahoo.com:8080/WileyCDA/"
```

Each time a property on `location` is changed, with the exception of `hash`, the page reloads with the new URL.



Changing the value of `hash` causes a new entry in the browser's history to be recorded as of Internet Explorer 8+, Firefox, Safari 2+, Opera 9+, and Chrome. In earlier Internet Explorer versions, the `hash` property was updated not when Back or Forward was clicked but only when a link containing a hashed URL was clicked.

When the URL is changed using one of the previously mentioned approaches, an entry is made in the browser's history stack so the user may click the Back button to navigate to the previous page. It is possible to disallow this behavior by using the `replace()` method. This method accepts a single argument, the URL to navigate to, but does not make an entry in the history stack. After calling `replace()`, the user cannot go back to the previous page. Consider this example:



```
<!DOCTYPE html>
<html>
<head>
<title>You won't be able to get back here</title>
</head>
<body>
<p>Enjoy this page for a second, because you won't be coming back here.</p>
<script type="text/javascript">
    setTimeout(function () {
        location.replace("http://www.wrox.com/");
    }, 1000);
</script>
</body>
</html>
```

LocationReplaceExample01.htm

If this page is loaded into a web browser, it will redirect to www.wrox.com after a second. At that point, the Back button will be disabled, and you won't be able to navigate back to this example page without typing in the complete URL again.

The last method of location is `reload()`, which reloads the currently displayed page. When `reload()` is called with no argument, the page is reloaded in the most efficient way possible, which is to say that the page may be reloaded from the browser cache if it hasn't changed since the last request. To force a reload from the server, pass in `true` as an argument like this:

```
location.reload();           //reload - possibly from cache
location.reload(true);      //reload - go back to the server
```

Any code located after a `reload()` call may or may not be executed, depending on factors such as network latency and system resources. For this reason, it is best to have `reload()` as the last line of code.

THE NAVIGATOR OBJECT

Originally introduced in Netscape Navigator 2, the `navigator` object is the standard for browser identification on the client. Though some browsers offer alternate ways to provide the same or similar information (for example, `window.clientInformation` in Internet Explorer and `window.opera` in Opera), the `navigator` object is common among all JavaScript-enabled web browsers. As with other BOM objects, each browser supports its own set of properties. The following table lists each available property and method, along with which browser versions support it.

PROPERTY/METHOD	DESCRIPTION	IE	FIREFOX	SAFARI/CHROME	OPERA
<code>appCodeName</code>	The name of the browser. Typically "Mozilla" even in non-Mozilla browsers.	3.0+	1.0+	1.0+	7.0+
<code>appMinorVersion</code>	Extra version information.	4.0+	—	—	9.5+
<code>appName</code>	Full browser name.	3.0+	1.0+	1.0+	7.0+
<code>appVersion</code>	Browser version. Typically does not correspond to the actual browser version.	3.0+	1.0+	1.0+	7.0+
<code>buildID</code>	Build number for the browser.	—	2.0+	—	—
<code>cookieEnabled</code>	Indicates if cookies are enabled.	4.0+	1.0+	1.0+	7.0+

continues

(continued)

PROPERTY/METHOD	DESCRIPTION	IE	FIREFOX	SAFARI/ CHROME	OPERA
cpuClass	The type of processor used on the client computer ("x86", "68K", "Alpha", "PPC", or "Other").	4.0+	—	—	—
javaEnabled()	Indicates if Java is enabled in the browser.	4.0+	1.0+	1.0+	7.0+
language	The browser's primary language.	—	1.0+	1.0+	7.0+
mimeTypes	Array of MIME types registered with the browser.	4.0+	1.0+	1.0+	7.0+
onLine	Indicates if the browser is connected to the Internet.	4.0+	1.0+	—	9.5+
opsProfile	Apparently unused. No documentation available.	4.0+	—	—	—
oscpu	The operating system and/or CPU on which the browser is running.	—	1.0+	—	—
platform	The system platform on which the browser is running.	4.0+	1.0+	1.0+	7.0+
plugins	Array of plug-ins installed on the browser. In Internet Explorer only, this is an array of all <embed> elements on the page.	4.0+	1.0+	1.0+	7.0+
preference()	Sets a user preference. Accessible only in privileged mode.	—	1.5+	—	—
product	The name of the product (typically "Gecko").	—	1.0+	1.0+	—

PROPERTY/METHOD	DESCRIPTION	IE	FIREFOX	SAFARI/ CHROME	OPERA
productSub	Extra information about the product (typically Gecko version information).	—	1.0+	1.0+	—
registerContentHandler()	Registers a web site as a handler for a specific MIME type.	—	2.0+	—	—
registerProtocolHandler()	Registers a web site as a handler for a particular protocol.	—	2.0+	—	—
securityPolicy	Deprecated. Name of the security policy. Retained for backwards compatibility with Netscape Navigator 4.	—	1.0+	—	—
systemLanguage	The language used by the operating system.	4.0+	—	—	—
taintEnabled()	Deprecated. Indicates if variable tainting is enabled. Retained for backwards compatibility with Netscape Navigator 3.	4.0+	1.0+	—	7.0+
userAgent	The user-agent string for the browser.	3.0+	1.0+	1.0+	7.0+
userLanguage	The default language for the operating system.	4.0+	—	—	7.0+
userProfile	Object for accessing user profile information.	4.0+	—	—	—
vendor	The brand name of the browser.	—	1.0+	1.0+	—
vendorSub	Extra information about the vendor.	—	1.0+	1.0+	—

The `navigator` object's properties are typically used to determine the type of browser that is running a web page (discussed fully in Chapter 9).

Detecting Plug-ins

One of the most common detection procedures is to determine whether the browser has a particular plug-in installed. For browsers other than Internet Explorer, this can be determined using the `plugins` array. Each item in the array contains the following properties:

- `name` — The name of the plug-in
- `description` — The description of the plug-in
- `filename` — The filename for the plug-in
- `length` — The number of MIME types handled by this plug-in

Typically, the name contains all of the information that's necessary to identify a plug-in, though this is not an exact science. Plug-in detection is done by looping over the available plug-ins and comparing a plug-in's name to a given name, as in this example:



```
//plugin detection - doesn't work in Internet Explorer
function hasPlugin(name){
    name = name.toLowerCase();
    for (var i=0; i < navigator.plugins.length; i++){
        if (navigator.plugins[i].name.toLowerCase().indexOf(name) > -1) {
            return true;
        }
    }
    return false;
}

//detect flash
alert(hasPlugin("Flash"));

//detect quicktime
alert(hasPlugin("QuickTime"));
```

PluginDetectionExample01.htm

The `hasPlugin()` example accepts a single argument: the name of a plug-in to detect. The first step is to convert that name to lowercase for easier comparison. Next, the `plugins` array is iterated over, and each `name` property is checked via `indexOf()` to see if the passed-in name appears somewhere in that string. This comparison is done in all lowercase to avoid casing errors. The argument should be as specific as possible to avoid confusion. Strings such as "Flash" and "QuickTime" are unique enough that there should be little confusion. This method works for detecting plug-ins in Firefox, Safari, Opera, and Chrome.



Each plugin object is also an array of `MimeType` objects that can be accessed using bracket notation. Each `MimeType` object has four properties: `description`, which is a description of the MIME type; `enabledPlugin`, which is a pointer back to the plugin object; `suffixes`, which is a comma-delimited string of file extensions for the MIME type; and `type`, which is the full MIME type string.

Detecting plug-ins in Internet Explorer is more problematic, because it doesn't support Netscape-style plug-ins. The only way to detect plug-ins in Internet Explorer is to use the proprietary `ActiveXObject` type and attempt to instantiate a particular plug-in. Plug-ins are implemented in Internet Explorer using COM objects, which are identified by unique strings. So to check for a particular plug-in, you must know its COM identifier. For instance, the identifier for Flash is "`ShockwaveFlash.ShockwaveFlash`". With this information, you can write a function to determine if the plug-in is installed in Internet Explorer as follows:



Available for download on
Wrox.com

```
//plugin detection for Internet Explorer
function hasIEPlugin(name){
    try {
        new ActiveXObject(name);
        return true;
    } catch (ex){
        return false;
    }
}

//detect flash
alert(hasIEPlugin("ShockwaveFlash.ShockwaveFlash"));

//detect quicktime
alert(hasIEPlugin("QuickTime.QuickTime"));
```

[PluginDetectionExample02.htm](#)

In this example, the `hasIEPlugin()` function accepts a COM identifier as its sole argument. In the function, an attempt is made to create a new `ActiveXObject` instance. This is encapsulated in a `try-catch` statement because an attempt to create an unknown COM object will throw an error. Therefore, if the attempt is successful, the function returns `true`. If there is an error, the `catch` block gets executed, which returns `false`. This code then checks to see if the Flash and QuickTime plug-ins are available in Internet Explorer.

Since these two plug-in-detection methods are so different, it's typical to create functions that test for specific plug-ins rather than use the generic methods described previously. Consider this example:

```
//detect flash for all browsers
function hasFlash(){
    var result = hasPlugin("Flash");
    if (!result){
        result = hasIEPlugin("ShockwaveFlash.ShockwaveFlash");
```

```

        }
        return result;
    }

//detect quicktime for all browsers
function hasQuickTime(){
    var result = hasPlugin("QuickTime");
    if (!result){
        result = hasIEPlugin("QuickTime.QuickTime");
    }
    return result;
}

//detect flash
alert(hasFlash());

//detect quicktime
alert(hasQuickTime());

```

PluginDetectionExample03.htm

This code defines two functions: `hasFlash()` and `hasQuickTime()`. Each function attempts to use the non–Internet Explorer plug-in–detection code first. If that method returns `false` (which it will for Internet Explorer), the Internet Explorer plug-in–detection method is called. If the Internet Explorer plug-in–detection method also returns `false`, then the result of the overall method is `false`. If either plug-in–detection function returns `true`, then the overall method returns `true`.



The plugins collection has a method called `refresh()`, which refreshes plugins to reflect any newly installed plug-ins. This method accepts a single argument: a Boolean value indicating if the page should be reloaded. When set to true, all pages containing plug-ins are reloaded; otherwise the plugins collection is updated, but the page is not reloaded.

Registering Handlers

Firefox 2 introduced the `registerContentHandler()` and `registerProtocolHandler()` methods to the `navigator` object. (These are now formally defined in HTML 5.) These methods allow a Website to indicate that it can handle specific types of information. With the rise of online RSS readers and online e-mail applications, this is a way for those applications to be used by default just as desktop applications are used.

The `registerContentHandler()` method accepts three arguments: the MIME type to handle, the URL of the page that can handle that MIME type, and the name of the application. For instance, to register a site as a handler of RSS feeds, you can use the following code:

```
navigator.registerContentHandler("application/rss+xml",
    "http://www.somereader.com?feed=%s", "Some Reader");
```

The first argument is the MIME type for RSS feeds. The second argument is the URL that should receive the RSS-feed URL. In this second argument, the %s represents the URL of the RSS feed, which the browser inserts automatically. The next time a request is made for an RSS feed, the browser will navigate to the URL specified and the web application can handle the request in the appropriate way.



Firefox through version 4 allows only three MIME types to be used in registerContentHandler(): "application/rss+xml", "application/atom+xml", and "application/vnd.mozilla.maybe.feed". All three do the same thing: register a handler for all RSS and Atom feeds.

A similar call can be made for protocols by using `registerProtocolHandler()`, which also accepts three arguments: the protocol to handle (i.e., "mailto" or "ftp"), the URL of the page that handles the protocol, and the name of the application. For example, to register a web application as the default mail client, you can use the following:

```
navigator.registerProtocolHandler("mailto",
    "http://www.somemailclient.com?cmd=%s", "Some Mail Client");
```

In this example, a handler is registered for the `mailto` protocol, which will now point to a web-based e-mail client. Once again, the second argument is the URL that should handle the request, and %s represents the original request.



In Firefox 2, `registerProtocolHandler()` was implemented but does not work. Firefox 3 fully implemented this method.

THE SCREEN OBJECT

The `screen` object (also a property of `window`) is one of the few JavaScript objects that have little to no programmatic use; it is used purely as an indication of client capabilities. This object provides information about the client's display outside the browser window, including information such as pixel width and height. Each browser provides different properties on the `screen` object. The following table indicates the properties and which browsers support them.

PROPERTY	DESCRIPTION	IE	FIREFOX	SAFARI/ CHROME	OPERA
availHeight	The pixel height of the screen minus system elements such as Windows (read only).	X	X	X	X
availLeft	The first pixel from the left that is not taken up by system elements (read only).		X	X	
availTop	The first pixel from the top that is not taken up by system elements (read only).		X	X	
availWidth	The pixel width of the screen minus system elements (read only).	X	X	X	X
bufferDepth	Reads or writes the number of bits used for offscreen bitmap rendering.	X			
colorDepth	The number of bits used to represent colors; for most systems, 32 (read only).	X	X	X	X
deviceXDPI	The actual horizontal DPI of the screen (read only).	X			
deviceYDPI	The actual vertical DPI of the screen (read only).	X			
fontSmoothingEnabled	Indicates if font smoothing is turned on (read only).	X			
height	The pixel height of the screen.	X	X	X	X
left	The pixel distance of the current screen's left side.		X		
logicalXDPI	The logical horizontal DPI of the screen (read only).	X			
logicalYDPI	The logical vertical DPI of the screen (read only).	X			
pixelDepth	The bit depth of the screen (read only).		X	X	X
top	The pixel distance of the current screen's top.		X		
updateInterval	Reads or writes the update interval for the screen in milliseconds.	X			
width	The pixel width of the screen.	X	X	X	X

This information is often aggregated by site-tracking tools that measure client capabilities, but typically it is not used to affect functionality. This information is sometimes used to resize the browser to take up the available space in the screen as follows:

```
window.resizeTo(screen.availWidth, screen.availHeight);
```

As noted previously, many browsers turn off the capability to resize the browser window, so this code may not work in all circumstances.

Mobile devices behave a little differently with respect to screen dimensions. A device running iOS will always return dimensions as if the device is being held in portrait mode (1024 × 768). Android devices, on the other hand, properly adjust the values of `screen.width` and `screen.height`.

THE HISTORY OBJECT

The `history` object represents the user's navigation history since the given window was first used. Because `history` is a property of `window`, each browser window, tab, and frame has its own `history` object relating specifically to that `window` object. For security reasons, it's not possible to determine the URLs that the user has visited. It is possible, however, to navigate backwards and forwards through the list of places the user has been without knowing the exact URL.

The `go()` method navigates through the user's history in either direction, backward or forward. This method accepts a single argument, which is an integer representing the number of pages to go backward or forward. A negative number moves backward in history (similar to clicking the browser's Back button), and a positive number moves forward (similar to clicking the browser's Forward button). Here's an example:

```
//go back one page  
history.go(-1);  
  
//go forward one page  
history.go(1);  
  
//go forward two pages  
history.go(2);
```

The `go()` method argument can also be a string, in which case the browser navigates to the first location in history that contains the given string. The closest location may be either backward or forward. If there's no entry in history matching the string, then the method does nothing, as in this example:

```
//go to nearest wrox.com page  
history.go("wrox.com");  
  
//go to nearest nczonline.net page  
history.go("nczonline.net");
```

Two shortcut methods, `back()` and `forward()`, may be used in place of `go()`. As you might expect, these mimic the browser Back and Forward buttons as follows:

```
//go back one page
history.back();

//go forward one page
history.forward();
```

The `history` object also has a property, `length`, which indicates how many items are in the history stack. This property reflects all items in the history stack, both those going backward and those going forward. For the first page loaded into a window, tab, or frame, `history.length` is equal to 0. By testing for this value as shown here, it's possible to determine if the user's start point was your page:

```
if (history.length == 0){
    //this is the first page in the user's window
}
```

Though not used very often, the `history` object typically is used to create custom Back and Forward buttons and to determine if the page is the first in the user's history. HTML5 further augments the `history` object. See Chapter 16 for more information.



Entries are made in the history stack whenever the page's URL changes. For Internet Explorer 8+, Opera, Firefox, Safari 3+, and Chrome, this includes changes to the URL hash (thus, setting location.hash causes a new entry to be inserted into the history stack for these browsers).

SUMMARY

The Browser Object Model (BOM) is based on the `window` object, which represents the browser window and the viewable page area. The `window` object doubles as the ECMAScript `Global` object, so all global variables and functions become properties on it, and all native constructors and functions exist on it initially. This chapter discussed the following elements of the BOM:

- When frames are used, each frame has its own `window` object and its own copies of all native constructors and functions. Each frame is stored in the `frames` collection, indexed both by position and by name.
- To reference other frames, including parent frames, there are several `window` pointers.
- The `top` object always points to the outermost frame, which represents the entire browser window.
- The `parent` object represents the containing frame, and `self` points back to `window`.

-
- The `location` object allows programmatic access to the browser's navigation system. By setting properties, it's possible to change the browser's URL piece by piece or altogether.
 - The `replace()` method allows for navigating to a new URL and replacing the currently displayed page in the browser's history.
 - The `navigator` object provides information about the browser. The type of information provided depends largely on the browser being used, though some common properties, such as `userAgent`, are available in all browsers.

Two other objects available in the BOM perform very limited functions. The `screen` object provides information about the client display. This information is typically used in metrics gathering for web sites. The `history` object offers a limited peek into the browser's history stack, allowing developers to determine how many sites are in the history stack and giving them the ability to go back or forward to any page in the history.

9

Client Detection

WHAT'S IN THIS CHAPTER?

- Using capability detection
- The history of user-agent detection
- When to use each type of detection

Although browser vendors have made a concerted effort to implement common interfaces, the fact remains that each browser presents its own capabilities and flaws. Browsers that are available cross-platform often have different issues, even though they are technically the same version. These differences force web developers to either design for the lowest common denominator or, more commonly, use various methods of client detection to work with or around limitations.

Client detection remains one of the most controversial topics in web development. The idea that browsers should support a common set of functionality pervades most conversations on the topic. In an ideal world, this would be the case. In reality, however, there are enough browser differences and quirks that client detection becomes not just an afterthought but also a vital part of the development strategy.

There are several approaches to determine the web client being used, and each has advantages and disadvantages. It's important to understand that client detection should be the very last step in solving a problem; whenever a more common solution is available, that solution should be used. Design for the most common solution first and then augment it with browser-specific solutions later.

CAPABILITY DETECTION

The most commonly used and widely accepted form of client detection is called *capability detection*. Capability detection (also called feature detection) aims not to identify a specific browser being used but rather to identify the browser's capabilities. This approach presumes

that specific browser knowledge is unnecessary and that the solution may be found by determining if the capability in question actually exists. The basic pattern for capability detection is as follows:

```
if (object.propertyInQuestion){  
    //use object.propertyInQuestion  
}
```

For example, the DOM method `document.getElementById()` didn't exist in Internet Explorer prior to version 5. This method simply didn't exist in earlier versions, although the same functionality could be achieved using the nonstandard `document.all` property. This led to a capability detection fork such as the following:

```
function getElement(id){  
    if (document.getElementById){  
        return document.getElementById(id);  
    } else if (document.all){  
        return document.all[id];  
    } else {  
        throw new Error("No way to retrieve element!");  
    }  
}
```

The purpose of the `getElement()` function is to return an element with the given ID. Since `document.getElementById()` is the standard way of achieving this, it is tested for first. If the function exists (it isn't undefined), then it is used. Otherwise, a check is done to determine if `document.all` is available, and if so, that is used. If neither method is available (which is highly unlikely), an error is thrown to indicate that the function won't work.

There are two important concepts to understand in capability detection. As just mentioned, the most common way to achieve the result should be tested for first. In the previous example, this meant testing for `document.getElementById()` before `document.all`. Testing for the most common solution ensures optimal code execution by avoiding multiple-condition testing in the common case.

The second important concept is that you must test for exactly what you want to use. Just because one capability exists doesn't necessarily mean another exists. Consider the following example:

```
function getWindowWidth(){  
    if (document.all){ //assumes IE  
        return document.documentElement.clientWidth; //INCORRECT USAGE!!!  
    } else {  
        return window.innerWidth;  
    }  
}
```

This example shows an incorrect usage of capability detection. The `getWindowWidth()` function first checks to see if `document.all` exists. It does, so the function then returns `document.documentElement.clientWidth`. As discussed in Chapter 8, Internet Explorer 8 and earlier versions do not support the `window.innerWidth` property. The problem in this code is that a test for

`document.all` does not necessarily indicate that the browser is Internet Explorer. It could, in fact, be an early version of Opera, which supported `document.all` and `window.innerWidth`.

Safer Capability Detection

Capability detection is most effective when you verify not just that the feature is present but also that the feature is likely to behave in an appropriate manner. The examples in the previous section rely on type coercion of the tested object member to make a determination as to its presence.

While this tells you about the presence of the object member, there is no indication if the member is the one you're expecting. Consider the following function that tries to determine if an object is sortable:

```
//AVOID! Incorrect capability detection - only checks for existence
function isSortable(object){
    return !object.sort;
}
```

This function attempts to determine that an object can be sorted by checking for the presence of the `sort()` method. The problem is that any object with a `sort` property will also return `true`:

```
var result = isSortable({ sort: true });
```

Simply testing for the existence of a property doesn't definitively indicate that the object in question is sortable. The better approach is to check that `sort` is actually a function:

```
//Better - checks if sort is a function
function isSortable(object){
    return typeof object.sort == "function";
}
```

The `typeof` operator is used in this code to determine that `sort` is actually a function and therefore can be called to sort the data contained within.

Capability detection using `typeof` is preferred whenever possible, but it is not infallible. In particular, host objects are under no obligation to return rational values for `typeof`. The most egregious example of this occurs with Internet Explorer. In most browsers, the following code returns `true` if `document.createElement()` is present:

```
//doesn't work properly in Internet Explorer <= 8
function hascreateElement(){
    return typeof document.createElement == "function";
}
```

In Internet Explorer 8 and earlier, the function returns `false` because `typeof document.createElement` returns `"object"` instead of `"function"`. As mentioned previously, DOM objects are host objects, and host objects are implemented via COM instead of JScript in Internet Explorer 8 and earlier. As such, the actual function `document.createElement()` is implemented as a COM object and `typeof` then returns `"object"`. Internet Explorer 9 correctly returns `"function"` for DOM methods.

Internet Explorer has further examples where using `typeof` doesn't behave as expected. ActiveX objects (supported only in Internet Explorer) act very differently than other objects. For instance, testing for a property without using `typeof` may cause an error, as in this code:

```
//causes an error in Internet Explorer
var xhr = new ActiveXObject("Microsoft.XMLHttp");
if (xhr.open){      //error occurs here
    //do something
}
```

Simply accessing a function as a property, which this example does, causes a JavaScript error. It is safer to use `typeof`; however, Internet Explorer returns "unknown" for `typeof xhr.open`. That means the most complete way to test for the existence of a function on any object in a browser environment is along the lines of this function:

```
//credit: Peter Michaux
function isHostMethod(object, property) {
    var t = typeof object[property];
    return t=='function' ||
        (!(t=='object' && object[property])) ||
        t=='unknown';
}
```

You can then use this function as follows:

```
result = isHostMethod(xhr, "open"); //true
result = isHostMethod(xhr, "foo"); //false
```

The `isHostMethod()` function is the safest to use today, understanding the quirks of browsers. Note that host objects are under no obligation to maintain their current implementation details or to mimic already-existing host object behavior. Because of this, there is no guarantee that this function, or any other, will continue to be accurate if implementations change. As the developer, you must assess your risk tolerance based on the functionality you're trying to implement.



For an exhaustive discussion of the ins and outs of capability detection in JavaScript, please see Peter Michaux's article "Feature Detection: State of the Art Browser Scripting" at <http://peter.michaux.ca/articles/feature-detection-state-of-the-art-browser-scripting>.

Capability Detection Is Not Browser Detection

Detecting a particular capability or set of capabilities does not necessarily indicate the browser in use. The following "browser detection" code, or something similar, can be found on numerous web sites and is an example of improper capability detection:

```
//AVOID! Not specific enough
var isFirefox = !(navigator.vendor && navigator.vendorSub);

//AVOID! Makes too many assumptions
var isIE = !(document.all && document.uniqueID);
```

This code represents a classic misuse of capability detection. In the past, Firefox could be determined by checking for `navigator.vendor` and `navigator.vendorSub`, but then Safari came along and implemented the same properties, meaning this code would give a false positive. To detect Internet Explorer, the code checks for the presence of `document.all` and `document.uniqueID`. This assumes that both of these properties will continue to exist in future versions of IE and won't ever be implemented by any other browser. Both checks use a double NOT operator to produce a Boolean result (which is more optimal to store and access).

It is appropriate, however, to group capabilities together into classes of browsers. If you know that your application needs to use specific browser functionality, it may be useful to do detection for all of the capabilities once rather than doing it repeatedly. Consider this example:



Available for
download on
Wrox.com

```
//determine if the browser has Netscape-style plugins
var hasNSPlugins = !(navigator.plugins && navigator.plugins.length);

//determine if the browser has basic DOM Level 1 capabilities
var hasDOM1 = !(document.getElementById && document.createElement &&
document.getElementsByTagName);
```

CapabilitiesDetectionExample01.htm

In this example, two detections are done: one to see if the browser supports Netscape-style plug-ins, and one to determine if the browser supports basic DOM Level 1 capabilities. These Boolean values can later be queried, and it will take less time than it would to retest the capabilities.



Capability detection should be used only to determine the next step in a solution, not as a flag indicating a particular browser is being used.

QUIRKS DETECTION

Similar to capability detection, *quirks detection* aims to identify a particular behavior of the browser. Instead of looking for something that's supported, however, quirks detection attempts to figure out what isn't working correctly ("quirk" really means "bug"). This often involves running a short amount of code to determine that a feature isn't working correctly. For example, a bug in Internet Explorer 8 and earlier causes instance properties with the same name as prototype



Available for
download on
Wrox.com

```
var hasDontEnumQuirk = function(){
    var o = { toString : function(){}} ;
    for (var prop in o){
        if (prop == "toString"){
            return false;
        }
    }
    return true;
}();
```

[QuirksDetectionExample01.htm](#)

This code uses an anonymous function to test for the quirk. An object is created with the `toString()` method defined. In proper ECMAScript implementations, `toString` should be returned as a property in the `for-in` loop.

Another quirk commonly tested for is Safari versions prior to 3 enumerating over shadowed properties. This can be tested for as follows:

```
var hasEnumShadowsQuirk = function(){
    var o = { toString : function(){}} ;
    var count = 0;
    for (var prop in o){
        if (prop == "toString"){
            count++;
        }
    }
    return (count > 1);
}();
```

[QuirksDetectionExample01.htm](#)

If the browser has this quirk, an object with a custom `toString()` method will cause two instances of `toString` to appear in the `for-in` loop.

Quirks are frequently browser-specific and often are recognized as bugs that may or may not be fixed in later versions. Since quirks detection requires code to be run, it's advisable to test for only the quirks that will affect you directly and to do so at the beginning of the script to get it out of the way.

USER-AGENT DETECTION

The third, and most controversial, client-detection method is called *user-agent detection*. User-agent detection uses the browser's user-agent string to determine the exact browser being used. The user-agent string is sent as a response header for every HTTP request and is made accessible in

JavaScript through `navigator.userAgent`. On the server side, it is a common and accepted practice to look at the user-agent string to determine the browser being used and to act accordingly. On the client side, however, user-agent detection is generally considered a last-ditch approach for when capability detection and/or quirks detection cannot be used.

Among the controversial aspects of the user-agent string is its long history of *spoofing*, when browsers try to fool servers by including erroneous or misleading information in their user-agent string. To understand this problem, it's necessary to take a look back at how the user-agent string has evolved since the Web first appeared.

History

The HTTP specification, both versions 1.0 and 1.1, indicates that browsers should send short user-agent strings specifying the browser name and version. RFC 2616 (the HTTP 1.1 protocol specification) describes the user-agent string in this way:

Product tokens are used to allow communicating applications to identify themselves by software name and version. Most fields using product tokens also allow sub-products which form a significant part of the application to be listed, separated by white space. By convention, the products are listed in order of their significance for identifying the application.

The specification further stipulates that the user-agent string should be specified as a list of products in the form token/product version. In reality, however, user-agent strings have never been that simple.

Early Browsers

The first web browser, Mosaic, was released in 1993 by the National Center for Supercomputing Applications (NCSA). Its user-agent string was fairly simple, taking a form similar to this:

Mosaic/0.9

Though this would vary depending on the operating system and platform, the basic format was simple and straightforward. The text before the forward slash indicated the product name (sometimes appearing as NCSA Mosaic or other derivatives), and the text after the slash is the product version.

When Netscape Communications began developing its web browser, its code name was Mozilla (short for “Mosaic Killer”). Netscape Navigator 2, the first publicly available version, had a user-agent string with the following format:

Mozilla/Version [Language] (Platform; Encryption)

Netscape kept the tradition of using the product name and version as the first part of the user-agent string but added the following information afterward:

- **Language** — The language code indicating where the application was intended to be used.
- **Platform** — The operating system and/or platform on which the application is running.

- **Encryption** — The type of security encryption included. Possible values are U (128-bit encryption), I (40-bit encryption), and N (no encryption).

A typical user-agent string from Netscape Navigator 2 looked like this:

```
Mozilla/2.02 [fr] (WinNT; I)
```

This string indicates Netscape Navigator 2.02 is being used, is compiled for use in French-speaking countries, and is being run on Windows NT with 40-bit encryption. At this point in time, it was fairly easy to determine what browser was being used just by looking at the product name in the user-agent string.

Netscape Navigator 3 and Internet Explorer 3

In 1996, Netscape Navigator 3 was released and became the most popular web browser, surpassing Mosaic. The user-agent string went through only a small change, removing the language token and allowing optional information about the operating system or CPU used on the system. The format became the following:

```
Mozilla/Version (Platform; Encryption [, OS-or-CPU description])
```

A typical user-agent string for Netscape Navigator 3 running on a Windows system looked like this:

```
Mozilla/3.0 (Win95; U)
```

This string indicates Netscape Navigator 3 running on Windows 95 with 128-bit encryption. Note that the OS or CPU description was left off when the browser ran on Windows systems.

Shortly after the release of Netscape Navigator 3, Microsoft released its first publicly available web browser, Internet Explorer 3. Since Netscape was the dominant browser at the time, many servers specifically checked for it before serving up pages. The inability to access pages in Internet Explorer would have crippled adoption of the fledgling browser, so the decision was made to create a user-agent string that would be compatible with the Netscape user-agent string. The result was the following format:

```
Mozilla/2.0 (compatible; MSIE Version; Operating System)
```

For example, Internet Explorer 3.02 running on Windows 95 had this user-agent string:

```
Mozilla/2.0 (compatible; MSIE 3.02; Windows 95)
```

Since most browser sniffers at the time looked only at the product-name part of the user-agent string, Internet Explorer successfully identified itself as Mozilla, the same as Netscape Navigator. This move caused some controversy since it broke the convention of browser identification. Furthermore, the true browser version is buried in the middle of the string.

Another interesting part of this string is the identification of Mozilla 2.0 instead of 3.0. Since 3.0 was the dominant browser at the time, it would have made more sense to use that. The actual reason remains a mystery — it was more likely an oversight than anything else.

Netscape Communicator 4 and Internet Explorer 4–8

In August 1997, Netscape Communicator 4 was released (the name was changed from *Navigator* to *Communicator* for this release). Netscape opted to keep the following user-agent string format from version 3:

```
Mozilla/Version (Platform; Encryption [; OS-or-CPU description])
```

With version 4 on a Windows 98 machine, the user-agent string looked like this:

```
Mozilla/4.0 (Win98; I)
```

As Netscape released patches and fixes for its browser, the version was incremented accordingly, as the following user-agent string from version 4.79 indicates:

```
Mozilla/4.79 (Win98; I)
```

When Microsoft released Internet Explorer 4, the user-agent string featured an updated version, taking the following format:

```
Mozilla/4.0 (compatible; MSIE Version; Operating System)
```

For example, Internet Explorer 4 running on Windows 98 returned the following user-agent string:

```
Mozilla/4.0 (compatible; MSIE 4.0; Windows 98)
```

With this change, the reported Mozilla version and the actual version of Internet Explorer were synchronized, allowing for easy identification of these fourth-generation browsers. Unfortunately, the synchronization ended there. When Internet Explorer 4.5 (released only for Macs) debuted, the Mozilla version remained 4 while the Internet Explorer version changed as follows:

```
Mozilla/4.0 (compatible; MSIE 4.5; Mac_PowerPC)
```

In Internet Explorer versions through version 7, the following pattern has remained:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
```

Internet Explorer 8 introduced an additional token called Trident, which is the name of the rendering engine. The format became:

```
Mozilla/4.0 (compatible; MSIE Version; Operating System; Trident/TridentVersion)
```

For example:

```
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)
```

The extra Trident token is designed to help developers determine when Internet Explorer 8 is running in compatibility mode. In that case the MSIE version becomes 7, but the Trident version remains in the user-agent string:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0)
```

Adding this extra token makes it possible to determine if a browser is Internet Explorer 7 (in which case there is no Trident token) or Internet Explorer 8 running in compatibility mode.

Internet Explorer 9 slightly changed this format. The Mozilla version was incremented to 5.0, and the Trident version was also incremented to 5.0. The default user-agent string for Internet Explorer 9 looks like this:

```
Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)
```

When Internet Explorer 9 runs in compatibility mode, the old Mozilla version and MSIE version are restored while the Trident version remains at 5.0. For example, the following user-agent string is Internet Explorer 9 running in Internet Explorer 7 compatibility mode:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; Trident/5.0)
```

All of these changes were made to ensure that past user-agent detection scripts continue to work correctly while enabling new scripts to have additional information.

Gecko

The Gecko rendering engine is at the heart of Firefox. When Gecko was first developed, it was as part of the generic Mozilla browser that was to become Netscape 6. A specification was written for Netscape 6, indicating how the user-agent string should be constructed in all future versions. The new format represented a fairly drastic departure from its relatively simple user-agent string used through version 4.x. The format is as follows:

```
Mozilla/MozillaVersion (Platform; Encryption; OS-or-CPU; Language;  
PrereleaseVersion)Gecko/GeckoVersion  
ApplicationProduct/ApplicationProductVersion
```

A lot of thought went into this remarkably complex user-agent string. The following table lists the meaning of each section.

STRING	REQUIRED?	DESCRIPTION
<i>MozillaVersion</i>	Yes	The version of Mozilla.
<i>Platform</i>	Yes	The platform on which the browser is running. Possible values include Windows, Mac, and X11 (for Unix X-windows systems).
<i>Encryption</i>	Yes	Encryption capabilities: U for 128-bit, I for 40-bit, or N for no encryption.

STRING	REQUIRED?	DESCRIPTION
OS-or-CPU	Yes	The operating system the browser is being run on or the processor type of the computer running the browser. If the platform is Windows, this is the version of Windows (such as WinNT, Win95, and so on). If the platform is Macintosh, then this is the CPU (either 68k, PPC for PowerPC, or MacIntel). If the Platform is X11, this is the Unix operating-system name as obtained by the Unix command <code>uname -sm</code> .
<i>Language</i>	Yes	The language that the browser was created for use in.
<i>Prerelease Version</i>	No	Originally intended as the prerelease version number for Mozilla, it now indicates the version number of the Gecko rendering engine.
<i>GeckoVersion</i>	Yes	The version of the Gecko rendering engine represented by a date in the format <code>yyyymmdd</code> .
<i>ApplicationProduct</i>	No	The name of the product using Gecko. This may be Netscape, Firefox, and so on.
<i>ApplicationProductVersion</i>	No	The version of the <i>ApplicationProduct</i> ; this is separate from the <i>MozillaVersion</i> and the <i>GeckoVersion</i> .

To better understand the Gecko user-agent string format, consider the following user-agent strings taken from various Gecko-based browsers.

Netscape 6.21 on Windows XP:

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:0.9.4) Gecko/20011128
Netscape6/6.2.1
```

SeaMonkey 1.1a on Linux:

```
Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1b2) Gecko/20060823 SeaMonkey/1.1a
```

Firefox 2.0.0.11 on Windows XP:

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.11) Gecko/20071127
Firefox/2.0.0.11
```

Camino 1.5.1 on Mac OS X:

```
Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en; rv:1.8.1.6) Gecko/20070809
Camino/1.5.1
```

All of these user-agent strings indicate Gecko-based browsers (albeit using different versions). Oftentimes, looking for a particular browser is not as important as understanding whether it's

Gecko-based. The Mozilla version hasn't changed from 5.0 since the first Gecko-based browser was released, and it likely won't change again.

With the release of Firefox 4, Mozilla simplified the user-agent string. The major changes include:

- Removal of the Language token (i.e., "en-US" in the previous examples).
- The Encryption token is not present when the browser uses strong encryption (which is the default). That means there will no longer be a "U" in Mozilla user-agent strings, but "I" and "N" might still be present.
- The Platform token has been removed for Windows user-agent strings as "Windows" was redundant with the OS-or-CPU token, which always contained the string "Windows".
- The GeckoVersion token is now frozen to "Gecko/20100101".

An example of the final Firefox 4 user-agent string is:

```
Mozilla/5.0 (Windows NT 6.1; rv:2.0.1) Gecko/20100101 Firefox 4.0.1
```

WebKit

In 2003, Apple announced that it would release its own web browser, called Safari. The Safari rendering engine, called WebKit, began as a fork of the KHTML rendering engine used in the Linux-based Konqueror web browser. A couple of years later, WebKit was split off into its own open-source project, focusing on development of the rendering engine.

Developers of this new browser and rendering engine faced a problem similar to that faced by Internet Explorer 3: how do you ensure that the browser isn't locked out of popular sites? The answer is, put enough information into the user-agent string to convince web sites that the browser is compatible with another popular browser. This led to a user-agent string with the following format:

```
Mozilla/5.0 (Platform; Encryption; OS-or-CPU; Language)  
AppleWebKit/AppleWebKitVersion (KHTML, like Gecko) Safari/SafariVersion
```

Here's an example:

```
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/124 (KHTML, like Gecko)  
Safari/125.1
```

As you can see, this is another long user-agent string. It takes into account not only the version of the Apple WebKit but also the Safari version. A point of contention over whether to identify the browser as Mozilla was resolved rather quickly for compatibility reasons. Now, all WebKit-based browsers identify themselves as Mozilla 5.0, the same as all Gecko-based browsers. The Safari version has typically been the build number of the browser, not necessarily a representation of the release version number. So although Safari 1.25 has the number 125.1 in the user-agent string, there may not always be a one-to-one match.

The most interesting and controversial part of this user-agent string is the addition of the string "(KHTML, like Gecko)" in a pre-1.0 version of Safari. Apple got a lot of pushback from developers who saw this as a blatant attempt to trick clients and servers into thinking Safari was actually

Gecko (as if adding Mozilla/5.0 wasn't enough). Apple's response was similar to Microsoft's when the Internet Explorer user-agent string came under fire: Safari is compatible with Mozilla, and web sites shouldn't block out Safari users because they appear to be using an unsupported browser.

Safari's user-agent string was augmented slightly when version 3 was released. The following version token is now used to identify the actual version of Safari being used:

```
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/522.15.5
(KHTML, like Gecko) Version/3.0.3 Safari/522.15.5
```

Note that this change was made only to Safari, not to WebKit, so other WebKit-based browsers may not have this change. Generally speaking, as with Gecko, it's typical to determine that a browser is WebKit-based rather than trying to identify Safari specifically.

Konqueror

Konqueror, the browser bundled with the KDE Linux desktop environment, is based on the KHTML open-source rendering engine. Though available only on Linux, Konqueror has an active user base. For optimal compatibility, Konqueror opted to format its user-agent string after Internet Explorer as follows:

```
Mozilla/5.0 (compatible; Konqueror/Version; OS-or-CPU)
```

However, Konqueror 3.2 introduced a change to coincide with changes to the WebKit user-agent string, identifying itself as KHTML as follows:

```
Mozilla/5.0 (compatible; Konqueror/Version; OS-or-CPU) KHTML/KHTMLVersion
(like Gecko)
```

Here's an example:

```
Mozilla/5.0 (compatible; Konqueror/3.5; SunOS) KHTML/3.5.0 (like Gecko)
```

The version numbers for Konqueror and KHTML tend to coincide or be within a subpoint difference, such as Konqueror 3.5 using KHTML 3.5.1.

Chrome

Google's Chrome web browser uses WebKit as its rendering engine but uses a different JavaScript engine. Chrome's user-agent string carries along all of the information from WebKit and an extra section for the Chrome version. The format is as follows:

```
Mozilla/5.0 (Platform; Encryption; OS-or-CPU; Language)
AppleWebKit/AppleWebKitVersion (KHTML, like Gecko)
Chrome/ChromeVersion Safari/SafariVersion
```

The full user-agent string for Chrome 7 is as follows:

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/534.7
(KHTML, like Gecko) Chrome/7.0.517.44 Safari/534.7
```

It's likely that the WebKit version and Safari version will always be synchronized going forward, though this is not guaranteed.

Opera

One of the most controversial web browsers, as far as user-agent strings are concerned, is Opera. The default user-agent string for Opera is the most logical of all modern browsers, correctly identifying itself and its version. Prior to version 8, the Opera user-agent string was in the following format:

Opera/Version (OS-or-CPU; Encryption) [Language]

Using Opera 7.54 on a Windows XP computer, the user-agent string is as follows:

Opera/7.54 (Windows NT 5.1; U) [en]

With the release of Opera 8, the language part of the user-agent string was moved inside of the parentheses to better match other browsers, as follows:

Opera/Version (OS-or-CPU; Encryption; Language)

Opera 8 on Windows XP yields the following user-agent string:

Opera/8.0 (Windows NT 5.1; U; en)

By default, Opera returns a user-agent string in this simple format. Currently it is the only one of the major browsers to use the product name and version to fully and completely identify itself. As with other browsers, however, Opera found problems with using its own user-agent string. Even though it's technically correct, there is a lot of browser-sniffing code on the Internet that is geared toward user-agent strings reporting the Mozilla product name. There is also a fair amount of code looking specifically for Internet Explorer or Gecko. Instead of confusing sniffers by changing its own user-agent string, Opera identifies itself as a different browser completely by changing its own user-agent string.

As of Opera 9, there are two ways to change the user-agent string. One way is to identify it as another browser, either Firefox or Internet Explorer. When using this option, the user-agent string changes to look just like the corresponding one for Firefox or Internet Explorer, with the addition of the string "Opera" and Opera's version number at the end. Here's an example:

Mozilla/5.0 (Windows NT 5.1; U; en; rv:1.8.1) Gecko/20061208 Firefox/2.0.0
Opera 9.50

Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; en) Opera 9.50

The first string identifies Opera 9.5 as Firefox 2 while maintaining the Opera version information. The second string identifies Opera 9.5 as Internet Explorer 6 and includes the Opera version information. Although these user-agent strings pass most tests for Firefox and Internet Explorer, the possibility of identifying Opera is open.

Another option for identifying the browser is to mask it as either Firefox or Internet Explorer. When masking the browser's identity, the user-agent strings are exactly the same as would be returned from the other browsers — the string "Opera" does not appear, nor does any Opera version information. There is literally no way to distinguish Opera from the other browsers when identity masking is used. Further complicating the issue is Opera's tendency to set site-specific user-agent strings without notifying the user. For instance, navigating to the My Yahoo! site (<http://my.yahoo.com>) automatically causes Opera to mask itself as Firefox. This makes identifying Opera by user-agent string very difficult.



Before version 7, Opera could interpret the meaning of Windows operating-system strings. For example, Windows NT 5.1 actually means Windows XP, so in Opera 6, the user-agent string included Windows XP instead of Windows NT 5.1. In an effort to be more compatible with other browsers, version 7 started including the officially reported operating-system version instead of an interpreted one.

Opera 10 introduced changes to its user-agent string. The format is now:

```
Opera/9.80 (OS-or-CPU; Encryption; Language) Presto/PrestoVersion Version/Version
```

Note that the initial version, Opera/9.80, remains fixed. There was no Opera 9.8, but Opera engineers were afraid that poor browser sniffing might cause a token of Opera/10.0 to be incorrectly interpreted as Opera 1 instead of Opera 10. Thus, Opera 10 introduced the additional Presto token (Presto is the rendering engine for Opera) and the Version token to hold the actual browser version. This is the user-agent string for Opera 10.63 on Windows 7:

```
Opera/9.80 (Windows NT 6.1; U; en) Presto/2.6.30 Version/10.63
```

iOS and Android

The default web browsers for both iOS and Android mobile operating systems are based on WebKit and so share the same basic user-agent string format as their desktop counterparts. iOS devices follow this basic format:

```
Mozilla/5.0 (Platform; Encryption; OS-or-CPU like Mac OS X; Language)
AppleWebKit/AppleWebKitVersion (KHTML, like Gecko) Version/BrowserVersion
Mobile/MobileVersion Safari/SafariVersion
```

Note the addition of "like Mac OS X" to aid in detecting Mac operating systems and the addition of a Mobile token. The Mobile token version number is typically not useful and is used primarily to determine a mobile WebKit versus a desktop one. The platform will be "iPhone", "iPod", or "iPad", depending on the device. Example:

```
Mozilla/5.0 (iPhone; U; CPU iPhone OS 3_0 like Mac OS X; en-us)
AppleWebKit/528.18 (KHTML, like Gecko) Version/4.0 Mobile/7A341 Safari/528.16
```

Note that prior to iOS 3, the version number of the operating system did not appear in the user-agent string.

The default Android browser generally follows the format set forth on iOS but without a Mobile version (the Mobile token is still present). For example:

```
Mozilla/5.0 (Linux; U; Android 2.2; en-us; Nexus One Build/FRF91)
AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533.1
```

This user-agent string is from a Google Nexus One phone, but other Android devices follow the same pattern.

Working with User-Agent Detection

Using the user-agent string to detect specific browsers can get quite complicated because of the history and usage of user-agent strings in modern browsers. It's often necessary to first determine how specific you need the browser information to be. Typically, knowing the rendering engine and a minimum version is enough to determine the correct course of action. For instance, the following is not recommended:

```
if (isIE6 || isIE7) {    //avoid!!!
    //code
}
```

In this example, code is executed if the browser is Internet Explorer version 6 or 7. This code is very fragile because it relies on specific browser versions to determine what to do. What should happen for version 8? Anytime a new version of Internet Explorer is released, this code would have to be updated. However, using relative version numbers as shown in the following example avoids this problem:

```
if (ieVer >= 6) {
    //code
}
```

This rewritten example checks to see if the version of Internet Explorer is at least 6 to determine the correct course of action. Doing so ensures that this code will continue functioning appropriately in the future. The browser-detection script focuses on this methodology for identifying browsers.

Identifying the Rendering Engine

As mentioned previously, the exact name and version of a browser isn't as important as the rendering engine being used. If Firefox, Camino, and Netscape all use the same version of Gecko, their capabilities will be the same. Likewise, any browser using the same version of WebKit that Safari 3 uses will likely have the same capabilities. Therefore, this script focuses on detecting the five major rendering engines: Internet Explorer, Gecko, WebKit, KHTML, and Opera.

This script uses the module-augmentation pattern to encapsulate the detection script and avoid adding unnecessary global variables. The basic code structure is as follows:

```

var client = function(){
    var engine = {

        //rendering engines
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,

        //specific version
        ver: null
    };

    //detection of rendering engines/platforms/devices here

    return {
        engine: engine
    };
}();

```

In this code, a global variable named `client` is declared to hold the information. Within the anonymous function is a local variable named `engine` that contains an object literal with some default settings. Each rendering engine is represented by a property that is set to 0. If a particular engine is detected, the version of that engine will be placed into the corresponding property as a floating-point value. The full version of the rendering engine (a string) is placed into the `ver` property. This setup allows code such as the following:

```

if (client.engine.ie) { //if it's IE, client.engine.ie is greater than 0
    //IE-specific code
} else if (client.engine.gecko > 1.5){
    if (client.engine.ver == "1.8.1"){
        //do something specific to this version
    }
}

```

Whenever a rendering engine is detected, its property on `client.engine` gets set to a number greater than 0, which converts to a Boolean true. This allows a property to be used with an `if` statement to determine the rendering engine being used, even if the specific version isn't necessary. Since each property contains a floating-point value, it's possible that some version information may be lost. For instance, the string "1.8.1" becomes the number 1.8 when passed into `parseFloat()`. The `ver` property ensures that the full version is available if necessary.

To identify the correct rendering engine, you need to test in the correct order. Testing out of order may result in incorrect results because of the user-agent inconsistencies. For this reason, the first step is to identify Opera, since its user-agent string may completely mimic other browsers. Opera's user-agent string cannot be trusted since it won't, in all cases, identify itself as Opera.

To identify Opera, you need to look for the `window.opera` object. This object is present in all versions of Opera 5 and later and is used to identify information about the browser and to interact directly with the browser. In versions later than 7.6, a method called `version()` returns the browser version number as a string, which is the best way to determine the Opera version number. Earlier versions may be detected using the user-agent string, since identity masking wasn't supported. However, since Opera's most recent version at the end of 2010 was 10.63, it's unlikely that anyone is using a version older than 7.6. The first step in the rendering engine's detection code is as follows:

```
if (window.opera) {
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
}
```

The string representation of the version is stored in `engine.ver`, and the floating-point representation is stored in `engine.opera`. If the browser is Opera, the test for `window.opera` will return `true`. Otherwise, it's time to detect another browser.

The next logical rendering engine to detect is WebKit. Since WebKit's user-agent string contains "Gecko" and "KHTML", incorrect results could be returned if you were to check for those rendering engines first.

WebKit's user-agent string, however, is the only one to contain the string "AppleWebKit", so it's the most logical one to check for. The following is an example of how to do this:

```
var ua = navigator.userAgent;

if (window.opera) {
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);
}
```

This code begins by storing the user-agent string in a variable called `ua`. A regular expression tests for the presence of "AppleWebKit" in the user-agent string and uses a capturing group around the version number. Since the actual version number may contain a mixture of numbers, decimal points, and letters, the non-white-space special character (`\S`) is used. The separator between the version number and the next part of the user-agent string is a space, so this pattern ensures all of the versions will be captured. The `test()` method runs the regular expression against the user-agent string. If it returns `true`, then the captured version number is stored in `engine.ver` and the floating-point representation is stored in `engine.webkit`. WebKit versions correspond to Safari versions, as detailed in the following table.

SAFARI VERSION	MINIMUM WEBKIT VERSION
1.0 through 1.0.2	85.7
1.0.3	85.8.2
1.1 through 1.1.1	100
1.2.2	125.2
1.2.3	125.4
1.2.4	125.5.5
1.3	312.1
1.3.1	312.5
1.3.2	312.8
2.0	412
2.0.1	412.7
2.0.2	416.11
2.0.3	417.9
2.0.4	418.8
3.0.4	523.10
3.1	525



Sometimes Safari versions don't match up exactly to WebKit versions and may be a subpoint off. The preceding table indicates the most-likely WebKit versions but is not exact.

The next rendering engine to test for is KHTML. Once again, this user-agent string contains "Gecko", so you cannot accurately detect a Gecko-based browser before first ruling out KHTML. The KHTML version is included in the user-agent string in a format similar to WebKit, so a similar regular expression is used. Also, since Konqueror 3.1 and earlier don't include the KHTML version specifically, the Konqueror version is used instead. Here's an example:

```
var ua = navigator.userAgent;

if (window.opera){
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/(\S+).test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);
```

```

} else if (/KHTML\//(\S+).test(ua) || /Konqueror\//([^\;]+).test(ua)) {
    engine.ver = RegExp["$1"];
    engine.html = parseFloat(engine.ver);
}

```

Once again, since the KHTML version number is separated from the next token by a space, the non-white-space character is used to grab all of the characters in the version. Then the string version is stored in `engine.ver`, and the floating-point version is stored in `engine.html`. If KHTML isn't in the user-agent string, then the match is against Konqueror, followed by a slash, followed by all characters that aren't a semicolon.

If both WebKit and KHTML have been ruled out, it is safe to check for Gecko. The actual Gecko version does not appear after the string "Gecko" in the user-agent; instead, it appears after the string "rv:". This requires a more complicated regular expression than the previous tests, as you can see in the following example:

```

var ua = navigator.userAgent;

if (window.opera) {
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\//(\S+).test(ua)) {
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);
} else if (/KHTML\//(\S+).test(ua)) {
    engine.ver = RegExp["$1"];
    engine.html = parseFloat(engine.ver);
} else if (/rv:([^\)]+)\) Gecko\/\d{8}.test(ua)){
    engine.ver = RegExp["$1"];
    engine.gecko = parseFloat(engine.ver);
}

```

The Gecko version number appears between "rv:" and a closing parenthesis, so to extract the version number, the regular expression looks for all characters that are not a closing parenthesis. The regular expression also looks for the string "Gecko/" followed by eight numbers. If the pattern matches, then the version number is extracted and stored in the appropriate properties. Gecko version numbers are related to Firefox versions, as detailed in the following table.

FIREFOX VERSION	MINIMUM GECKO VERSION
1.0	1.7.5
1.5	1.8.0
2.0	1.8.1
3.0	1.9.0
3.5	1.9.1
3.6	1.9.2
4.0	2.0.0



As with Safari and WebKit, matches between Firefox and Gecko version numbers are not exact.

Internet Explorer is the last rendering engine to detect. The version number is found following "MSIE" and before a semicolon, so the regular expression is fairly simple, as you can see in the following example:

```
var ua = navigator.userAgent;

if (window.opera){
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);
} else if (/KHTML\/\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.khtml = parseFloat(engine.ver);
} else if (/rv:([^\\]+)\) Gecko\/\d{8}/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.gecko = parseFloat(engine.ver);
} else if (/MSIE ([^;]+)\.test(ua)){
    engine.ver = RegExp["$1"];
    engine.ie = parseFloat(engine.ver);
}
```

The last part of this rendering engine's detection script uses a negation class in the regular expression to get all characters that aren't a semicolon. Even though Internet Explorer typically keeps version numbers as standard floating-point values, that won't necessarily always be so. The negation class `[^;]` is used to allow for multiple decimal points and possibly letters.

Identifying the Browser

In most cases, identifying the browser's rendering engine is specific enough to determine a correct course of action. However, the rendering engine alone doesn't indicate that JavaScript functionality is present. Apple's Safari browser and Google's Chrome browser both use WebKit as their rendering engine but use different JavaScript engines. Both browsers would return a value for `client.engine.webkit`, but that may not be specific enough. For these two browsers, it's helpful to add new properties to the `client` object, as shown in the following example:

```
var client = function(){

    var engine = {

        //rendering engines
        ie: 0,
        gecko: 0,
        webkit: 0,
```

```
        khtml: 0,
        opera: 0,

        //specific version
        ver: null
    };

    var browser = {

        //browsers
        ie: 0,
        firefox: 0,
        safari: 0,
        konq: 0,
        opera: 0,
        chrome: 0,


        //specific version
        ver: null
    };

    //detection of rendering engines/platforms/devices here

    return {
        engine: engine,
        browser: browser
    };
};

}();
```

This code adds a private variable called `browser` that contains properties for each of the major browsers. As with the `engine` variable, these properties remain zero unless the browser is being used, in which case the floating-point version is stored in the property. Also, the `ver` property contains the full string version of the browser in case it's necessary. As you can see in the following example, the detection code for browsers is intermixed with the rendering-engine-detection code because of the tight coupling between most browsers and their rendering engines:

```
//detect rendering engines/browsers
var ua = navigator.userAgent;
if (window.opera){
    engine.ver = browser.ver = window.opera.version();
    engine.opera = browser.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);

    //figure out if it's Chrome or Safari
    if (/Chrome\/(\S+)/.test(ua)){
        browser.ver = RegExp["$1"];
        browser.chrome = parseFloat(browser.ver);
    } else if (/Version\/(\S+)/.test(ua)){
        browser.ver = RegExp["$1"];
```

```

        browser.safari = parseFloat(browser.ver);
    } else {
        //approximate version
        var safariVersion = 1;
        if (engine.webkit < 100){
            safariVersion = 1;
        } else if (engine.webkit < 312){
            safariVersion = 1.2;
        } else if (engine.webkit < 412){
            safariVersion = 1.3;
        } else {
            safariVersion = 2;
        }

        browser.safari = browser.ver = safariVersion;
    }
} else if (/KHTML\//(\S+).test(ua) || /Konqueror\//([^\;]+).test(ua)){
    engine.ver = browser.ver = RegExp["$1"];
    engine.khtml = browser.konq = parseFloat(engine.ver);
} else if (/rv:([^\;]+)\) Gecko\/\d{8}/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.gecko = parseFloat(engine.ver);

    //determine if it's Firefox
    if (/Firefox\//(\S+).test(ua)){
        browser.ver = RegExp["$1"];
        browser.firefox = parseFloat(browser.ver);
    }
} else if (/MSIE ([^\;]+).test(ua)){
    engine.ver = browser.ver = RegExp["$1"];
    engine.ie = browser.ie = parseFloat(engine.ver);
}

```

For Opera and Internet Explorer, the values in the `browser` object are equal to those in the `engine` object. For Konqueror, the `browser.konq` and `browser.ver` properties are equivalent to the `engine.khtml` and `engine.ver` properties, respectively.

To detect Chrome and Safari, add additional `if` statements into the engine-detection code. The version number for Chrome is extracted by looking for the string `"Chrome/"` and then taking the numbers after that. Safari detection is done by looking for the `"Version/"` string and taking the number after that. Since this works only for Safari versions 3 and higher, there's some fallback logic to map WebKit version numbers to the approximate Safari version numbers (see the table in the previous section).

For the Firefox version, the string `"Firefox/"` is found and the numbers after it are extracted as the version number. This happens only if the detected rendering engine is Gecko.

Using this code, you can now write logic such as the following:

```

if (client.engine.webkit) { //if it's WebKit
    if (client.browser.chrome){
        //do something for Chrome
    } else if (client.browser.safari){
        //do something for Safari
    }
}

```

```
    } else if (client.engine.gecko){
        if (client.browser.firefox){
            //do something for Firefox
        } else {
            //do something for other Gecko browsers
        }
    }
}
```

Identifying the Platform

In many cases, simply knowing the rendering engine is enough to get your code working. In some circumstances, however, the platform is of particular interest. Browsers that are available cross-platform (such as Safari, Firefox, and Opera) may have different issues on different platforms. The three major platforms are Windows, Mac, and Unix (including flavors of Linux). To allow for detection of these platforms, add a new object to `client` as follows:

```
var client = function(){

    var engine = {

        //rendering engines
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,

        //specific version
        ver: null
    };

    var browser = {

        //browsers
        ie: 0,
        firefox: 0,
        safari: 0,
        konq: 0,
        opera: 0,
        chrome: 0,

        //specific version
        ver: null
    };

    var system = {
        win: false,
        mac: false,
        x11: false
    };

    //detection of rendering engines/platforms/devices here

    return {

```

```

        engine: engine,
        browser: browser,
system: system
    };
}

}();

```

This code introduces a new `system` variable that has three properties. The `win` property indicates if the platform is Windows, `mac` indicates Mac, and `x11` indicates Unix. Unlike rendering engines, platform information is typically very limited, without access to operating systems or versions. Of these three platforms, browsers regularly report only Windows versions. For this reason, each of these properties is represented initially by a Boolean `false` instead of a number (as with the rendering-engine properties).

To determine the platform, it's much easier to look at `navigator.platform` than to look at the user-agent string, which may represent platform information differently across browsers. The possible values for `navigator.platform` are "Win32", "Win64", "MacPPC", "MacIntel", "X11", and "Linux i686", which are consistent across browsers. The platform-detection code is very straightforward, as you can see in the following example:

```

var p = navigator.platform;
system.win = p.indexOf("Win") == 0;
system.mac = p.indexOf("Mac") == 0;
system.x11 = (p.indexOf("X11") == 0) || (p.indexOf("Linux") == 0);

```

This code uses the `indexOf()` method to look at the beginning of the platform string. To detect Windows, the platform-detection code simply looks for the string "Win" at the beginning of the platform string (covers both "Win32" and "Win64"). Testing for a Mac platform is done in the same way to accommodate both "MacPPC" and "MacIntel". The test for Unix looks for both "X11" and "Linux" at the beginning of the platform string to future-proof this code against other variants.



Earlier versions of Gecko returned "Windows" for all Windows platforms and "Macintosh" for all Mac platforms. This occurred prior to the release of Firefox 1, which stabilized `navigator.platform` values.

Identifying Windows Operating Systems

If the platform is Windows, it's possible to get specific operating-system information from the user-agent string. Prior to Windows XP, there were two versions of Windows: one for home use and one for business use. The version for home use was simply called Windows and had specific versions of 95, 98, and ME. The business version was called Windows NT and eventually was marketed as Windows 2000. Windows XP represented the convergence of these two product lines into a common code base evolved from Windows NT. Windows Vista then was built on Windows XP.

This information is important because of the way a Windows operating system is represented in the user-agent string. The following table shows the different strings used to represent the various Windows operating systems across browsers.

WINDOWS VERSION	IE 4+	GECKO	OPERA < 7	OPERA 7+	WEBKIT
95	"Windows 95"	"Win95"	"Windows 95"	"Windows 95"	n/a
98	"Windows 98"	"Win98"	"Windows 98"	"Windows 98"	n/a
NT 4.0	"Windows NT"	"WinNT4.0"	"Windows NT 4.0"	"Windows NT 4.0"	n/a
2000	"Windows NT 5.0"	"Windows NT 5.0"	"Windows 2000"	"Windows NT 5.0"	n/a
ME	"Win 9x 4.90"	"Win 9x 4.90"	"Windows ME"	"Win 9x 4.90"	n/a
XP	"Windows NT 5.1"	"Windows NT 5.1"	"Windows XP"	"Windows NT 5.1"	"Windows NT 5.1"
Vista	"Windows NT 6.0"	"Windows NT 6.0"	n/a	"Windows NT 6.0"	"Windows NT 6.0"
7	"Windows NT 6.1"	"Windows NT 6.1"	n/a	"Windows NT 6.1"	"Windows NT 6.1"

Because of the various ways the Windows operating system is represented in the user-agent string, detection isn't completely straightforward. The good news is that since Windows 2000, the string representation has remained mostly the same, with only the version number changing. To detect the different Windows operating systems, you need a regular expression. Keep in mind that Opera versions prior to 7 are no longer in significant use, so there's no need to prepare for them.

The first step is to match the strings for Windows 95 and Windows 98. The only difference between the strings returned by Gecko and the other browsers is the absence of "dows" and a space between "win" and the version number. This is a fairly easy regular expression, as you can see here:

```
/Win(?:dows )?([^\do]{2})/
```

Using this regular expression, the capturing group returns the operating-system version. Since this may be any two-character code not containing "d" or "o" (such as 95, 98, 9x, NT, ME, or XP) two non-white-space characters are used.

The Gecko representation for Windows NT adds a "4.0" at the end. Instead of looking for that exact string, it makes more sense to look for a decimal number like this:

```
/Win(?:dows )?([^\d]{2})(\d+\.\d+)?/
```

This regular expression introduces a second capturing group to get the NT version number. Since that number won't be there for Windows 95 or 98, it must be optional. The only difference between this pattern and the Opera representation of Windows NT is the space between "NT" and "4.0", which can easily be added as follows:

```
/Win(?:dows )?([^\d]{2})\s?(\d+\.\d+)?/
```

With these changes, the regular expression will also successfully match the strings for Windows ME, Windows XP, and Windows Vista. The first capturing group will capture 95, 98, 9x, NT, ME, or XP. The second capturing group is used only for Windows ME and all Windows NT derivatives. This information can be used to assign specific operating-system information to the `system.win` property, as in the following example:

```
if (system.win) {
    if (/Win(?:dows )?([^\d]{2})\s?(\d+\.\d+)?/.test(ua)) {
        if (RegExp["$1"] == "NT") {
            switch(RegExp["$2"]){
                case "5.0":
                    system.win = "2000";
                    break;
                case "5.1":
                    system.win = "XP";
                    break;
                case "6.0":
                    system.win = "Vista";
                    break;
                case "6.1":
                    system.win = "7";
                    break;
                default:
                    system.win = "NT";
                    break;
            }
        } else if (RegExp["$1"] == "9x") {
            system.win = "ME";
        } else {
            system.win = RegExp["$1"];
        }
    }
}
```

If `system.win` is true, then the regular expression is used to extract specific information from the user-agent string. It's possible that some future version of Windows won't be detectable via this method, so the first step is to check if the pattern is matched in the user-agent string. When the pattern matches, the first capturing group will contain one of the following: "95", "98", "9x", or "NT". If the value is "NT", then `system.win` is set to a specific string for the operating system in question; if

the value is "9x", then `system.win` is set to "ME"; otherwise the captured value is assigned directly to `system.win`. This setup allows code such as the following:

```
if (client.system.win){  
    if (client.system.win == "XP") {  
        //report XP  
    } else if (client.system.win == "Vista"){  
        //report Vista  
    }  
}
```

Since a nonempty string converts to the Boolean value of `true`, the `client.system.win` property can be used as a Boolean in an `if` statement. When additional information about the operating system is necessary, the string value can be used.

Identifying Mobile Devices

In 2006–2007, the use of web browsers on mobile devices exploded. There are mobile versions of all major browsers, and versions that run on other devices, so it's important to identify these cases. The first step is to add properties for all of the mobile devices to detect for, as in the following example:

```
var client = function(){  
  
    var engine = {  
  
        //rendering engines  
        ie: 0,  
        gecko: 0,  
        webkit: 0,  
        khtml: 0,  
        opera: 0,  
  
        //specific version  
        ver: null  
    };  
  
    var browser = {  
  
        //browsers  
        ie: 0,  
        firefox: 0,  
        safari: 0,  
        kong: 0,  
        opera: 0,  
        chrome: 0,  
  
        //specific version  
        ver: null  
    };  
  
    var system = {  
        win: false,  
        mac: false,
```

```

        x11: false,

        //mobile devices
        iphone: false,
        ipod: false,
        ipad: false,
        ios: false,
        android: false,
        nokiaN: false,
        winMobile: false    };

//detection of rendering engines/platforms/devices here

return {
    engine: engine,
    browser: browser,
    system: system
};

}();

```

Detecting iOS devices is as simple as searching for the strings "iPhone", "iPod", and "iPad":

```

system.iphone = ua.indexOf("iPhone") > -1;
system.ipod = ua.indexOf("iPod") > -1;
system.ipad = ua.indexOf("iPad") > -1;

```

In addition to knowing the iOS device, it's also helpful to know the version of iOS. Prior to iOS 3, the user-agent string simply said "CPU like Mac OS X", while later it was changed to "CPU iPhone OS 3_0 like Mac OS X" for iPhone and "CPU OS 3_2 like Mac OS X" for the iPad. This means detecting iOS requires a regular expression to take these changes into account:

```

//determine iOS version
if (system.mac && ua.indexOf("Mobile") > -1){
    if (/CPU(?:iPhone)?OS (\d+\.\d+)/.test(ua)){
        system.ios = parseFloat(RegExp.$1.replace("_", "."));
    } else {
        system.ios = 2; //can't really detect - so guess
    }
}

```

Checking to see if the system is a Mac OS and if the string "Mobile" is present ensures that `system.ios` will be nonzero regardless of the version. After that, a regular expression is used to determine if the iOS version is present in the user-agent string. If it is, then `system.ios` is set to a floating-point value for the version; otherwise, the version is hardcoded to 2. (There is no way to determine the actual version, so picking the previous version is safe for making this value useful.)

Detecting the Android operating system is a simple search for the string "Android" and retrieving the version number immediately after:

```

//determine Android version
if (/Android (\d+)\.\d+/.test(ua)){
    system.android = parseFloat(RegExp.$1);
}

```

Since all versions of Android include the version number, this regular expression accurately detects all versions and sets `system.android` to the correct value.

Nokia Nseries mobile phones also use WebKit. The user-agent string is very similar to other WebKit-based phones, such as the following:

```
Mozilla/5.0 (SymbianOS/9.2; U; Series60/3.1 NokiaN95/11.0.026;
Profile MIDP-2.0 Configuration/CLDC-1.1) AppleWebKit/413 (KHTML, like Gecko)
Safari/413
```

Note that even though the Nokia Nseries phones report "Safari" in the user-agent string, the browser is not actually Safari though it is WebKit-based. A simple check for "NokiaN" in the user-agent string, as shown here, is sufficient to detect this series of phones:

```
system.nokiaN = ua.indexOf("NokiaN") > -1;
```

With this device information, it's possible to figure out how the user is accessing a page with WebKit by using code such as this:

```
if (client.engine.webkit){
    if (client.system.ios){
        //iOS stuff
    } else if (client.system.android){
        //android stuff
    } else if (client.system.nokiaN){
        //nokia stuff
    }
}
```

The last major mobile-device platform is Windows Mobile (previously called Windows CE), which is available on both Pocket PCs and smartphones. Since these devices are technically a Windows platform, the Windows platform and operating system will return correct values. For Windows Mobile 5.0 and earlier, the user-agent strings for these two devices were very similar, such as the following:

```
Mozilla/4.0 (compatible; MSIE 4.01; Windows CE; PPC; 240x320)
Mozilla/4.0 (compatible; MSIE 4.01; Windows CE; Smartphone; 176x220)
```

The first of these is mobile Internet Explorer 4.01 on the Pocket PC, and the second one is the same browser on a smartphone. When the Windows operating system detection script is run against either of these strings, `client.system.win` gets filled with "CE", so detection for early Windows Mobile devices can be done using this value.

It's not advisable to test for "PPC" or "Smartphone" in the string, because these tokens have been removed in browsers on Windows Mobile later than 5.0. Oftentimes, simply knowing that the device is using Windows Mobile is enough.

Windows Phone 7 features a slightly augmented user-agent string with the following basic format:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows Phone OS 7.0; Trident/3.1; IEMobile/7.0)
Asus;Galaxy6
```

The format of the Windows operating system identifier breaks from tradition, so the value of `client.system.win` is equal to "Ph" when this user-agent is encountered. This information can be used to get more information about the system:

```
//windows mobile
if (system.win == "CE"){
    system.winMobile = system.win;
} else if (system.win == "Ph"){
    if(/Windows Phone OS (\d+\.\d+)/.test(ua)){
        system.win = "Phone";
        system.winMobile = parseFloat(RegExp["$1"]);
    }
}
```

If the value of `system.win` is "CE", that means it's an older version of Windows Mobile, so `system.winMobile` is set to that value (it's the only information you have). If `system.win` is "Ph", then the device is probably Windows Phone 7 or later, so another regular expression is used to test for the format and extract the version number. The value of `system.win` is then reset to "Phone" and `system.winMobile` is set to the version number.

Identifying Game Systems

Another new area in which web browsers have become increasingly popular is on video game systems. Both the Nintendo Wii and Playstation 3 have web browsers either built in or available for download. The Wii browser is actually a custom version of Opera, designed specifically for use with the Wii remote. The Playstation browser is custom and is not based on any of the rendering engines previously mentioned. The user-agent strings for these browsers are as follows:

```
Opera/9.10 (Nintendo Wii;U; ; 1621; en)
Mozilla/5.0 (PLAYSTATION 3; 2.00)
```

The first user-agent string is Opera running on the Wii. It stays true to the original Opera user-agent string (keep in mind that Opera on the Wii does not have identity-masking capabilities). The second string is from a Playstation 3, which reports itself as Mozilla 5.0 for compatibility but doesn't give much information. Oddly, it uses all uppercase letters for the device name, prompting concerns that future versions may change the case.

Before detecting these devices, you must add appropriate properties to the `client.system` object as follows:

```
var client = function(){
    var engine = {
        //rendering engines
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,
        //specific version
    };
    ...
}
```

```
        ver: null
    };

var browser = {

    //browsers
    ie: 0,
    firefox: 0,
    safari: 0,
    kong: 0,
    opera: 0,
    chrome: 0,

    //specific version
    ver: null
};

var system = {
    win: false,
    mac: false,
    x11: false,

    //mobile devices
    iphone: false,
    ipod: false,
    ipad: false,
    ios: false,
    android: false,
    nokiaN: false,
    winMobile: false,
    //game systems
    wii: false,
    ps: false
};

//detection of rendering engines/platforms/devices here

return {
    engine: engine,
    browser: browser,
    system: system
};

}();
```

The following code detects each of these game systems:

```
system.wii = ua.indexOf("Wii") > -1;
system.ps = /playstation/i.test(ua);
```

For the Wii, a simple test for the string "Wii" is enough. The rest of the code will pick up that the browser is Opera and return the correct version number in `client.browser.opera`. For the Playstation, a regular expression is used to test against the user-agent string in a case-insensitive way.

The Complete Script

The complete user-agent detection script, including rendering engines, platforms, Windows operating systems, mobile devices, and game systems is as follows:



Available for
download on
Wrox.com

```
var client = function(){

    //rendering engines
    var engine = {
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,

        //complete version
        ver: null
    };

    //browsers
    var browser = {

        //browsers
        ie: 0,
        firefox: 0,
        safari: 0,
        konq: 0,
        opera: 0,
        chrome: 0,

        //specific version
        ver: null
    };

    //platform/device/OS
    var system = {
        win: false,
        mac: false,
        x11: false,

        //mobile devices
        iphone: false,
        ipod: false,
        ipad: false,
        ios: false,
        android: false,
        nokiaN: false,
        winMobile: false,

        //game systems
        wii: false,
        ps: false
    };

    //detect rendering engines/browsers
}
```

```
var ua = navigator.userAgent;
if (window.opera){
    engine.ver = browser.ver = window.opera.version();
    engine.opera = browser.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);

    //figure out if it's Chrome or Safari
    if (/Chrome\/\/(\S+)/.test(ua)){
        browser.ver = RegExp["$1"];
        browser.chrome = parseFloat(browser.ver);
    } else if (/Version\/\/(\S+)/.test(ua)){
        browser.ver = RegExp["$1"];
        browser.safari = parseFloat(browser.ver);
    } else {
        //approximate version
        var safariVersion = 1;
        if (engine.webkit < 100){
            safariVersion = 1;
        } else if (engine.webkit < 312){
            safariVersion = 1.2;
        } else if (engine.webkit < 412){
            safariVersion = 1.3;
        } else {
            safariVersion = 2;
        }

        browser.safari = browser.ver = safariVersion;
    }
} else if (/KHTML\/\/(\S+)/.test(ua) || /Konqueror\/\/([^\;]+)\.test(ua)){
    engine.ver = browser.ver = RegExp["$1"];
    engine.khtml = browser.konq = parseFloat(engine.ver);
} else if (/rv:([^\;]+)\) Gecko\/\d{8}\.test(ua)){
    engine.ver = RegExp["$1"];
    engine.gecko = parseFloat(engine.ver);

    //determine if it's Firefox
    if (/Firefox\/\/(\S+)/.test(ua)){
        browser.ver = RegExp["$1"];
        browser.firefox = parseFloat(browser.ver);
    }
} else if (/MSIE ([^\;]+)\.test(ua)){
    engine.ver = browser.ver = RegExp["$1"];
    engine.ie = browser.ie = parseFloat(engine.ver);
}

//detect browsers
browser.ie = engine.ie;
browser.opera = engine.opera;

//detect platform
var p = navigator.platform;
```

```

system.win = p.indexOf("Win") == 0;
system.mac = p.indexOf("Mac") == 0;
system.x11 = (p == "X11") || (p.indexOf("Linux") == 0);

//detect windows operating systems
if (system.win){
    if (/Win(?:dows )?([^\d]{2})\s?(\d+\.\d+)?/.test(ua)){
        if (RegExp["$1"] == "NT"){
            switch(RegExp["$2"]){
                case "5.0":
                    system.win = "2000";
                    break;
                case "5.1":
                    system.win = "XP";
                    break;
                case "6.0":
                    system.win = "Vista";
                    break;
                case "6.1":
                    system.win = "7";
                    break;
                default:
                    system.win = "NT";
                    break;
            }
        } else if (RegExp["$1"] == "9x"){
            system.win = "ME";
        } else {
            system.win = RegExp["$1"];
        }
    }
}

//mobile devices
system.iphone = ua.indexOf("iPhone") > -1;
system.ipod = ua.indexOf("iPod") > -1;
system.ipad = ua.indexOf("iPad") > -1;
system.nokiaN = ua.indexOf("NokiaN") > -1;

//windows mobile
if (system.win == "CE"){
    system.winMobile = system.win;
} else if (system.win == "Ph"){
    if(/Windows Phone OS (\d+\.\d+)/.test(ua)){
        system.win = "Phone";
        system.winMobile = parseFloat(RegExp["$1"]);
    }
}
//determine iOS version
if (system.mac && ua.indexOf("Mobile") > -1){
    if (/CPU (?:iPhone )?OS (\d+\.\d+)/.test(ua)){
        system.ios = parseFloat(RegExp.$1.replace("_", "."));
    } else {
}

```

```
        system.ios = 2; //can't really detect - so guess
    }
}

//determine Android version
if (/Android (\d+\.\d+)/.test(ua)){
    system.android = parseFloat(RegExp.$1);
}

//gaming systems
system.wii = ua.indexOf("Wii") > -1;
system.ps = /playstation/i.test(ua);

//return it
return {
    engine:     engine,
    browser:   browser,
    system:    system
};

}();
```

client.js

Usage

As mentioned previously, user-agent detection is considered the last option for client detection. Whenever possible, capability detection and/or quirks detection should be used first. User-agent detection is best used under the following circumstances:

- If a capability or quirk cannot be accurately detected directly. For example, some browsers implement functions that are stubs for future functionality. In that case, testing for the existence of the function doesn't give you enough information.
- If the same browser has different capabilities on different platforms. It may be necessary to determine which platform is being used.
- If you need to know the exact browser for tracking purposes.

SUMMARY

Client detection is one of the most controversial topics in JavaScript. Because of differences in browsers, it is often necessary to fork code based on the browser's capabilities. There are several approaches to client detection, but the following three are used most frequently:

- **Capability detection** — Tests for specific browser capabilities before using them. For instance, a script may check to see if a function exists before calling it. This approach frees developers from worrying about specific browser types and versions, letting them simply focus on whether the capability exists or not. Capabilities detection cannot accurately detect a specific browser or version.

- **Quirks detection** — Quirks are essentially bugs in browser implementations, such as WebKit's early quirk of returning shadowed properties in a `for-in` loop. Quirks detection often involves running a short piece of code to determine if the browser has the particular quirk. Since it is less efficient than capability detection, quirks detection is used only when a specific quirk may interfere with the processing of the script. Quirks detection cannot detect a specific browser or version.
- **User-agent detection** — Identifies the browser by looking at its user-agent string. The user-agent string contains a great deal of information about the browser, often including the browser, platform, operating system, and browser version. There is a long history to the development of the user-agent string, with browser vendors attempting to fool web sites into believing they are another browser. User-agent detection can be tricky, especially when dealing with Opera's ability to mask its user-agent string. Even so, the user-agent string can determine the rendering engine being used and the platform on which it runs, including mobile devices and gaming systems.

When you are deciding which client-detection method to use, it's preferable to use capability detection first. Quirks detection is the second choice for determining how your code should proceed. User-agent detection is considered the last choice for client detection, because it is so dependent on the user-agent string.

10

The Document Object Model

WHAT'S IN THIS CHAPTER?

- Understanding the DOM as a hierarchy of nodes
- Working with the various node types
- Coding the DOM around browser incompatibilities and gotchas

The Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents. The DOM represents a document as a hierarchical tree of nodes, allowing developers to add, remove, and modify individual parts of the page. Evolving out of early Dynamic HTML (DHTML) innovations from Netscape and Microsoft, the DOM is now a truly cross-platform, language-independent way of representing and manipulating pages for markup.

DOM Level 1 became a W3C recommendation in October 1998, providing interfaces for basic document structure and querying. This chapter focuses on the features and uses of DOM Level 1 as it relates to HTML pages in the browser and its implementation in JavaScript. The most recent versions of Internet Explorer, Firefox, Safari, Chrome, and Opera all have excellent DOM implementations.



Note that all DOM objects are represented by COM objects in Internet Explorer 8 and earlier. This means that the objects don't behave or function the same way as native JavaScript objects. These differences are highlighted throughout the chapter.

HIERARCHY OF NODES

Any HTML or XML document can be represented as a hierarchy of nodes using the DOM. There are several node types, each representing different information and/or markup in the document. Each node type has different characteristics, data, and methods, and each may have relationships with other nodes. These relationships create a hierarchy that allows markup to be represented as a tree, rooted at a particular node. For instance, consider the following HTML:

```
<html>
  <head>
    <title>Sample Page</title>
  </head>
  <body>
    <p>Hello World!</p>
  </body>
</html>
```

This simple HTML document can be represented in a hierarchy, as illustrated in Figure 10-1.

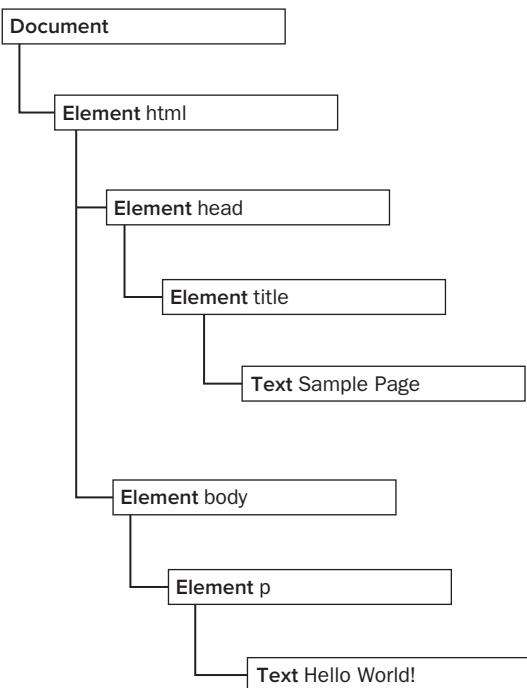


FIGURE 10-1

A document node represents every document as the root. In this example, the only child of the document node is the `<html>` element, which is called the *document element*. The document element is the outermost element in the document within which all other elements exist. There can be only one document element per document. In HTML pages, the document element is always the `<html>` element. In XML, where there are no predefined elements, any element may be the document element.

Every piece of markup can be represented by a node in the tree: HTML elements are represented by element nodes, attributes are represented by attribute nodes, the document type is represented by a document type node, and comments are represented by comment nodes. In total, there are 12 node types, all of which inherit from a base type.

The Node Type

DOM Level 1 describes an interface called `Node` that is to be implemented by all node types in the DOM. The `Node` interface is implemented in JavaScript as the `Node` type, which is accessible in all browsers except Internet Explorer. All node types inherit from `Node` in JavaScript, so all node types share the same basic properties and methods.

Every node has a `nodeType` property that indicates the type of node that it is. Node types are represented by one of the following 12 numeric constants on the `Node` type:

- `Node.ELEMENT_NODE` (1)
- `Node.ATTRIBUTE_NODE` (2)

- Node.TEXT_NODE (3)
- Node.CDATA_SECTION_NODE (4)
- Node.ENTITY_REFERENCE_NODE (5)
- Node.ENTITY_NODE (6)
- Node.PROCESSING_INSTRUCTION_NODE (7)
- Node.COMMENT_NODE (8)
- Node.DOCUMENT_NODE (9)
- Node.DOCUMENT_TYPE_NODE (10)
- Node.DOCUMENT_FRAGMENT_NODE (11)
- Node.NOTATION_NODE (12)

A node's type is easy to determine by comparing against one of these constants, as shown here:

```
if (someNode.nodeType == Node.ELEMENT_NODE){    //won't work in IE < 9
    alert("Node is an element.");
}
```

This example compares the `someNode.nodeType` to the `Node.ELEMENT_NODE` constant. If they're equal, it means `someNode` is actually an element. Unfortunately, since Internet Explorer 8 and earlier doesn't expose the `Node` type constructor, this code will cause an error. For cross-browser compatibility, it's best to compare the `nodeType` property against a numeric value, as in the following:

```
if (someNode.nodeType == 1){    //works in all browsers
    alert("Node is an element.");
}
```

Not all node types are supported in web browsers. Developers most often work with element and text nodes. The support level and usage of each node type is discussed later in the chapter.

The `nodeName` and `nodeValue` Properties

Two properties, `nodeName` and `nodeValue`, give specific information about the node. The values of these properties are completely dependent on the node type. It's always best to test the node type before using one of these values, as the following code shows:

```
if (someNode.nodeType == 1){
    value = someNode.nodeName;    //will be the element's tag name
}
```

In this example, the node type is checked to see if the node is an element. If so, the `nodeName` value is assigned to a variable. For elements, `nodeName` is always equal to the element's tag name, and `nodeValue` is always `null`.

Node Relationships

All nodes in a document have relationships to other nodes. These relationships are described in terms of traditional family relationships as if the document tree were a family tree. In HTML, the `<body>` element is considered a child of the `<html>` element; likewise the `<html>` element is considered the parent of the `<body>` element. The `<head>` element is considered a sibling of the `<body>` element, because they both share the same immediate parent, the `<html>` element.

Each node has a `childNodes` property containing a `NodeList`. A `NodeList` is an array-like object used to store an ordered list of nodes that are accessible by position. Keep in mind that a `NodeList` is not an instance of `Array` even though its values can be accessed using bracket notation and the `length` property is present. `NodeList` objects are unique in that they are actually queries being run against the DOM structure, so changes will be reflected in `NodeList` objects automatically. It is often said that a `NodeList` is a living, breathing object rather than a snapshot of what happened at the time it was first accessed.

The following example shows how nodes stored in a `NodeList` may be accessed via bracket notation or by using the `item()` method:

```
var firstChild = someNode.childNodes[0];
var secondChild = someNode.childNodes.item(1);
var count = someNode.childNodes.length;
```

Note that using bracket notation and using the `item()` method are both acceptable practices, although most developers use bracket notation because of its similarity to arrays. Also note that the `length` property indicates the number of nodes in the `NodeList` at that time. It's possible to convert `NodeList` objects into arrays using `Array.prototype.slice()` as was discussed earlier for the `arguments` object. Consider the following example:

```
//won't work in IE8 and earlier
var arrayOfNodes = Array.prototype.slice.call(someNode.childNodes, 0);
```

This works in all browsers except Internet Explorer 8 and earlier versions, which throw an error because a `NodeList` is implemented as a COM object and thus cannot be used where a `JScript` object is necessary. To convert a `NodeList` to an array in Internet Explorer, you must manually iterate over the members. The following function works in all browsers:

```
function convertToArray(nodes) {
    var array = null;
    try {
        array = Array.prototype.slice.call(nodes, 0); //non-IE and IE9+
    } catch (ex) {
        array = new Array();
        for (var i=0, len=nodes.length; i < len; i++) {
            array.push(nodes[i]);
        }
    }
    return array;
}
```

The `convertToArray()` function first attempts to use the easiest manner of creating an array. If that throws an error (which it will in Internet Explorer through version 8), the error is caught by the `try-catch` block and the array is created manually. This is another form of quirks detection.

Each node has a `parentNode` property pointing to its parent in the document tree. All nodes contained within a `childNodes` list have the same parent, so each of their `parentNode` properties points to the same node. Additionally, each node within a `childNodes` list is considered to be a sibling of the other nodes in the same list. It's possible to navigate from one node in the list to another by using the `previousSibling` and `nextSibling` properties. The first node in the list has `null` for the value of its `previousSibling` property, and the last node in the list has `null` for the value of its `nextSibling` property, as shown in the following example:

```
if (someNode.nextSibling === null){
    alert("Last node in the parent's childNodes list.");
} else if (someNode.previousSibling === null){
    alert("First node in the parent's childNodes list.");
}
```

Note that if there's only one child node, both `nextSibling` and `previousSibling` will be `null`.

Another relationship exists between a parent node and its first and last child nodes. The `firstChild` and `lastChild` properties point to the first and last node in the `childNodes` list, respectively. The value of `someNode.firstChild` is always equal to `someNode.childNodes[0]`, and the value of `someNode.lastChild` is always equal to `someNode.childNodes[someNode.childNodes.length-1]`. If there is only one child node, `firstChild` and `lastChild` point to the same node; if there are no children, then `firstChild` and `lastChild` are both `null`. All of these relationships help to navigate easily between nodes in a document structure. Figure 10-2 illustrates these relationships.

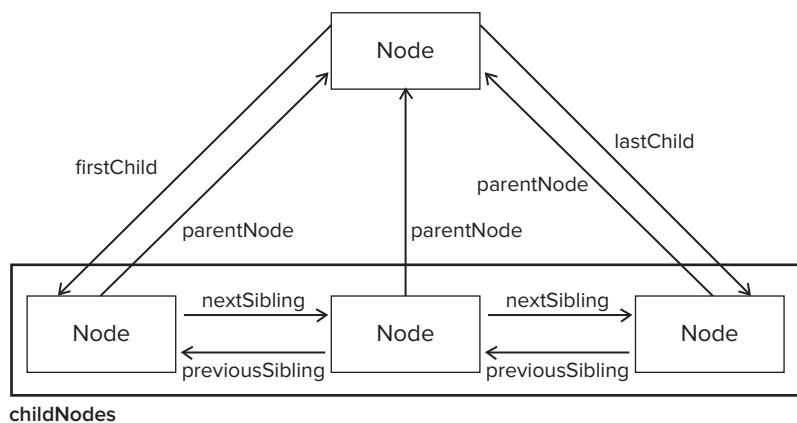


FIGURE 10-2

With all of these relationships, the `childNodes` property is really more of a convenience than a necessity, since it's possible to reach any node in a document tree by simply using the relationship pointers. Another convenience method is `hasChildNodes()`, which returns `true` if the node has one or more child nodes and is more efficient than querying the `length` of the `childNodes` list.

One final relationship is shared by every node. The `ownerDocument` property is a pointer to the document node that represents the entire document. Nodes are considered to be owned by the document in which they were created (typically the same in which they reside), because nodes cannot exist simultaneously in two or more documents. This property provides a quick way to access the document node without needing to traverse the node hierarchy back up to the top.



Not all node types can have child nodes even though all node types inherit from Node. The differences among node types are discussed later in this chapter.

Manipulating Nodes

Because all relationship pointers are read-only, several methods are available to manipulate nodes. The most often-used method is `appendChild()`, which adds a node to the end of the `childNodes` list. Doing so updates all of the relationship pointers in the newly added node, the parent node, and the previous last child in the `childNodes` list. When complete, `appendChild()` returns the newly added node. Here is an example:

```
var returnedNode = someNode.appendChild(newNode);
alert(returnedNode == newNode);           //true
alert(someNode.lastChild == newNode);     //true
```

If the node passed into `appendChild()` is already part of the document, it is removed from its previous location and placed at the new location. Even though the DOM tree is connected by a series of pointers, no DOM node may exist in more than one location in a document. So if you call `appendChild()` and pass in the first child of a parent, as the following example shows, it will end up as the last child:

```
//assume multiple children for someNode
var returnedNode = someNode.appendChild(someNode.firstChild);
alert(returnedNode == someNode.firstChild); //false
alert(returnedNode == someNode.lastChild); //true
```

When a node needs to be placed in a specific location within the `childNodes` list, instead of just at the end, the `insertBefore()` method may be used. The `insertBefore()` method accepts two arguments: the node to insert and a reference node. The node to insert becomes the previous sibling of the reference node and is ultimately returned by the method. If the reference node is `null`, then `insertBefore()` acts the same as `appendChild()`, as this example shows:

```
//insert as last child
returnedNode = someNode.insertBefore(newNode, null);
alert(newNode == someNode.lastChild); //true

//insert as the new first child
returnedNode = someNode.insertBefore(newNode, someNode.firstChild);
alert(returnedNode == newNode); //true
alert(newNode == someNode.firstChild); //true

//insert before last child
returnedNode = someNode.insertBefore(newNode, someNode.lastChild);
alert(newNode == someNode.childNodes[someNode.childNodes.length-2]); //true
```

Both `appendChild()` and `insertBefore()` insert nodes without removing any. The `replaceChild()` method accepts two arguments: the node to insert and the node to replace. The node to replace is returned by the function and is removed from the document tree completely while the inserted node takes its place. Here is an example:

```
//replace first child  
var returnedNode = someNode.replaceChild(newNode, someNode.firstChild);  
  
//replace last child  
returnedNode = someNode.replaceChild(newNode, someNode.lastChild);
```

When a node is inserted using `replaceChild()`, all of its relationship pointers are duplicated from the node it is replacing. Even though the replaced node is technically still owned by the same document, it no longer has a specific location in the document.

To remove a node without replacing it, you can use the `removeChild()` method. This method accepts a single argument, which is the node to remove. The removed node is then returned as the function value, as this example shows:

```
//remove first child  
var formerFirstChild = someNode.removeChild(someNode.firstChild);  
  
//remove last child  
var formerLastChild = someNode.removeChild(someNode.lastChild);
```

As with `replaceChild()`, a node removed via `removeChild()` is still owned by the document but doesn't have a specific location in the document.

All four of these methods work on the immediate children of a specific node, meaning that to use them you must know the immediate parent node (which is accessible via the previously mentioned `parentNode` property). Not all node types can have child nodes, and these methods will throw errors if you attempt to use them on nodes that don't support children.

Other Methods

Two other methods are shared by all node types. The first is `cloneNode()`, which creates an exact clone of the node on which it's called. The `cloneNode()` method accepts a single Boolean argument indicating whether to do a deep copy. When the argument is `true`, a deep copy is used, cloning the node and its entire subtree; when `false`, only the initial node is cloned. The cloned node that is returned is owned by the document but has no parent node assigned. As such, the cloned node is an orphan and doesn't exist in the document until added via `appendChild()`, `insertBefore()`, or `replaceChild()`. For example, consider the following HTML:

```
<ul>  
  <li>item 1</li>  
  <li>item 2</li>  
  <li>item 3</li>  
</ul>
```

If a reference to this `` element is stored in a variable named `myList`, the following code shows the two modes of the `cloneNode()` method:

```
var deepList = myList.cloneNode(true);
alert(deepList.childNodes.length);      //3 (IE < 9) or 7 (others)

var shallowList = myList.cloneNode(false);
alert(shallowList.childNodes.length);   //0
```

In this example, `deepList` is filled with a deep copy of `myList`. This means `deepList` has three list items, each of which contains text. The variable `shallowList` contains a shallow copy of `myList`, so it has no child nodes. The difference in `deepList.childNodes.length` is due to the different ways that white space is handled in Internet Explorer 8 and earlier as compared to other browsers. Internet Explorer prior to version 9 did not create nodes for white space.



The `cloneNode()` method doesn't copy JavaScript properties that you add to DOM nodes, such as event handlers. This method copies only attributes and, optionally, child nodes. Everything else is lost. Internet Explorer has a bug where event handlers are also cloned, so removing event handlers before cloning is recommended.

The last remaining method is `normalize()`. Its sole job is to deal with text nodes in a document subtree. Because of parser implementations or DOM manipulations, it's possible to end up with text nodes that contain no text or text nodes that are siblings. When `normalize()` is called on a node, that node's descendants are searched for both of these circumstances. If an empty text node is found, it is removed; if text nodes are immediate siblings, they are joined into a single text node. This method is discussed further later on in this chapter.

The Document Type

JavaScript represents document nodes via the `Document` type. In browsers, the `document` object is an instance of `HTMLDocument` (which inherits from `Document`) and represents the entire HTML page. The `document` object is a property of `window` and so is accessible globally. A `Document` node has the following characteristics:

- `nodeType` is 9.
- `nodeName` is "`#document`".
- `nodeValue` is `null`.
- `parentNode` is `null`.
- `ownerDocument` is `null`.
- Child nodes may be a `DocumentType` (maximum of one), `Element` (maximum of one), `ProcessingInstruction`, or `Comment`.

The Document type can represent HTML pages or other XML-based documents, though the most common use is through an instance of `HTMLDocument` through the `document` object. The `document` object can be used to get information about the page and to manipulate both its appearance and the underlying structure.



The Document type constructor and prototype are accessible in script for Firefox, Safari, Chrome, and Opera. Internet Explorer through version 9 still does not expose Document. The HTMLDocument type constructor and prototype are accessible in all browsers, including Internet Explorer beginning with version 8.

Document Children

Though the DOM specification states that the children of a `Document` node can be a `DocumentType`, `Element`, `ProcessingInstruction`, or `Comment`, there are two built-in shortcuts to child nodes. The first is the `documentElement` property, which always points to the `<html>` element in an HTML page. The `document` element is always represented in the `childNodes` list as well, but the `documentElement` property gives faster and more direct access to that element. Consider the following simple page:

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

When this page is parsed by a browser, the document has only one child node, which is the `<html>` element. This element is accessible from both `documentElement` and the `childNodes` list, as shown here:

```
var html = document.documentElement;      //get reference to <html>
alert(html === document.childNodes[0]);  //true
alert(html === document.firstChild);     //true
```

This example shows that the values of `documentElement`, `firstChild`, and `childNodes[0]` are all the same — all three point to the `<html>` element.

As an instance of `HTMLDocument`, the `document` object also has a `body` property that points to the `<body>` element directly. Since this is the element most often used by developers, `document.body` tends to be used quite frequently in JavaScript, as this example shows:

```
var body = document.body;    //get reference to <body>
```

Both `document.documentElement` and `document.body` are supported in all major browsers.

Another possible child node of a `Document` is a `DocumentType`. The `<!DOCTYPE>` tag is considered to be a separate entity from other parts of the document, and its information is accessible through the `doctype` property (`document.doctype` in browsers), as shown here:

```
var doctype = document.doctype;  //get reference to <!DOCTYPE>
```

Browser support for `document.doctype` varies considerably, as described here:

- **Internet Explorer 8 and earlier** — A document type, if present, is misinterpreted as a comment and treated as a `Comment` node. `document.doctype` is always `null`.
- **Internet Explorer 9+ and Firefox** — A document type, if present, is the first child node of the document. `document.doctype` is a `DocumentType` node, and the same node is accessible via `document.firstChild` or `document.childNodes[0]`.
- **Safari, Chrome, and Opera** — A document type, if present, is parsed but is not considered a child node of the document. `document.doctype` is a `DocumentType` node, but the node does not appear in `document.childNodes`.

Because of the inconsistent browser support for `document.doctype`, it is of limited usefulness.

Comments that appear outside of the `<html>` element are, technically, child nodes of the document. Once again, browser support varies greatly as to whether these comments will be recognized and represented appropriately. Consider the following HTML page:

```
<!-- first comment -->
<html>
  <body>

    </body>
</html>
<!-- second comment -->
```

This page seems to have three child nodes: a comment, the `<html>` element, and another comment. Logically, you would expect `document.childNodes` to have three items corresponding to what appears in the code. In practice, however, browsers handle comments outside of the `<html>` element in the following very different ways:

- Internet Explorer 8 and earlier, Safari 3.1 and later, Opera, and Chrome create a comment node for the first comment but not for the second. The first comment becomes the first node in `document.childNodes`.
- Internet Explorer 9 and later create a comment node for the first comment as part of `document.childNodes`. They also create a comment node for the second comment as part of `document.body.childNodes`.
- Firefox and Safari prior to version 3.1 ignore both comments.

Once again, the inconsistent behavior makes accessing comments outside the `<html>` element essentially useless.

For the most part, the `appendChild()`, `removeChild()`, and `replaceChild()` methods aren't used on `document`, since the document type (if present) is read-only and there can be only one element child node (which is already present).

Document Information

The `document` object, as an instance of `HTMLDocument`, has several additional properties that standard `Document` objects do not have. These properties provide information about the web page

that is loaded. The first such property is `title`, which contains the text in the `<title>` element and is displayed in the title bar or tab of the browser window. This property can be used to retrieve the current page title and to change the page title such that the changes are reflected in the browser title bar. Changing the value of the `title` property does not change the `<title>` element at all. Here is an example:

```
//get the document title  
var originalTitle = document.title;  
  
//set the document title  
document.title = "New page title";
```

The next three properties are all related to the request for the web page: `URL`, `domain`, and `referrer`. The `URL` property contains the complete URL of the page (the URL in the address bar), the `domain` property contains just the domain name of the page, and the `referrer` property gives the URL of the page that linked to this page. The `referrer` property may be an empty string if there is no referrer to the page. All of this information is available in the HTTP header of the request and is simply made available in JavaScript via these properties, as shown in the following example:

```
//get the complete URL  
var url = document.URL;  
  
//get the domain  
var domain = document.domain;  
  
//get the referrer  
var referrer = document.referrer;
```

The `URL` and `domain` properties are related. For example, if `document.URL` is `http://www.wrox.com/WileyCDA/`, then `document.domain` will be `www.wrox.com`.

Of these three properties, the `domain` property is the only one that can be set. There are some restrictions as to what the value of `domain` can be set to because of security issues. If the `URL` contains a subdomain, such as `p2p.wrox.com`, the `domain` may be set only to “`wrox.com`” (the same is true when the `URL` contains “`www`,” such as `www.wrox.com`). The `property` can never be set to a domain that the `URL` doesn’t contain, as this example demonstrates:

```
//page from p2p.wrox.com  
  
document.domain = "wrox.com";           //succeeds  
  
document.domain = "nczonline.net";      //error!
```

The ability to set `document.domain` is useful when there is a frame or iframe on the page from a different subdomain. Pages from different subdomains can’t communicate with one another via JavaScript because of cross-domain security restrictions. By setting `document.domain` in each page to the same value, the pages can access JavaScript objects from each other. For example, if a page is loaded from `www.wrox.com` and it has an iframe with a page loaded from `p2p.wrox.com`, each page’s `document.domain` string will be different, and the outer page and the inner page are

restricted from accessing each other's JavaScript objects. If the `document.domain` value in each page is set to "wrox.com", the pages can then communicate.

A further restriction in the browser disallows tightening of the domain property once it has been loosened. This means you cannot set `document.domain` to "wrox.com" and then try to set it back to "p2p.wrox.com", because the latter would cause an error, as shown here:

```
//page from p2p.wrox.com

document.domain = "wrox.com";           //loosen - succeeds

document.domain = "p2p.wrox.com";        //tighten - error!
```

This restriction exists in all browsers but was implemented in Internet Explorer beginning with version 8.

Locating Elements

Perhaps the most common DOM activity is to retrieve references to a specific element or sets of elements to perform certain operations. This capability is provided via a number of methods on the `document` object. The `Document` type provides two methods to this end: `getElementById()` and `getElementsByName()`.

The `getElementById()` method accepts a single argument — the ID of an element to retrieve — and returns the element if found, or `null` if an element with that ID doesn't exist. The ID must be an exact match, including character case, to the `id` attribute of an element on the page. Consider the following element:

```
<div id="myDiv">Some text</div>
```

This element can be retrieved using the following code:

```
var div = document.getElementById("myDiv");           //retrieve reference to the <div>
```

The following code, however, would return `null` in all browsers except Internet Explorer 7 and earlier:

```
var div = document.getElementById("mydiv");           //won't work (except in IE <= 7)
```

Internet Explorer prior to version 8 considered IDs to be case-insensitive, so "myDiv" and "mydiv" are considered to be the same element ID.

If there is more than one element with the same ID in a page, `getElementById()` returns the element that appears first in the document. Internet Explorer 7 and earlier add an interesting quirk to this, also returning form elements (`<input>`, `<textarea>`, `<button>`, and `<select>`) that have a `name` attribute matching the given ID. If one of these form elements has a `name` attribute equal to the specified ID, and it appears before an element with the given ID in the document, Internet Explorer returns the form element. Here's an example:

```
<input type="text" name="myElement" value="Text field">
<div id="myElement">A div</div>
```

Using this HTML, a call to `document.getElementById()` in Internet Explorer 7 returns a reference to the `<input>` element, whereas the same call returns a reference to the `<div>` element in all other browsers. To avoid this issue in Internet Explorer, you should ensure that form fields don't have `name` attributes that are equivalent to other element IDs.

The `getElementsByName()` method is another commonly used method for retrieving element references. It accepts a single argument — the tag name of the elements to retrieve — and returns a `NodeList` containing zero or more elements. In HTML documents, this method returns an `HTMLCollection` object, which is very similar to a `NodeList` in that it is considered a “live” collection. For example, the following code retrieves all `` elements in the page and returns an `HTMLCollection`:

```
var images = document.getElementsByName("img");
```

This code stores an `HTMLCollection` object in the `images` variable. As with `NodeList` objects, items in `HTMLCollection` objects can be accessed using bracket notation or the `item()` method. The number of elements in the object can be retrieved via the `length` property, as this example demonstrates:

```
alert(images.length);           //output the number of images
alert(images[0].src);          //output the src attribute of the first image
alert(images.item(0).src);      //output the src attribute of the first image
```

The `HTMLCollection` object has an additional method, `namedItem()`, that lets you reference an item in the collection via its `name` attribute. For example, suppose you had the following `` element in a page:

```

```

A reference to this `` element can be retrieved from the `images` variable like this:

```
var myImage = images.namedItem("myImage");
```

In this way, an `HTMLCollection` gives you access to named items in addition to indexed items, making it easier to get exactly the elements you want. You can also access named items by using bracket notation, as shown in the following example:

```
var myImage = images["myImage"];
```

For `HTMLCollection` objects, bracket notation can be used with either numeric or string indices. Behind the scenes, a numeric index calls `item()` and a string index calls `namedItem()`.

To retrieve all elements in the document, pass in “`*`” to `getElementsByName()`. The asterisk is generally understood to mean “all” in JavaScript and Cascading Style Sheets (CSS). Here’s an example:

```
var allElements = document.getElementsByName("*");
```

This single line of code returns an `HTMLCollection` containing all of the elements in the order in which they appear. So the first item is the `<html>` element, the second is the `<head>` element, and

so on. The Internet Explorer 8 and earlier implementation of comments actually makes them into elements, so the browser will return comment nodes when `getElementsByTagName("*)` is called. Internet Explorer 9 does not treat comments as elements and so does not return them.



Even though the specification states that tag names are case-sensitive, the `getElementsByTagName()` method is case-insensitive for maximum compatibility with existing HTML pages. When used in XML pages, including XHTML, `getElementsByTagName()` switches to case-sensitive mode.

A third method, which is defined on the `HTMLDocument` type only, is `getElementsByName()`. As its name suggests, this method returns all elements that have a given `name` attribute. The `getElementsByName()` method is most often used with radio buttons, all of which must have the same name to ensure the correct value gets sent to the server, as the following example shows:

```
<fieldset>
  <legend>Which color do you prefer?</legend>
  <ul>
    <li><input type="radio" value="red" name="color" id="colorRed">
      <label for="colorRed">Red</label></li>
    <li><input type="radio" value="green" name="color" id="colorGreen">
      <label for="colorGreen">Green</label></li>
    <li><input type="radio" value="blue" name="color" id="colorBlue">
      <label for="colorBlue">Blue</label></li>
  </ul>
</fieldset>
```

In this code, the radio buttons all have a `name` attribute of "color" even though their IDs are different. The IDs allow the `<label>` elements to be applied to the radio buttons, and the `name` attribute ensures that only one of the three values will be sent to the server. These radio buttons can all then be retrieved using the following line of code:

```
var radios = document.getElementsByName("color");
```

As with `getElementsByTagName()`, the `getElementsByName()` method returns an `HTMLCollection`. In this context, however, the `namedItem()` method always retrieves the first item (since all items have the same name).

Special Collections

The `document` object has several special collections. Each of these collections is an `HTMLCollection` object and provides faster access to common parts of the document, as described here:

- `document.anchors` — Contains all `<a>` elements with a `name` attribute in the document.
- `document.applets` — Contains all `<applet>` elements in the document. This collection is deprecated, because the `<applet>` element is no longer recommended for use.
- `document.forms` — Contains all `<form>` elements in the document. The same as `document.getElementsByTagName("form")`.

- `document.images` — Contains all `` elements in the document. The same as `document.getElementsByTagName("img")`.
- `document.links` — Contains all `<a>` elements with an `href` attribute in the document.

These special collections are always available on `HTMLDocument` objects and, like all `HTMLCollection` objects, are constantly updated to match the contents of the current document.

DOM Conformance Detection

Because there are multiple levels and multiple parts of the DOM, it became necessary to determine exactly what parts of the DOM a browser has implemented. The `document.implementation` property is an object containing information and functionality tied directly to the browser's implementation of the DOM. DOM Level 1 specifies only one method on `document.implementation`, which is `hasFeature()`. The `hasFeature()` method accepts two arguments: the name and version of the DOM feature to check for. If the browser supports the named feature and version, this method returns `true`, as with this example:

```
var hasXmlDom = document.implementation.hasFeature("XML", "1.0");
```

The various values that can be tested are listed in the following table.

FEATURE	SUPPORTED VERSIONS	DESCRIPTION
Core	1.0, 2.0, 3.0	Basic DOM that spells out the use of a hierarchical tree to represent documents
XML	1.0, 2.0, 3.0	XML extension of the Core that adds support for CDATA sections, processing instructions, and entities
HTML	1.0, 2.0	HTML extension of XML that adds support for HTML-specific elements and entities
Views	2.0	Accomplishes formatting of a document based on certain styles
StyleSheets	2.0	Relates style sheets to documents
CSS	2.0	Support for Cascading Style Sheets Level 1
CSS2	2.0	Support for Cascading Style Sheets Level 2
Events	2.0, 3.0	Generic DOM events
UIEvents	2.0, 3.0	User interface events
MouseEvents	2.0, 3.0	Events caused by the mouse (click, mouseover, and so on)
MutationEvents	2.0, 3.0	Events fired when the DOM tree is changed
HTMLEvents	2.0	HTML 4.01 events

continues

(continued)

FEATURE	SUPPORTED VERSIONS	DESCRIPTION
Range	2.0	Objects and methods for manipulating a range in a DOM tree
Traversal	2.0	Methods for traversing a DOM tree
LS	3.0	Loading and saving between files and DOM trees synchronously
LS-Async	3.0	Loading and saving between files and DOM trees asynchronously
Validation	3.0	Methods to modify a DOM tree and still make it valid

Although it is a nice convenience, the drawback of using `hasFeature()` is that the implementer gets to decide if the implementation is indeed conformant with the various parts of the DOM specification. It's very easy to make this method return `true` for any and all values, but that doesn't necessarily mean that the implementation conforms to all the specifications it claims to. Safari 2.x and earlier, for example, return `true` for some features that aren't fully implemented. In most cases, it's a good idea to use capability detection in addition to `hasFeature()` before using specific parts of the DOM.

Document Writing

One of the older capabilities of the `document` object is the ability to write to the output stream of a web page. This capability comes in the form of four methods: `write()`, `writeln()`, `open()`, and `close()`. The `write()` and `writeln()` methods each accept a string argument to write to the output stream. `write()` simply adds the text as is, whereas `writeln()` appends a new-line character (`\n`) to the end of the string. These two methods can be used as a page is being loaded to dynamically add content to the page, as shown in the following example:



```
<html>
<head>
    <title>document.write() Example</title>
</head>
<body>
    <p>The current date and time is:
    <script type="text/javascript">
        document.write("<strong>" + (new Date()).toString() + "</strong>");
    </script>
    </p>
</body>
</html>
```

[DocumentWriteExample01.htm](#)

This example outputs the current date and time as the page is being loaded. The date is enclosed by a `` element, which is treated the same as if it were included in the HTML portion of the

page, meaning that a DOM element is created and can later be accessed. Any HTML that is output via `write()` or `writeln()` is treated this way.

The `write()` and `writeln()` methods are often used to dynamically include external resources such as JavaScript files. When including JavaScript files, you must be sure not to include the string "`</script>`" directly, as the following example demonstrates, because it will be interpreted as the end of a script block and the rest of the code won't execute.



Available for
download on
Wrox.com

```
<html>
<head>
    <title>document.write() Example</title>
</head>
<body>
    <script type="text/javascript">
        document.write("<script type=\"text/javascript\" src=\"file.js\">" +
                      "</script>");
    </script>
</body>
</html>
```

[DocumentWriteExample02.htm](#)

Even though this file looks correct, the closing "`</script>`" string is interpreted as matching the outermost `<script>` tag, meaning that the text ") ; will appear on the page. To avoid this, you simply need to change the string, as mentioned in Chapter 2 and shown here:

```
<html>
<head>
    <title>document.write() Example</title>
</head>
<body>
    <script type="text/javascript">
        document.write("<script type=\"text/javascript\" src=\"file.js\">" +
                      "<\\>"); // Note the extra backslash before the closing tag
    </script>
</body>
</html>
```

[DocumentWriteExample03.htm](#)

The string "`<\\>`" no longer registers as a closing tag for the outermost `<script>` tag, so there is no extra content output to the page.

The previous examples use `document.write()` to output content directly into the page as it's being rendered. If `document.write()` is called after the page has been completely loaded, the content overwrites the entire page, as shown in the following example:

```
<html>
<head>
    <title>document.write() Example</title>
</head>
```

```

<body>
    <p>This is some content that you won't get to see because it will be
    overwritten.</p>
    <script type="text/javascript">
        window.onload = function(){
            document.write("Hello world!");
        };
    </script>
</body>
</html>

```

DocumentWriteExample04.htm

In this example, the `window.onload` event handler is used to delay the execution of the function until the page is completely loaded (events are discussed in Chapter 13). When that happens, the string "Hello world!" overwrites the entire page content.

The `open()` and `close()` methods are used to open and close the web page output stream, respectively. Neither method is required to be used when `write()` or `writeln()` is used during the course of page loading.



Document writing is not supported in strict XHTML documents. For pages that are served with the application/xhtml+xml content type, these methods will not work.

The Element Type

Next to the `Document` type, the `Element` type is most often used in web programming. The `Element` type represents an XML or HTML element, providing access to information such as its tag name, children, and attributes. An `Element` node has the following characteristics:

- `nodeType` is 1.
- `nodeName` is the element's tag name.
- `nodeValue` is null.
- `parentNode` may be a `Document` or `Element`.
- Child nodes may be `Element`, `Text`, `Comment`, `ProcessingInstruction`, `CDataSection`, or `EntityReference`.

An element's tag name is accessed via the `nodeName` property or by using the `tagName` property; both properties return the same value (the latter is typically used for clarity). Consider the following element:

```
<div id="myDiv"></div>
```

This element can be retrieved and its tag name accessed in the following way:

```
var div = document.getElementById("myDiv");
alert(div.tagName);      // "DIV"
alert(div.tagName == div.nodeName); // true
```

The element in question has a tag name of `div` and an ID of "myDiv". Note, however, that `div.tagName` actually outputs "DIV" instead of "div". When used with HTML, the tag name is always represented in all uppercase; when used with XML (including XHTML), the tag name always matches the case of the source code. If you aren't sure whether your script will be on an HTML or XML document, it's best to convert tag names to a common case before comparison, as this example shows:

```
if (element.tagName == "div"){ //AVOID! Error prone!
    //do something here
}

if (element.tagName.toLowerCase() == "div"){ //Preferred - works in all documents
    //do something here
}
```

This example shows two comparisons against a `tagName` property. The first is quite error prone because it won't work in HTML documents. The second approach, converting the tag name to all lowercase, is preferred because it will work for both HTML and XML documents.



The Element type constructor and prototype are accessible in script in all modern browsers, including Internet Explorer as of version 8. Older browsers, such as Safari prior to version 2 and Opera prior to version 8, do not expose the Element type constructor.

HTML Elements

All HTML elements are represented by the `HTMLElement` type, either directly or through subtyping. The `HTMLElement` inherits directly from `Element` and adds several properties. Each property represents one of the following standard attributes that are available on every HTML element:

- `id` — A unique identifier for the element in the document.
- `title` — Additional information about the element, typically represented as a tooltip.
- `lang` — The language code for the contents of the element (rarely used).
- `dir` — The direction of the language, "ltr" (left-to-right) or "rtl" (right-to-left); also rarely used.
- `className` — The equivalent of the `class` attribute, which is used to specify CSS classes on an element. The property could not be named `class` because `class` is an ECMAScript reserved word (see Chapter 1 for information about reserved words).



Available for
download on
Wrox.com

```
<div id="myDiv" class="bd" title="Body text" lang="en" dir="ltr"></div>
```

[HTMLElementsExample01.htm](#)

All of the information specified by this element may be retrieved using the following JavaScript code:

```
var div = document.getElementById("myDiv");
alert(div.id);           // "myDiv"
alert(div.className);    // "bd"
alert(div.title);        // "Body text"
alert(div.lang);         // "en"
alert(div.dir);          // "ltr"
```

It's also possible to use the following code to change each of the attributes by assigning new values to the properties:

```
div.id = "someOtherId";
div.className = "ft";
div.title = "Some other text";
div.lang = "fr";
div.dir = "rtl";
```

[HTMLElementsExample01.htm](#)

Not all of the properties effect changes on the page when overwritten. Changes to `id` or `lang` will be transparent to the user (assuming no CSS styles are based on these values), whereas changes to `title` will be apparent only when the mouse is moved over the element. Changes to `dir` will cause the text on the page to be aligned to either the left or the right as soon as the property is written. Changes to `className` may appear immediately if the class has different CSS style information than the previous one.

As mentioned previously, all HTML elements are represented by an instance of `HTMLElement` or a more specific subtype. The following table lists each HTML element and its associated type (italicized elements are deprecated). Note that these types are accessible in Opera, Safari, Chrome, and Firefox via JavaScript but not in Internet Explorer prior to version 8.

ELEMENT	TYPE	ELEMENT	TYPE
A	HTMLAnchorElement	INPUT	HTMLInputElement
ABBR	HTMLElement	INS	HTMLModElement
ACRONYM	HTMLElement	<i>ISINDEX</i>	<i>HTMLISIndexElement</i>
ADDRESS	HTMLElement	KBD	HTMLElement

ELEMENT	TYPE	ELEMENT	TYPE
APPLET	<i>HTMLAppletElement</i>	LABEL	HTMLLabelElement
AREA	HTMLAreaElement	LEGEND	HTMLLegendElement
B	HTMLElement	LI	HTMLListElement
BASE	HTMLBaseElement	LINK	HTMLLinkElement
BASEFONT	<i>HTMLBaseFontElement</i>	MAP	HTMLMapElement
BDO	HTMLElement	MENU	HTMLMenuElement
BIG	HTMLElement	META	HTMLMetaElement
BLOCKQUOTE	HTMLQuoteElement	NOFRAMES	HTMLElement
BODY	HTMLBodyElement	NOSCRIPT	HTMLElement
BR	HTMLBRElement	OBJECT	HTMLObjectElement
BUTTON	HTMLButtonElement	OL	HTMLListElement
CAPTION	HTMLTableCaptionElement	OPTGROUP	HTMLOptGroupElement
CENTER	<i>HTMLElement</i>	OPTION	HTMLOptionElement
CITE	HTMLElement	P	HTMLParagraphElement
CODE	HTMLElement	PARAM	HTMLParamElement
COL	HTMLTableColElement	PRE	HTMLPreElement
COLGROUP	HTMLTableColElement	Q	HTMLQuoteElement
DD	HTMLElement	S	<i>HTMLElement</i>
DEL	HTMLModElement	SAMP	HTMLElement
DFN	HTMLElement	SCRIPT	HTMLScriptElement
DIR	<i>HTMLDirectoryElement</i>	SELECT	HTMLSelectElement
DIV	HTMLDivElement	SMALL	HTMLElement
DL	HTMLDListElement	SPAN	HTMLElement
DT	HTMLElement	STRIKE	<i>HTMLElement</i>
EM	HTMLElement	STRONG	HTMLElement
FIELDSET	HTMLFieldSetElement	STYLE	HTMLStyleElement
FONT	<i>HTMLFontElement</i>	SUB	HTMLElement

continues

(continued)

ELEMENT	TYPE	ELEMENT	TYPE
FORM	HTMLFormElement	SUP	HTMLElement
FRAME	HTMLFrameElement	TABLE	HTMLTableElement
FRAMESET	HTMLFrameSetElement	TBODY	HTMLTableSectionElement
H1	HTMLHeadingElement	TD	HTMLTableCellElement
H2	HTMLHeadingElement	TEXTAREA	HTMLTextAreaElement
H3	HTMLHeadingElement	TFOOT	HTMLTableSectionElement
H4	HTMLHeadingElement	TH	HTMLTableCellElement
H5	HTMLHeadingElement	THEAD	HTMLTableSectionElement
H6	HTMLHeadingElement	TITLE	HTMLTitleElement
HEAD	HTMLHeadElement	TR	HTMLTableRowElement
HR	HTMLHRElement	TT	HTMLElement
HTML	HTMLHtmlElement	U	HTMLElement
I	HTMLElement	UL	HTMLULListElement
IFRAME	HTMLIFrameElement	VAR	HTMLElement
IMG	HTMLImageElement		

Each of these types has attributes and methods associated with it. Many of these types are discussed throughout this book.

Getting Attributes

Each element may have zero or more attributes, which are typically used to give extra information about the particular element or its contents. The three primary DOM methods for working with attributes are `getAttribute()`, `setAttribute()`, and `removeAttribute()`. These methods are intended to work on any attribute, including those defined as properties on the `HTMLElement` type. Here's an example:

```
var div = document.getElementById("myDiv");
alert(div.getAttribute("id"));           // "myDiv"
alert(div.getAttribute("class"));        // "bd"
alert(div.getAttribute("title"));        // "Body text"
alert(div.getAttribute("lang"));         // "en"
alert(div.getAttribute("dir"));          // "ltr"
```

Note that the attribute name passed into `getAttribute()` is exactly the same as the actual attribute name, so you pass in "class" to get the value of the `class` attribute (not `className`, which is

necessary when the attribute is accessed as an object property). If the attribute with the given name doesn't exist, `getAttribute()` always returns `null`.

The `getAttribute()` method can also retrieve the value of custom attributes that aren't part of the formal HTML language. Consider the following element:

```
<div id="myDiv" my_special_attribute="hello!"></div>
```

In this element, a custom attribute named `my_special_attribute` is defined to have a value of "`hello!`". This value can be retrieved using `getAttribute()` just like any other attribute, as shown here:

```
var value = div.getAttribute("my_special_attribute");
```

Note that attribute names are case-insensitive, so "`ID`" and "`id`" are considered the same attribute. Also note that, according to HTML5, custom attributes should be prepended with `data-` in order to validate.

All attributes on an element are also accessible as properties of the DOM element object itself. There are, of course, the five properties defined on `HTMLElement` that map directly to corresponding attributes, but all recognized (noncustom) attributes get added to the object as properties. Consider the following element:

```
<div id="myDiv" align="left" my_special_attribute="hello"></div>
```

Since `id` and `align` are recognized attributes for the `<div>` element in HTML, they will be represented by properties on the element object. The `my_special_attribute` attribute is custom and so won't show up as a property on the element in Safari, Opera, Chrome, or Firefox. Internet Explorer through version 8 creates properties for custom attributes as well, as this example demonstrates:



Available for
download on
[Wrox.com](#)

```
alert(div.id);           // "myDiv"  
alert(div.my_special_attribute); // undefined (except in IE)  
alert(div.align);        // "left"
```

[ElementAttributesExample02.htm](#)

Two types of attributes have property names that don't map directly to the same value returned by `getAttribute()`. The first attribute is `style`, which is used to specify stylistic information about the element using CSS. When accessed via `getAttribute()`, the `style` attribute contains CSS text while accessing it via a property that returns an object. The `style` property is used to programmatically access the styling of the element (discussed in Chapter 12) and so does not map directly to the `style` attribute.

The second category of attribute that behaves differently is event-handler attributes such as `onclick`. When used on an element, the `onclick` attribute contains JavaScript code, and that code string is returned when using `getAttribute()`. When the `onclick` property is accessed, however, it returns a JavaScript function (or `null` if the attribute isn't specified). This is because `onclick` and other event-handling properties are provided such that functions can be assigned to them.

Because of these differences, developers tend to forego `getAttribute()` when programming the DOM in JavaScript and instead use the object properties exclusively. The `getAttribute()` method is used primarily to retrieve the value of a custom attribute.



In Internet Explorer versions 7 and earlier, the `getAttribute()` method for the `style` attribute and event-handling attributes such as `onclick` always return the same value as if they were accessed via a property. So, `getAttribute("style")` returns an object and `getAttribute("onclick")` returns a function. Though fixed in Internet Explorer 8, this inconsistency is another reason to avoid using `getAttribute()` for HTML attributes.

Setting Attributes

The sibling method to `getAttribute()` is `setAttribute()`, which accepts two arguments: the name of the attribute to set and the value to set it to. If the attribute already exists, `setAttribute()` replaces its value with the one specified; if the attribute doesn't exist, `setAttribute()` creates it and sets its value. Here is an example:



Available for download on Wrox.com

```
div.setAttribute("id", "someOtherId");
div.setAttribute("class", "ft");
div.setAttribute("title", "Some other text");
div.setAttribute("lang", "fr");
div.setAttribute("dir", "rtl");
```

[ElementAttributesExample01.htm](#)

The `setAttribute()` method works with both HTML attributes and custom attributes in the same way. Attribute names get normalized to lowercase when set using this method, so "`ID`" ends up as "`id`".

Because all attributes are properties, assigning directly to the property can set the attribute values, as shown here:

```
div.id = "someOtherId";
div.align = "left";
```

Note that adding a custom property to a DOM element, as the following example shows, does not automatically make it an attribute of the element:

```
div.mycolor = "red";
alert(div.getAttribute("mycolor")); //null (except in Internet Explorer)
```

This example adds a custom property named `mycolor` and sets its value to "red". In most browsers, this property does not automatically become an attribute on the element, so calling `getAttribute()` to retrieve an attribute with the same name returns `null`. In Internet Explorer, however, custom properties are considered to be attributes of the element and vice versa.



Internet Explorer versions 7 and earlier had some abnormal behavior regarding `setAttribute()`. Attempting to set the `class` or `style` attributes has no effect, similar to setting an event-handler property using `setAttribute()`. Even though these issues were resolved in Internet Explorer 8, it's always best to set these attributes using properties.

The last method is `removeAttribute()`, which removes the attribute from the element altogether. This does more than just clear the attribute's value; it completely removes the attribute from the element, as shown here:

```
div.removeAttribute("class");
```

This method isn't used very frequently, but it can be useful for specifying exactly which attributes to include when serializing a DOM element.



Internet Explorer versions 6 and earlier don't support `removeAttribute()`.

The attributes Property

The `Element` type is the only DOM node type that uses the `attributes` property. The `attributes` property contains a `NamedNodeMap`, which is a “live” collection similar to a `NodeList`. Every attribute on an element is represented by an `Attr` node, each of which is stored in the `NamedNodeMap` object. A `NamedNodeMap` object has the following methods:

- `getNamedItem(name)` — Returns the node whose `nodeName` property is equal to `name`.
- `removeNamedItem(name)` — Removes the node whose `nodeName` property is equal to `name` from the list.
- `setNamedItem(node)` — Adds the node to the list, indexing it by its `nodeName` property.
- `item(pos)` — Returns the node in the numerical position `pos`.

Each node in the `attributes` property is a node whose `nodeName` is the attribute name and whose `nodeValue` is the attribute's value. To retrieve the `id` attribute of an element, you can use the following code:

```
var id = element.attributes.getNamedItem("id").nodeValue;
```

Following is a shorthand notation for accessing attributes by name using bracket notation:

```
var id = element.attributes["id"].nodeValue;
```

It's possible to use this notation to set attribute values as well, retrieving the attribute node and then setting the `nodeValue` to a new value, as this example shows:

```
element.attributes["id"].nodeValue = "someOtherId";
```

The `removeNamedItem()` method functions the same as the `removeAttribute()` method on the element — it simply removes the attribute with the given name. The following example shows how the sole difference is that `removeNamedItem()` returns the `Attr` node that represented the attribute:

```
var oldAttr = element.attributes.removeNamedItem("id");
```

The `setNamedItem()` is a rarely used method that allows you to add a new attribute to the element by passing in an attribute node, as shown in this example:

```
element.attributes.setNamedItem(newAttr);
```

Generally speaking, because of their simplicity, the `getAttribute()`, `removeAttribute()`, and `setAttribute()` methods are preferred to using any of the preceding attributes methods.

The one area where the `attributes` property is useful is to iterate over the attributes on an element. This is done most often when serializing a DOM structure into an XML or HTML string. The following code iterates over each attribute on an element and constructs a string in the format `name="value"`:



```
function outputAttributes(element) {
    var pairs = new Array(),
        attrName,
        attrValue,
        i,
        len;

    for (i=0, len=element.attributes.length; i < len; i++) {
        attrName = element.attributes[i].nodeName;
        attrValue = element.attributes[i].nodeValue;
        pairs.push(attrName + "=\"" + attrValue + "\"");
    }
    return pairs.join(" ");
}
```

ElementAttributesExample03.htm

This function uses an array to store the name-value pairs until the end, concatenating them with a space in between. (This technique is frequently used when serializing into long strings.) Using the `attributes.length` property, the `for` loop iterates over each attribute, outputting the name and value into a string. Here are a couple of important things to note about the way this code works:

- Browsers differ on the order in which they return attributes in the `attributes` object. The order in which the attributes appear in the HTML or XML code may not necessarily be the order in which they appear in the `attributes` object.
- Internet Explorer 7 and earlier versions return all possible attributes on an HTML element, even if they aren't specified. This means often returning more than 100 attributes.

The previous function can be augmented to ensure that only specified attributes are included to provide for the issue with Internet Explorer versions 7 and earlier. Each attribute node has a property called `specified` that is set to `true` when the attribute is specified either as an HTML attribute or via the `setAttribute()` method. For Internet Explorer, this value is `false` for the extra attributes,

whereas the extra attributes aren't present in other browsers (thus, `specified` is always `true` for any attribute node). The code can then be augmented as follows:



```
function outputAttributes(element) {
    var pairs = new Array(),
        attrName,
        attrValue,
        i,
        len;

    for (i=0, len=element.attributes.length; i < len; i++) {
        attrName = element.attributes[i].nodeName;
        attrValue = element.attributes[i].nodeValue;
        if (element.attributes[i].specified) {
            pairs.push(attrName + "=" + attrValue);
        }
    }
    return pairs.join(" ");
}
```

ElementAttributesExample04.htm

This revised function ensures that only specified attributes are returned for Internet Explorer 7 and earlier.

Creating Elements

New elements can be created by using the `document.createElement()` method. This method accepts a single argument, which is the tag name of the element to create. In HTML documents, the tag name is case-insensitive, whereas it is case-sensitive in XML documents (including XHTML). To create a `<div>` element, the following code can be used:

```
var div = document.createElement("div");
```

Using the `createElement()` method creates a new element and sets its `ownerDocument` property. At this point, you can manipulate the element's attributes, add more children to it, and so on. Consider the following example:

```
div.id = "myNewDiv";
div.className = "box";
```

Setting these attributes on the new element assigns information only. Since the element is not part of the document tree, it doesn't affect the browser's display. The element can be added to the document tree using `appendChild()`, `insertBefore()`, or `replaceChild()`. The following code adds the newly created element to the document's `<body>` element:

```
document.body.appendChild(div);
```

CreateElementExample01.htm

Once the element has been added to the document tree, the browser renders it immediately. Any changes to the element after this point are immediately reflected by the browser.

Internet Explorer allows an alternate use of `createElement()`, allowing you to specify a full element, including attributes, as this example shows:

```
var div = document.createElement("<div id=\"myNewDiv\" class=\"box\"></div>");
```

This usage is helpful to work around some issues regarding dynamically created elements in Internet Explorer 7 and earlier. The known issues are as follows:

- Dynamically created `<iFrame>` elements can't have their `name` attribute set.
- Dynamically created `<input>` elements won't get reset via the form's `reset()` method (discussed in Chapter 14).
- Dynamically created `<button>` elements with a `type` attribute of "reset" won't reset the form.
- Dynamically created radio buttons with the same name have no relation to one another. Radio buttons with the same name are supposed to be different values for the same option, but dynamically created ones lose this relationship.

Each of these issues can be addressed by specifying the complete HTML for the tag in `createElement()`, as follows:

```
if (client.browser.ie && client.browser.ie <= 7){  
  
    //create iFrame with a name  
    var iframe = document.createElement("<iFrame name=\"myframe\"></iFrame>");  
  
    //create input element  
    var input = document.createElement("<input type=\"checkbox\">");  
  
    //create button  
    var button = document.createElement("<button type=\"reset\"></button>");  
  
    //create radio buttons  
    var radio1 = document.createElement("<input type=\"radio\" name=\"choice\" " +  
        " value=\"1\">");  
    var radio2 = document.createElement("<input type=\"radio\" name=\"choice\" " +  
        " value=\"2\">");  
  
}
```

Just as with the traditional way of using `createElement()`, using it in this way returns a DOM element reference that can be added into the document and otherwise augmented. This usage is recommended only when dealing with one of these specific issues in Internet Explorer 7 and earlier, because it requires browser detection. Note that no other browser supports this usage.

Element Children

Elements may have any number of children and descendants since elements may be children of elements. The `childNodes` property contains all of the immediate children of the element, which may be other elements, text nodes, comments, or processing instructions. There is a significant

difference between browsers regarding the identification of these nodes. For example, consider the following code:

```
<ul id="myList">
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ul>
```

When this code is parsed in Internet Explorer 8 and earlier, the `` element has three child nodes, one for each of the `` elements. In all other browsers, the `` element has seven elements: three `` elements and four `text` nodes representing the white space between `` elements. If the white space between elements is removed, as the following example demonstrates, all browsers return the same number of child nodes:

```
<ul id="myList"><li>Item 1</li><li>Item 2</li><li>Item 3</li></ul>
```

Using this code, all browsers return three child nodes for the `` element. It's important to keep these browser differences in mind when navigating children using the `childNodes` property. Oftentimes, it's necessary to check the `nodeType` before performing an action, as the following example shows:

```
for (var i=0, len=element.childNodes.length; i < len; i++){
    if (element.childNodes[i].nodeType == 1){
        //do processing
    }
}
```

This code loops through each child node of a particular element and performs an operation only if `nodeType` is equal to 1 (the `element` node type identified).

To get child nodes and other descendants with a particular tag name, elements also support the `getElementsByName()` method. When used on an element, this method works exactly the same as the document version except that the search is rooted on the element, so only descendants of that element are returned. In the `` code earlier in this section, all `` elements can be retrieved using the following code:

```
var ul = document.getElementById("myList");
var items = ul.getElementsByName("li");
```

Keep in mind that this works because the `` element has only one level of descendants. If there were more levels, all `` elements contained in all levels would be returned.

The Text Type

Text nodes are represented by the `Text` type and contain plain text that is interpreted literally and may contain escaped HTML characters but no HTML code. A `Text` node has the following characteristics:

- `nodeType` is 3.
- `nodeName` is "#text".

- `nodeValue` is text contained in the node.
- `parentNode` is an `Element`.
- Child nodes are not supported.

The text contained in a `Text` node may be accessed via either the `nodeValue` property or the `data` property, both of which contain the same value. Changes to either `nodeValue` or `data` are reflected in the other as well. The following methods allow for manipulation of the text in the node:

- `appendData(text)` — Appends *text* to the end of the node.
- `deleteData(offset, count)` — Deletes *count* number of characters starting at position *offset*.
- `insertData(offset, text)` — Inserts *text* at position *offset*.
- `replaceData(offset, count, text)` — Replaces the text starting at *offset* through *offset + count* with *text*.
- `splitText(offset)` — Splits the text node into two `Text` nodes separated at position *offset*.
- `substringData(offset, count)` — Extracts a string from the text beginning at position *offset* and continuing until *offset + count*.

In addition to these methods, the `length` property returns the number of characters in the node. This value is the same as using `nodeValue.length` or `data.length`.

By default, every element that may contain content will have at most one text node when content is present. Here is an example:

```
<!-- no content, so no text node -->
<div></div>

<!-- white space content, so one text node -->
<div> </div>

<!-- content, so one text node -->
<div>Hello World!</div>
```

The first `<div>` element in this code has no content, so there is no text node. Any content between the opening and closing tags means that a text node must be created, so the second `<div>` element has a single text node as a child even though its content is white space. The text node's `nodeValue` is a single space. The third `<div>` also has a single text node whose `nodeValue` is "Hello World!". The following code lets you access this node:

```
var textNode = div.firstChild; //or div.childNodes[0]
```

Once a reference to the text node is retrieved, it can be changed like this:

```
div.firstChild.nodeValue = "Some other message";
```



Available for
download on
Wrox.com

[TextNodeExample01.htm](#)

As long as the node is currently in the document tree, the changes to the text node will be reflected immediately. Another note about changing the value of a text node is that the string is HTML- or XML-encoded (depending on the type of document), meaning that any less-than symbols, greater-than symbols, or quotation marks are escaped, as shown in this example:



Available for download on
Wrox.com

```
//outputs as "Some &lt;strong&ampgtother&lt;/strong&ampgt message"
div.firstChild.nodeValue = "Some <strong>other</strong> message";
```

[TextNodeExample02.htm](#)

This is an effective way of HTML-encoding a string before inserting it into the DOM document.



The Text type constructor and prototype are accessible in script in Internet Explorer 8, Firefox, Safari, Chrome, and Opera.

Creating Text Nodes

New text nodes can be created using the `document.createTextNode()` method, which accepts a single argument — the text to be inserted into the node. As with setting the value of an existing text node, the text will be HTML- or XML-encoded, as shown in this example:

```
var textNode = document.createTextNode("<strong>Hello</strong> world!");
```

When a new text node is created, its `ownerDocument` property is set, but it does not appear in the browser window until it is added to a node in the document tree. The following code creates a new `<div>` element and adds a message to it:

```
var element = document.createElement("div");
element.className = "message";

var textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

document.body.appendChild(element);
```

[TextNodeExample03.htm](#)

This example creates a new `<div>` element and assigns it a `class` of "message". Then a text node is created and added to that element. The last step is to add the element to the document's body, which makes both the element and the text node appear in the browser.



Available for
download on
Wrox.com

```
var element = document.createElement("div");
element.className = "message";

var textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

var anotherTextNode = document.createTextNode("Yippee!");
element.appendChild(anotherTextNode);

document.body.appendChild(element);
```

TextNodeExample04.htm

When a text node is added as a sibling of another text node, the text in those nodes is displayed without any space between them.

Normalizing Text Nodes

Sibling text nodes can be confusing in DOM documents since there is no simple text string that can't be represented in a single text node. Still, it is not uncommon to come across sibling text nodes in DOM documents, so there is a method to join sibling text nodes together. This method is called `normalize()`, and it exists on the `Node` type (and thus is available on all node types). When `normalize()` is called on a parent of two or more text nodes, those nodes are merged into one text node whose `nodeValue` is equal to the concatenation of the `nodeValue` properties of each text node. Here's an example:

```
var element = document.createElement("div");
element.className = "message";

var textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

var anotherTextNode = document.createTextNode("Yippee!");
element.appendChild(anotherTextNode);

document.body.appendChild(element);

alert(element.childNodes.length); //2

element.normalize();
alert(element.childNodes.length); //1
alert(element.firstChild.nodeValue); // "Hello world!Yippee!"
```

TextNodeExample05.htm

When the browser parses a document, it will never create sibling text nodes. Sibling text nodes can appear only by programmatic DOM manipulation.



The normalize() method causes Internet Explorer 6 to crash in certain circumstances. Though unconfirmed, this may have been fixed in later patches to Internet Explorer 6. This problem doesn't occur in Internet Explorer 7 or later.

Splitting Text Nodes

The Text type has a method that does the opposite of normalize(): the splitText() method splits a text node into two text nodes, separating thenodeValue at a given offset. The original text node contains the text up to the specified offset, and the new text node contains the rest of the text. The method returns the new text node, which has the same parentNode as the original. Consider the following example:



Available for download on
Wrox.com

```
var element = document.createElement("div");
element.className = "message";

var textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

document.body.appendChild(element);

var newNode = element.firstChild.splitText(5);
alert(element.firstChild.nodeValue); // "Hello"
alert(newNode.nodeValue); // " world!"
alert(element.childNodes.length); // 2
```

[TextNodeExample06.htm](#)

In this example, the text node containing the text "Hello world!" is split into two text nodes at position 5. Position 5 contains the space between "Hello" and "world!", so the original text node has the string "Hello" and the new one has the text " world!" (including the space).

Splitting text nodes is used most often with DOM parsing techniques for extracting data from text nodes.

The Comment Type

Comments are represented in the DOM by the Comment type. A Comment node has the following characteristics:

- nodeType is 8.
- nodeName is "#comment".
- nodeValue is the content of the comment.
- parentNode is a Document or Element.
- Child nodes are not supported.

The `Comment` type inherits from the same base as the `Text` type, so it has all of the same string-manipulation methods except `splitText()`. Also similar to the `Text` type, the actual content of the comment may be retrieved using either `nodeValue` or the `data` property.

A comment node can be accessed as a child node from its parent. Consider the following HTML code:



Available for
download on
[Wrox.com](#)

```
<div id="myDiv"><!-- A comment --></div>
```

In this case, the comment is a child node of the `<div>` element, which means it can be accessed like this:

```
var div = document.getElementById("myDiv");
var comment = div.firstChild;
alert(comment.data);      // "A comment"
```

CommentNodeExample01.htm

Comment nodes can also be created using the `document.createComment()` method and passing in the comment text, as shown in the following code:

```
var comment = document.createComment("A comment");
```

Not surprisingly, comment nodes are rarely accessed or created, because they serve very little purpose algorithmically. Additionally, browsers don't recognize comments that exist after the closing `</html>` tag. If you need to access comment nodes, make sure they appear as descendants of the `<html>` element.



The `Comment` type constructor and prototype are accessible in Firefox, Safari, Chrome, and Opera. The Internet Explorer 8 comment nodes are considered to be elements with a tag name of "`!`". This means comment nodes can be returned by `getElementsByName()`. Internet Explorer 9 represents comments via a custom constructor called `HTMLCommentElement` even though it doesn't treat comments as elements.

The CDATASection Type

CDATA sections are specific to XML-based documents and are represented by the `CDATASection` type. Similar to `Comment`, the `CDATASection` type inherits from the base `Text` type, so it has all of the same string manipulation methods except for `splitText()`. A `CDATASection` node has the following characteristics:

- `nodeType` is 4.
- `nodeName` is `"#cdata-section"`.

- `nodeValue` is the contents of the CDATA section.
- `parentNode` is a Document or Element.
- Child nodes are not supported.

CDATA sections are valid only in XML documents, so most browsers will incorrectly parse a CDATA section into either a Comment or an Element. Consider the following:

```
<div id="myDiv"><! [CDATA[This is some content.]]></div>
```

In this example, a `CDataSection` node should exist as the first child of the `<div>`; however, none of the four major browsers interpret it as such. Even in valid XHTML pages, the browsers don't properly support embedded CDATA sections.

True XML documents allow the creation of CDATA sections using `document.createCDataSection()` and pass in the node's content.



The CDataSection type constructor and prototype are accessible in Firefox, Safari, Chrome, and Opera. Internet Explorer through version 9 still does not support this type.

The DocumentType Type

The `DocumentType` type is not used very often in web browsers and is supported in only Firefox, Safari, and Opera. A `DocumentType` object contains all of the information about the document's doctype and has the following characteristics:

- `nodeType` is 10.
- `nodeName` is the name of the doctype.
- `nodeValue` is null.
- `parentNode` is a Document.
- Child nodes are not supported.

`DocumentType` objects cannot be created dynamically in DOM Level 1; they are created only as the document's code is being parsed. For browsers that support it, the `DocumentType` object is stored in `document.doctype`. DOM Level 1 describes three properties for `DocumentType` objects: `name`, which is the name of the doctype; `entities`, which is a `NamedNodeMap` of entities described by the doctype; and `notations`, which is a `NamedNodeMap` of notations described by the doctype. Because documents in browsers typically use an HTML or XHTML doctype, the `entities` and `notations` lists are typically empty. (They are filled only with inline doctypes.) For all intents and purposes, the `name` property is the only useful one available. This property is filled with the name of the doctype, which is the text that appears immediately after `<!DOCTYPE`. Consider the following HTML 4.01 strict doctype:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

For this doctype, the `name` property is "HTML":

```
alert(document.doctype.name); // "HTML"
```

Internet Explorer 8 and earlier did not support the `DocumentType` type, so `document.doctype` is always `null`. Furthermore, these browsers misinterpret the doctype as a comment and actually create a comment node for it. Internet Explorer 9 properly assigns an object to `document.doctype` but still does not expose the `DocumentType` type.

The DocumentFragment Type

Of all the node types, the `DocumentFragment` type is the only one that has no representation in markup. The DOM defines a document fragment as a “lightweight” document, capable of containing and manipulating nodes without all of the additional overhead of a complete document. `DocumentFragment` nodes have the following characteristics:

- `nodeType` is 11.
- `nodeName` is "#document-fragment".
- `nodeValue` is `null`.
- `parentNode` is `null`.
- Child nodes may be `Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection`, or `EntityReference`.

A document fragment cannot be added to a document directly. Instead, it acts as a repository for other nodes that may need to be added to the document. Document fragments are created using the `document.createDocumentFragment()` method, shown here:

```
var fragment = document.createDocumentFragment();
```

Document fragments inherit all methods from `Node` and are typically used to perform DOM manipulations that are to be applied to a document. If a node from the document is added to a document fragment, that node is removed from the document tree and won't be rendered by the browser. New nodes that are added to a document fragment are also not part of the document tree. The contents of a document fragment can be added to a document via `appendChild()` or `insertBefore()`. When a document fragment is passed in as an argument to either of these methods, all of the document fragment's child nodes are added in that spot; the document fragment itself is never added to the document tree. For example, consider the following HTML:

```
<ul id="myList"></ul>
```

Suppose you would like to add three list items to this `` element. Adding each item directly to the element causes the browser to rerender the page with the new information. To avoid this, the following code example uses a document fragment to create the list items and then add them all at the same time:

```
var fragment = document.createDocumentFragment();
var ul = document.getElementById("myList");
var li = null;

for (var i=0; i < 3; i++) {
```



```

        li = document.createElement("li");
        li.appendChild(document.createTextNode("Item " + (i+1)));
        fragment.appendChild(li);
    }

ul.appendChild(fragment);

```

DocumentFragmentExample01.htm

This example begins by creating a document fragment and retrieving a reference to the `` element. The `for` loop creates three list items, each with text indicating which item they are. To do this, an `` element is created and then a text node is created and added to that element. The `` element is then added to the document fragment using `appendChild()`. When the loop is complete, all of the items are added to the `` element by calling `appendChild()` and passing in the document fragment. At that point, the document fragment's child nodes are all removed and placed onto the `` element.

The Attr Type

Element attributes are represented by the `Attr` type in the DOM. The `Attr` type constructor and prototype are accessible in all browsers, including Internet Explorer beginning with version 8. Technically, attributes are nodes that exist in an element's `attributes` property. Attribute nodes have the following characteristics:

- `nodeType` is 11.
- `nodeName` is the attribute name.
- `nodeValue` is the attribute value.
- `parentNode` is `null`.
- Child nodes are not supported in HTML.
- Child nodes may be `Text` or `EntityReference` in XML.

Even though they are nodes, attributes are not considered part of the DOM document tree. Attribute nodes are rarely referenced directly, with most developers favoring the use of `getAttribute()`, `setAttribute()`, and `removeAttribute()`.

There are three properties on an `Attr` object: `name`, which is the attribute name (same as `nodeName`); `value`, which is the attribute value (same as `nodeValue`); and `specified`, which is a Boolean value indicating if the attribute was specified in code or if it is a default value.

New attribute nodes can be created by using `document.createAttribute()` and passing in the name of the attribute. For example, to add an `align` attribute to an element, the following code can be used:



Available for
download on
Wrox.com

```

var attr = document.createAttribute("align");
attr.value = "left";
element.setAttributeNode(attr);

alert(element.attributes["align"].value);           //"left"

```

```
alert(element.getAttributeNode("align").value); // "left"
alert(element.getAttribute("align"));           // "left"
```

AttrExample01.htm

In this example, a new attribute node is created. The name property is assigned by the call to `createAttribute()`, so there is no need to assign it directly afterward. The value property is then assigned to "left". To add the newly created attribute to an element, you can use the element's `setAttributeNode()` method. Once the attribute is added, it can be accessed in any number of ways: via the `attributes` property, using `getAttributeNode()`, or using `getAttribute()`. Both `attributes` and `getAttributeNode()` return the actual `Attr` node for the attribute, whereas `getAttribute()` returns only the attribute value.



There is really not a good reason to access attribute nodes directly. The `get Attribute()`, `setAttribute()`, and `removeAttribute()` methods are preferable over manipulating attribute nodes.

WORKING WITH THE DOM

In many cases, working with the DOM is fairly straightforward, making it easy to re-create with JavaScript what normally would be created using HTML code. There are, however, times when using the DOM is not as simple as it may appear. Browsers are filled with hidden gotchas and incompatibilities that make coding certain parts of the DOM more complicated than coding its other parts.

Dynamic Scripts

The `<script>` element is used to insert JavaScript code into the page, either using by the `src` attribute to include an external file or by including text inside the element itself. Dynamic scripts are those that don't exist when the page is loaded but are included later by using the DOM. As with the `HTML` element, there are two ways to do this: pulling in an external file or inserting text directly.

Dynamically loading an external JavaScript file works as you would expect. Consider the following `<script>` element:

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

This `<script>` element includes the text for the Chapter 9 client-detection script. The DOM code to create this node is as follows:

```
var script = document.createElement("script");
script.type = "text/javascript";
script.src = "client.js";
document.body.appendChild(script);
```

As you can see, the DOM code exactly mirrors the HTML code that it represents. Note that the external file is not downloaded until the `<script>` element is added to the page on the last line. The element could be added to the `<head>` element as well, though this has the same effect. This process can be generalized into the following function:

```
function loadScript(url){
    var script = document.createElement("script");
    script.type = "text/javascript";
    script.src = url;
    document.body.appendChild(script);
}
```

This function can now be used to load external JavaScript files via the following call:

```
loadScript("client.js");
```

Once loaded, the script is fully available to the rest of the page. This leaves only one problem: how do you know when the script has been fully loaded? Unfortunately, there is no standard way to handle this. Some events are available depending on the browser being used, as discussed in Chapter 13.

The other way to specify JavaScript code is inline, as in this example:

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

Using the DOM, it would be logical for the following to work:

```
var script = document.createElement("script");
script.type = "text/javascript";
script.appendChild(document.createTextNode("function sayHi(){alert('hi')}"));
document.body.appendChild(script);
```

This works in Firefox, Safari, Chrome, and Opera. In Internet Explorer, however, this causes an error. Internet Explorer treats `<script>` elements as special and won't allow regular DOM access to child nodes. A property called `text` exists on all `<script>` elements that can be used specifically to assign JavaScript code to, as in the following example:



Available for
download on
Wrox.com

```
var script = document.createElement("script");
script.type = "text/javascript";
script.text = "function sayHi(){alert('hi')}";
document.body.appendChild(script);
```

DynamicScriptExample01.htm

This updated code works in Internet Explorer, Firefox, Opera, and Safari 3 and later. Safari versions prior to 3 don't support the `text` property correctly; however, it will allow the assignment

of code using the text-node technique. If you need to do this in an earlier Safari version, the following code can be used:

```
var script = document.createElement("script");
script.type = "text/javascript";
var code = "function sayHi(){alert('hi');}";
try {
    script.appendChild(document.createTextNode("code"));
} catch (ex){
    script.text = "code";
}
document.body.appendChild(script);
```

Here, the standard DOM text-node method is attempted first, because it works in everything but Internet Explorer, which will throw an error. If that line causes an error, that means it is Internet Explorer, and the `text` property must be used. This can be generalized into the following function:



```
function loadScriptString(code){
    var script = document.createElement("script");
    script.type = "text/javascript";
    try {
        script.appendChild(document.createTextNode(code));
    } catch (ex){
        script.text = code;
    }
    document.body.appendChild(script);
}
```

The function is called as follows:

```
loadScriptString("function sayHi(){alert('hi');}");
```

DynamicScriptExample02.htm

Code loaded in this manner is executed in the global scope and is available immediately after the script finishes executing. This is essentially the same as passing the string into `eval()` in the global scope.

Dynamic Styles

CSS styles are included in HTML pages using one of two elements. The `<link>` element is used to include CSS from an external file, whereas the `<style>` element is used to specify inline styles. Similar to dynamic scripts, dynamic styles don't exist on the page when it is loaded initially; rather, they are added after the page has been loaded.

Consider this typical `<link>` element:

```
<link rel="stylesheet" type="text/css" href="styles.css">
```

This element can just as easily be created using the following DOM code:

```
var link = document.createElement("link");
link.rel = "stylesheet";
link.type = "text/css";
link.href = "styles.css";
var head = document.getElementsByTagName("head")[0];
head.appendChild(link);
```

This code works in all major browsers without any issue. Note that <link> elements should be added to the <head> instead of the body for this to work properly in all browsers. The technique can be generalized into the following function:

```
function loadStyles(url){
    var link = document.createElement("link");
    link.rel = "stylesheet";
    link.type = "text/css";
    link.href = url;
    var head = document.getElementsByTagName("head")[0];
    head.appendChild(link);
}
```

The `loadStyles()` function can then be called like this:

```
loadStyles("styles.css");
```

Loading styles via an external file is asynchronous, so the styles will load out of order with the JavaScript code being executed. Typically it's not necessary to know when the styles have been fully loaded; however, there are some techniques (events, discussed in Chapter 13) that can be used to accomplish this.

The other way to define styles is using the <style> element and including inline CSS, such as this:

```
<style type="text/css">
body {
    background-color: red;
}
</style>
```

Logically, the following DOM code should work:



```
Available for  
download on  
Wrox.com
```

```
var style = document.createElement("style");
style.type = "text/css";
style.appendChild(document.createTextNode("body{background-color:red}"));
var head = document.getElementsByTagName("head")[0];
head.appendChild(style);
```

[DynamicStyleExample01.htm](#)

This code works in Firefox, Safari, Chrome, and Opera but not in Internet Explorer. Internet Explorer treats <style> nodes as special, similar to <script> nodes, and so won't allow access to its child nodes. In fact, Internet Explorer throws the same error as when you try to add a child node

to a `<script>` element. The workaround for Internet Explorer is to access the element's `styleSheet` property, which in turn has a property called `cssText` that may be set to CSS code (both of these properties are discussed further in Chapter 12), as this code sample shows:

```
var style = document.createElement("style");
style.type = "text/css";
try{
    style.appendChild(document.createTextNode("body{background-color:red}"));
} catch (ex){
    style.styleSheet.cssText = "body{background-color:red}";
}
var head = document.getElementsByTagName("head")[0];
head.appendChild(style);
```

Similar to the code for adding inline scripts dynamically, this new code uses a `try-catch` statement to catch the error that Internet Explorer throws and then responds by using the Internet Explorer-specific way of setting styles. The generic solution is as follows:



```
function loadStyleString(css){
    var style = document.createElement("style");
    style.type = "text/css";
    try{
        style.appendChild(document.createTextNode(css));
    } catch (ex){
        style.styleSheet.cssText = css;
    }
    var head = document.getElementsByTagName("head")[0];
    head.appendChild(style);
}
```

DynamicStyleExample02.htm

The function can be called as follows:

```
loadStyleString("body{background-color:red}");
```

Styles specified in this way are added to the page instantly, so changes should be seen immediately.



If you're coding for Internet Explorer specifically, be careful using `styleSheet.cssText`. If you reuse the same `<style>` element and try to set this property more than once, it has a tendency to crash the browser. Also, setting `cssText` to the empty string has the potential to crash the browser as well. This is a bug in the browser that hopefully will be fixed in the future.

Manipulating Tables

One of the most complex structures in HTML is the `<table>` element. Creating new tables typically means numerous tags for table rows, table cells, table headers, and so forth. Because of this

complexity, using the core DOM methods to create and change tables can require a large amount of code. Suppose you want to create the following HTML table using the DOM:

```
<table border="1" width="100%">
  <tbody>
    <tr>
      <td>Cell 1,1</td>
      <td>Cell 2,1</td>
    </tr>
    <tr>
      <td>Cell 1,2</td>
      <td>Cell 2,2</td>
    </tr>
  </tbody>
</table>
```

To accomplish this with the core DOM methods, the code would look something like this:

```
//create the table
var table = document.createElement("table");
table.border = 1;
table.width = "100%";

//create the tbody
var tbody = document.createElement("tbody");
table.appendChild(tbody);

//create the first row
var row1 = document.createElement("tr");
tbody.appendChild(row1);
var cell1_1 = document.createElement("td");
cell1_1.appendChild(document.createTextNode("Cell 1,1"));
row1.appendChild(cell1_1);
var cell2_1 = document.createElement("td");
cell2_1.appendChild(document.createTextNode("Cell 2,1"));
row1.appendChild(cell2_1);

//create the second row
var row2 = document.createElement("tr");
tbody.appendChild(row2);
var cell1_2 = document.createElement("td");
cell1_2.appendChild(document.createTextNode("Cell 1,2"));
row2.appendChild(cell1_2);
var cell2_2= document.createElement("td");
cell2_2.appendChild(document.createTextNode("Cell 2,2"));
row2.appendChild(cell2_2);

//add the table to the document body
document.body.appendChild(table);
```

This code is quite verbose and a little hard to follow. To facilitate building tables, the HTML DOM adds several properties and methods to the `<table>`, `<tbody>`, and `<tr>` elements.

The `<table>` element adds the following:

- `caption` — Pointer to the `<caption>` element (if it exists).
- `tBodies` — An `HTMLCollection` of `<tbody>` elements.
- `tFoot` — Pointer to the `<tfoot>` element (if it exists).
- `tHead` — Pointer to the `<thead>` element (if it exists).
- `rows` — An `HTMLCollection` of all rows in the table.
- `createTHead()` — Creates a `<thead>` element, places it into the table, and returns a reference.
- `createTFoot()` — Creates a `<tfoot>` element, places it into the table, and returns a reference.
- `createCaption()` — Creates a `<caption>` element, places it into the table, and returns a reference.
- `deleteTHead()` — Deletes the `<thead>` element.
- `deleteTFoot()` — Deletes the `<tfoot>` element.
- `deleteCaption()` — Deletes the `<caption>` element.
- `deleteRow(pos)` — Deletes the row in the given position.
- `insertRow(pos)` — Inserts a row in the given position in the `rows` collection.

The `<tbody>` element adds the following:

- `rows` — An `HTMLCollection` of rows in the `<tbody>` element.
- `deleteRow(pos)` — Deletes the row in the given position.
- `insertRow(pos)` — Inserts a row in the given position in the `rows` collection and returns a reference to the new row.

The `<tr>` element adds the following:

- `cells` — An `HTMLCollection` of cells in the `<tr>` element.
- `deleteCell(pos)` — Deletes the cell in the given position.
- `insertCell(pos)` — Inserts a cell in the given position in the `cells` collection and returns a reference to the new cell.

These properties and methods can greatly reduce the amount of code necessary to create a table. For example, the previous code can be rewritten using these methods as follows (the highlighted code is updated):

```
//create the table
var table = document.createElement("table");
table.border = 1;
table.width = "100%";

//create the tbody
var tbody = document.createElement("tbody");
```

```

table.appendChild(tbody);

//create the first row
tbody.insertRow(0);
tbody.rows[0].insertCell(0);
tbody.rows[0].cells[0].appendChild(document.createTextNode("Cell 1,1"));
tbody.rows[0].insertCell(1);
tbody.rows[0].cells[1].appendChild(document.createTextNode("Cell 2,1"));

//create the second row
tbody.insertRow(1);
tbody.rows[1].insertCell(0);
tbody.rows[1].cells[0].appendChild(document.createTextNode("Cell 1,2"));
tbody.rows[1].insertCell(1);
tbody.rows[1].cells[1].appendChild(document.createTextNode("Cell 2,2"));

//add the table to the document body
document.body.appendChild(table);

```

In this code, the creation of the `<table>` and `<tbody>` elements remains the same. What has changed is the section creating the two rows, which now makes use of the HTML DOM table properties and methods. To create the first row, the `insertRow()` method is called on the `<tbody>` element with an argument of 0, which indicates the position in which the row should be placed. After that point, the row can be referenced by `tbody.rows[0]` because it is automatically created and added into the `<tbody>` element in position 0.

Creating a cell is done in a similar way — by calling `insertCell()` on the `<tr>` element and passing in the position in which the cell should be placed. The cell can then be referenced by `tbody.rows[0].cells[0]`, because the cell has been created and inserted into the row in position 0.

Using these properties and methods to create a table makes the code much more logical and readable, although technically both sets of code are correct.

Using NodeLists

Understanding a `NodeList` object and its relatives, `NamedNodeMap` and `HTMLCollection`, is critical to a good understanding of the DOM as a whole. Each of these collections is considered “live,” which is to say that they are updated when the document structure changes such that they are always current with the most accurate information. In reality, all `NodeList` objects are queries that are run against the DOM document whenever they are accessed. For instance, the following results in an infinite loop:

```

var divs = document.getElementsByTagName("div"),
    i,
    div;

for (i=0; i < divs.length; i++){
    div = document.createElement("div");
    document.body.appendChild(div);
}

```

The first part of this code gets an `HTMLCollection` of all `<div>` elements in the document. Since that collection is “live,” any time a new `<div>` element is added to the page, it gets added into the collection. Since the browser doesn’t want to keep a list of all the collections that were created, the collection is updated only when it is accessed again. This creates an interesting problem in terms of a loop such as the one in this example. Each time through the loop, the condition `i < divs.length` is being evaluated. That means the query to get all `<div>` elements is being run. Because the body of the loop creates a new `<div>` element and adds it to the document, the value of `divs.length` increments each time through the loop; thus `i` will never equal `divs.length` since both are being incremented.

Any time you want to iterate over a `NodeList`, it’s best to initialize a second variable with the length and then compare the iterator to that variable, as shown in the following example:

```
var divs = document.getElementsByTagName("div"),
    i,
    len,
    div;

for (i=0, len=divs.length; i < len; i++) {
    div = document.createElement("div");
    document.body.appendChild(div);
}
```

In this example, a second variable, `len`, is initialized. Since `len` contains a snapshot of `divs.length` at the time the loop began, it prevents the infinite loop that was experienced in the previous example. This technique has been used through this chapter to demonstrate the preferred way of iterating over `NodeList` objects.

Generally speaking, it is best to limit the number of times you interact with a `NodeList`. Since a query is run against the document each time, try to cache frequently used values retrieved from a `NodeList`.

SUMMARY

The Document Object Model (DOM) is a language-independent API for accessing and manipulating HTML and XML documents. DOM Level 1 deals with representing HTML and XML documents as a hierarchy of nodes that can be manipulated to change the appearance and structure of the underlying documents using JavaScript.

The DOM is made up of a series of node types, as described here:

- The base node type is `Node`, which is an abstract representation of an individual part of a document; all other types inherit from `Node`.
- The `Document` type represents an entire document and is the root node of a hierarchy. In JavaScript, the `document` object is an instance of `Document`, which allows for querying and retrieval of nodes in a number of different ways.
- An `Element` node represents all HTML or XML elements in a document and can be used to manipulate their contents and attributes.
- Other node types exist for text contents, comments, document types, the CDATA section, and document fragments.

DOM access works as expected in most cases, although there are often complications when working with `<script>` and `<style>` elements. Since these elements contain scripting and stylistic information, respectively, they are often treated differently in browsers than other elements.

Perhaps the most important thing to understand about the DOM is how it affects overall performance. DOM manipulations are some of the most expensive operations that can be done in JavaScript, with `NodeList` objects being particularly troublesome. `NodeList` objects are “live,” meaning that a query is run every time the object is accessed. Because of these issues, it is best to minimize the number of DOM manipulations.

11

DOM Extensions

WHAT'S IN THIS CHAPTER?

- Understanding the Selectors API
- Using HTML5 DOM extensions
- Working with proprietary DOM extensions

Even though the DOM is a fairly well-defined API, it is also frequently augmented with both standards-based and proprietary extensions to provide additional functionality. Prior to 2008, almost all of the DOM extensions found in browsers were proprietary. After that point, the W3C went to work to codify some of the proprietary extensions that had become de facto standards into formal specifications.

The two primary standards specifying DOM extensions are the Selectors API and HTML5. These both arose out of needs in the development community and a desire to standardize certain approaches and APIs. There is also a smaller Element Traversal specification with additional DOM properties. Proprietary extensions still exist, even though these two specifications, especially HTML5, cover a large number of DOM extensions. The proprietary extensions are also covered within this chapter.

SELECTORS API

One of the most popular capabilities of JavaScript libraries is the ability to retrieve a number of DOM elements matching a pattern specified using CSS selectors. Indeed, the library jQuery (www.jquery.com) is built completely around the CSS selector queries of a DOM document in order to retrieve references to elements instead of using `getElementById()` and `getElementsByName()`.

The Selectors API (www.w3.org/TR/selectors-api) was started by the W3C to specify native support for CSS queries in browsers. All JavaScript libraries implementing this feature had to

do so by writing a rudimentary CSS parser and then using existing DOM methods to navigate the document and identify matching nodes. Although library developers worked tirelessly to speed up the performance of such processing, there was only so much that could be done while the code ran in JavaScript. By making this a native API, the parsing and tree navigating can be done at the browser level in a compiled language and thus tremendously increase the performance of such functionality.

At the core of Selectors API Level 1 are two methods: `querySelector()` and `querySelectorAll()`. On a conforming browser, these methods are available on the `Document` type and on the `Element` type. Selectors API Level 1 was fully implemented in Internet Explorer 8+, Firefox 3.5+, Safari 3.1+, Chrome, and Opera 10+.

The `querySelector()` Method

The `querySelector()` method accepts a CSS query and returns the first descendant element that matches the pattern or `null` if there is no matching element. Here is an example:



```
//get the body element
var body = document.querySelector("body");

//get the element with the ID "myDiv"
var myDiv = document.querySelector("#myDiv");

//get first element with a class of "selected"
var selected = document.querySelector(".selected");

//get first image with class of "button"
var img = document.body.querySelector("img.button");
```

SelectorsAPIExample01.htm

When the `querySelector()` method is used on the `Document` type, it starts trying to match the pattern from the document element; when used on an `Element` type, the query attempts to make a match from the descendants of the element only.

The CSS query may be as complex or as simple as necessary. If there's a syntax error or an unsupported selector in the query, then `querySelector()` throws an error.

The `querySelectorAll()` Method

The `querySelectorAll()` method accepts the same single argument as `querySelector()` — the CSS query — but returns all matching nodes instead of just one. This method returns a static instance of `NodeList`.

To clarify, the return value is actually a `NodeList` with all of the expected properties and methods, but its underlying implementation acts as a snapshot of elements rather than a dynamic query that is constantly reexecuted against a document. This implementation eliminates most of the performance overhead associated with the use of `NodeList` objects.

Any call to `querySelectorAll()` with a valid CSS query will return a `NodeList` object regardless of the number of matching elements; if there are no matches, the `NodeList` is empty.

As with `querySelector()`, the `querySelectorAll()` method is available on the Document, DocumentFragment, and Element types. Here are some examples:



```
//get all <em> elements in a <div> (similar to getElementsByTagName("em"))
var ems = document.getElementById("myDiv").querySelectorAll("em");

//get all elements with class of "selected"
var selecteds = document.querySelectorAll(".selected");

//get all <strong> elements inside of <p> elements
var strongs = document.querySelectorAll("p strong");
```

SelectorsAPIExample02.htm

The resulting NodeList object may be iterated over using either `item()` or bracket notation to retrieve individual elements. Here's an example:

```
var i, len, strong;
for (i=0, len=strongs.length; i < len; i++){
    strong = strongs[i]; //or strongs.item(i)
    strong.className = "important";
}
```

As with `querySelector()`, `querySelectorAll()` throws an error when the CSS selector is not supported by the browser or if there's a syntax error in the selector.

The `matchesSelector()` Method

The Selectors API Level 2 specification introduces another method called `matchesSelector()` on the Element type. This method accepts a single argument, a CSS selector, and returns `true` if the given element matches the selector or `false` if not. For example:

```
if (document.body.matchesSelector("body.page1")){
    //true
}
```

This method allows you to easily check if an element would be returned by `querySelector()` or `querySelectorAll()` when you already have the element reference.

As of mid-2011, no browser supports `matchesSelector()`; however, several support experimental implementations. Internet Explorer 9+ supports the method via `msMatchesSelector()`, Firefox 3.6+ supports it via `mozMatchesSelector()`, and Safari 5+ and Chrome support it via `webkitMatchesSelector()`. To make use of this method now, you may want to create a wrapper function, such as:

```
function matchesSelector(element, selector){
    if (element.matchesSelector){
        return element.matchesSelector(selector);
    } else if (element.msMatchesSelector){
        return element.msMatchesSelector(selector);
    } else if (element.mozMatchesSelector){
        return element.mozMatchesSelector(selector);
    }
```

```

        } else if (element.webkitMatchesSelector){
            return element.webkitMatchesSelector(selector);
        } else {
            throw new Error("Not supported.");
        }
    }

    if (matchesSelector(document.body, "body.page1")){
        //do something
    }
}

```

SelectorsAPIExample03.htm

ELEMENT TRAVERSAL

Prior to version 9, Internet Explorer did not return text nodes for white space in between elements while all of the other browsers did. This led to differences in behavior when using properties such as `childNodes` and `firstChild`. In an effort to equalize the differences while still remaining true to the DOM specification, a new group of properties was defined in the Element Traversal (www.w3.org/TR/ElementTraversal/).

The Element Traversal API adds five new properties to DOM elements:

- `childElementCount` — Returns the number of child elements (excludes text nodes and comments).
- `firstElementChild` — Points to the first child that is an element. Element-only version of `firstChild`.
- `lastElementChild` — Points to the last child that is an element. Element-only version of `lastChild`.
- `previousElementSibling` — Points to the previous sibling that is an element. Element-only version of `previousSibling`.
- `nextElementSibling` — Points to the next sibling that is an element. Element-only version of `nextSibling`.

Supporting browsers add these properties to all DOM elements to allow for easier traversal of DOM elements without the need to worry about white space text nodes.

As an example, iterating over all child elements of a particular element in a traditional cross-browser way looks like this:

```

var i,
    len,
    child = element.firstChild;
while(child != element.lastChild){
    if (child.nodeType == 1){ //check for an element
        processChild(child);
    }
    child = child.nextSibling;
}

```

Using the Element Traversal properties allows a simplification of the code:

```
var i,
    len,
    child = element.firstChild;
while(child != element.lastElementChild){
    processChild(child); //already know it's an element
    child = child.nextElementSibling;
}
```

Element Traversal is implemented in Internet Explorer 9+, Firefox 3.5+, Safari 4+, Chrome, and Opera 10+.

HTML5

HTML5 represents a radical departure from the tradition of HTML. In all previous HTML specifications, the descriptions stopped short of describing any JavaScript interfaces, instead focusing purely on the markup of the language and deferring JavaScript bindings to the DOM specification.

The HTML5 specification, on the other hand, contains a large amount of JavaScript APIs designed for use with the markup additions. Part of these APIs overlap with the DOM and define DOM extensions that browsers should provide.



Because the scope of HTML5 is vast, this section focuses only on the parts that affect all DOM nodes. Other parts of HTML5 are discussed in their related topics throughout the book.

Class-Related Additions

One of the major changes in web development since the time HTML4 was adopted is the increased usage of the `class` attribute to indicate both stylistic and semantic information about elements. This caused a lot of JavaScript interaction with CSS classes, including the dynamic changing of classes and querying the document to find elements with a given class or set of classes. To adapt to developers and their newfound appreciation of the `class` attribute, HTML5 introduces a number of changes to make CSS class usage easier.

The `getElementsByClassName()` Method

One of HTML5's most popular additions is `getElementsByClassName()`, which is available on the `document` object and on all HTML elements. This method evolved out of JavaScript libraries

that implemented it using existing DOM features and is provided as a native implementation for performance reasons.

The `getElementsByClassName()` method accepts a single argument, which is a string containing one or more class names, and returns a `NodeList` containing all elements that have all of the specified classes applied. If multiple class names are specified, then the order is considered unimportant. Here are some examples:

```
//get all elements with a class containing "username" and "current", though it
//doesn't matter if one is declared before the other
var allCurrentUsernames = document.getElementsByClassName("username current");

//get all elements with a class of "selected" that exist in myDiv's subtree
var selected = document.getElementById("myDiv").getElementsByClassName("selected");
```

When this method is called, it will return only elements in the subtree of the root from which it was called. Calling `getElementsByClassName()` on `document` always returns all elements with matching class names, whereas calling it on an element will return only descendant elements.

This method is useful for attaching events to classes of elements rather than using IDs or tag names. Keep in mind that since the returned value is a `NodeList`, there are the same performance issues as when you're using `getElementsByTagName()` and other DOM methods that return `NodeList` objects.

The `getElementsByClassName()` method was implemented in Internet Explorer 9+, Firefox 3+, Safari 3.1+, Chrome, and Opera 9.5+.

The `classList` Property

In class name manipulation, the `className` property is used to add, remove, and replace class names. Since `className` contains a single string, it's necessary to set its value every time a change needs to take place, even if there are parts of the string that should be unaffected. For example, consider the following HTML code:

```
<div class="bd user disabled">...</div>
```

This `<div>` element has three classes assigned. To remove one of these classes, you need to split the `class` attribute into individual classes, remove the unwanted class, and then create a string containing the remaining classes. Here is an example:

```
//remove the "user" class

//first, get list of class names
var classNames = div.className.split(/\s+/);

//find the class name to remove
var pos = -1,
    i,
    len;
for (i=0, len=classNames.length; i < len; i++){
  if (classNames[i] == "user"){
    pos = i;
```

```

        break;
    }
}

//remove the class name
classNames.splice(i,1);

//set back the class name
div.className = classNames.join(" ");

```

All of this code is necessary to remove the "user" class from the <div> element's class attribute. A similar algorithm must be used for replacing class names and detecting if a class name is applied to an element. Adding class names can be done by using string concatenation, but checks must be done to ensure that you're not applying the same class more than one time. Many JavaScript libraries implement methods to aid in these behaviors.

HTML5 introduces a way to manipulate class names in a much simpler and safer manner through the addition of the `classList` property for all elements. The `classList` property is an instance of a new type of collection named `DOMTokenList`. As with other DOM collections, `DOMTokenList` has a `length` property to indicate how many items it contains, and individual items may be retrieved via the `item()` method or using bracket notation. It also has the following additional methods:

- `add(value)` — Adds the given string value to the list. If the value already exists, it will not be added.
- `contains(value)` — Indicates if the given value exists in the list (`true` if so; `false` if not).
- `remove(value)` — Removes the given string value from the list.
- `toggle(value)` — If the value already exists in the list, it is removed. If the value doesn't exist, then it's added.

The entire block of code in the previous example can quite simply be replaced with the following:

```
div.classList.remove("user");
```

Using this code ensures that the rest of the class names will be unaffected by the change. The other methods also greatly reduce the complexity of the basic operations, as shown in these examples:

```

//remove the "disabled" class
div.classList.remove("disabled");

//add the "current" class
div.classList.add("current");

//toggle the "user" class
div.classList.toggle("user");

//figure out what's on the element now
if (div.classList.contains("bd") && !div.classList.contains("disabled")){
    //do something
}

//iterate over the class names

```

```
for (var i=0, len=div.classList.length; i < len; i++){
    doSomething(div.classList[i]);
}
```

The addition of the `classList` property makes it unnecessary to access the `className` property unless you intend to completely remove or completely overwrite the element's `class` attribute. The `classList` property is implemented in Firefox 3.6+ and Chrome.

Focus Management

HTML5 adds functionality to aid with focus management in the DOM. The first is `document.activeElement`, which always contains a pointer to the DOM element that currently has focus. An element can receive focus automatically as the page is loading, via user input (typically using the Tab key), or programmatically using the `focus()` method. For example:

```
var button = document.getElementById("myButton");
button.focus();
alert(document.activeElement === button); //true
```

By default, `document.activeElement` is set to `document.body` when the document is first loaded. Before the document is fully loaded, `document.activeElement` is `null`.

The second addition is `document.hasFocus()`, which returns a Boolean value indicating if the document has focus:

```
var button = document.getElementById("myButton");
button.focus();
alert(document.hasFocus()); //true
```

Determining if the document has focus allows you to determine if the user is interacting with the page.

This combination of being able to query the document to determine which element has focus and being able to ask the document if it has focus is of the utmost importance for web application accessibility. One of the key components of accessible web applications is proper focus management, and being able to determine which elements currently have focus is a major improvement over the guesswork of the past.

These properties are implemented in Internet Explorer 4+, Firefox 3+, Safari 4+, Chrome, and Opera 8+.

Changes to `HTMLDocument`

HTML5 extends the `HTMLDocument` type to include more functionality. As with other DOM extensions specified in HTML5, the changes are based on proprietary extensions that are well-supported across browsers. As such, even though the standardization of the extensions is relatively new, some browsers have supported the functionality for a while.

The readyState Property

Internet Explorer 4 was the first to introduce a `readyState` property on the `document` object. Other browsers then followed suit and this property was eventually formalized in HTML5. The `readyState` property for `document` has two possible values:

- `loading` — The document is loading.
- `complete` — The document is completely loaded.

The best way to use the `document.readyState` property is as an indicator that the document has loaded. Before this property was widely available, you would need to add an `onload` event handler to set a flag indicating that the document was loaded. Basic usage:

```
if (document.readyState == "complete") {
    //do something
}
```

The `readyState` property is implemented in Internet Explorer 4+, Firefox 3.6+, Safari, Chrome, and Opera 9+.

Compatibility Mode

With the introduction of Internet Explorer 6 and the ability to render a document in either standards or quirks mode, it became necessary to determine in which mode the browser was rendering the page. Internet Explorer added a property on the `document` named `compatMode` whose sole job is to indicate what rendering mode the browser is in. As shown in the following example, when in standards mode, `document.compatMode` is equal to `"CSS1Compat"`; when in quirks mode, `document.compatMode` is `"BackCompat"`:

```
if (document.compatMode == "CSS1Compat") {
    alert("Standards mode");
} else {
    alert("Quirks mode");
}
```

This property was later implemented by Firefox, Safari 3.1+, Opera, and Chrome. As a result, the property was added to HTML5 to formalize its implementation.

The head Property

HTML5 introduces `document.head` to point to the `<head>` element of a document to complement `document.body`, which points to the `<body>` element of the document. You can retrieve a reference to the `<head>` element using this property or the fallback method:

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

This code uses `document.head`, if available; otherwise it falls back to the old method of using `getElementsByTagName()`.

The `document.head` property is implemented in Chrome and Safari 5.

Character Set Properties

HTML5 describes several new properties dealing with the character set of the document. The `charset` property indicates the actual character set being used by the document and can also be used to specify a new character set. By default, this value is "UTF-16", although it may be changed by using `<meta>` elements or response headers or through setting the `charset` property directly. Here's an example:

```
alert(document.charset); // "UTF-16"
document.charset = "UTF-8";
```

The `defaultCharset` property indicates what the default character set for the document should be based on default browser and system settings. The values of `charset` and `defaultCharset` may be different if the document doesn't use the default character set, as in this example:

```
if (document.charset != document.defaultCharset) {
    alert("Custom character set being used.");
}
```

These properties allow greater insight into, and control over, the character encoding used on the document. If used properly, this should allow web developers to ensure that their page or application is being viewed properly.

The `document.charset` property is supported by Internet Explorer, Firefox, Safari, Opera, and Chrome. The `document.defaultCharset` property is supported in Internet Explorer, Safari, and Chrome.

Custom Data Attributes

HTML5 allows elements to be specified with nonstandard attributes prefixed with `data-` in order to provide information that isn't necessary to the rendering or semantic value of the element. These attributes can be added as desired and named anything, provided that the name begins with `data-`. Here is an example:

```
<div id="myDiv" data-appId="12345" data-myname="Nicholas"></div>
```

When a custom data attribute is defined, it can be accessed via the `dataset` property of the element. The `dataset` property contains an instance of `DOMStringMap` that is a mapping of name-value pairs. Each attribute of the format `data-name` is represented by a property with a name equivalent to the attribute but without the `data-` prefix (for example, attribute `data-myname` is represented by a property called `myname`). Example usage:

```
//the methods used in this example are for illustrative purposes only
var div = document.getElementById("myDiv");
//get the values
var appId = div.dataset.appId;
```

```

var myName = div.dataset.myname;

//set the value
div.dataset.appId = 23456;
div.dataset.myname = "Michael";

//is there a "myname" value?
if (div.dataset.myname){
    alert("Hello, " + div.dataset.myname);
}

```

Custom data attributes are useful when nonvisual data needs to be tied to an element for some other form of processing. This is a common technique to use for link tracking and mashups in order to better identify parts of a page.

As of the time of this writing, only Firefox 6+ and Chrome have implemented this feature.

Markup Insertion

Although the DOM provides fine-grained control over nodes in a document, it can be cumbersome when attempting to inject a large amount of new HTML into the document. Instead of creating a series of DOM nodes and connecting them in the correct order, it's much easier (and faster) to use one of the markup insertion capabilities to inject a string of HTML. The following DOM extensions have been standardized in HTML5 for this purpose.

The innerHTML Property

When used in read mode, `innerHTML` returns the HTML representing all of the child nodes, including elements, comments, and text nodes. When used in write mode, `innerHTML` completely replaces all of the child nodes in the element with a new DOM subtree based on the specified value. Consider the following HTML code:

```

<div id="content">
    <p>This is a <strong>paragraph</strong> with a list following it.</p>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
</div>

```

For the `<div>` element in this example, the `innerHTML` property returns the following string:

```

<p>This is a <strong>paragraph</strong> with a list following it.</p>
<ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ul>

```

The exact text returned from `innerHTML` differs from browser to browser. Internet Explorer and Opera tend to convert all tags to uppercase, whereas Safari, Chrome, and Firefox return HTML in

the way it is specified in the document, including white space and indentation. You cannot depend on the returned value of `innerHTML` being exactly the same from browser to browser.

When used in write mode, `innerHTML` parses the given string into a DOM subtree and replaces all of the existing child nodes with it. Because the string is considered to be HTML, all tags are converted into elements in the standard way that the browser handles HTML (again, this differs from browser to browser). Setting simple text without any HTML tags, as shown here, sets the plain text:

```
div.innerHTML = "Hello world!";
```

Setting `innerHTML` to a string containing HTML behaves quite differently as `innerHTML` parses them. Consider the following example:

```
div.innerHTML = "Hello & welcome, <b>"reader"</b>";
```

The result of this operation is as follows:

```
<div id="content">Hello & welcome, <b>"reader"</b></div>
```

After setting `innerHTML`, you can access the newly created nodes as you would any other nodes in the document.



Setting `innerHTML` causes the HTML string to be parsed by the browser into an appropriate DOM tree. This means that setting `innerHTML` and then reading it back typically results in a different string being returned. This is because the returned string is the result of serializing the DOM subtree that was created for the original HTML string.

There are some limitations to `innerHTML`. For one, `<script>` elements cannot be executed when inserted via `innerHTML` in most browsers. Internet Explorer, 8 and earlier, is the only browser that allows this but only as long as the `defer` attribute is specified and the `<script>` element is preceded by what Microsoft calls a *scoped element*. The `<script>` element is considered a *NoScope element*, which effectively means that it has no visual representation on the page, like a `<style>` element or a comment. Internet Explorer strips out all NoScope elements from the beginning of strings inserted via `innerHTML`, which means the following won't work:

```
div.innerHTML = "<script defer>alert('hi');</script>"; //won't work
```

In this case, the `innerHTML` string begins with a NoScope element, so the entire string becomes empty. To allow this script to work appropriately, you must precede it with a scoped element, such as a text node or an element without a closing tag such as `<input>`. The following lines will all work:

```
div.innerHTML = "<script defer>alert('hi');</script>";
div.innerHTML = "<div>&nbsp;</div><script defer>alert('hi');</script>";
div.innerHTML = "<input type='hidden'><script defer>alert('hi');</script>";
```

The first line results in a text node being inserted immediately before the `<script>` element. You may need to remove this after the fact so as not to disrupt the flow of the page. The second line has a similar approach, using a `<div>` element with a nonbreaking space. An empty `<div>` alone won't do the trick; it must contain some content that will force a text node to be created. Once again, the first node may need to be removed to avoid layout issues. The third line uses a hidden `<input>` field to accomplish the same thing. Since it doesn't affect the layout of the page, this may be the optimal case for most situations.

In most browsers, the `<style>` element causes similar problems with `innerHTML`. Most browsers support the insertion of `<style>` elements using `innerHTML` in the exact way you'd expect, as shown here:

```
div.innerHTML = "<style type=\"text/css\">body {background-color: red; }</style>";
```

In Internet Explorer 8 and earlier, `<style>` is yet another NoScope element, so it must be preceded by a scoped element such as this:

```
div.innerHTML = "_<style type=\"text/css\">body {background-color: red; }</style>";  
div.removeChild(div.firstChild);
```

The `innerHTML` property is not available on all elements. The following elements do not support `innerHTML`: `<col>`, `<colgroup>`, `<frameset>`, `<head>`, `<html>`, `<style>`, `<table>`, `<tbody>`, `<thead>`, `<tfoot>`, and `<tr>`. Additionally, Internet Explorer 8 and earlier do not support `innerHTML` on the `<title>` element.



Firefox's support of `innerHTML` is stricter in XHTML documents served with the application/xhtml+xml content type. When using `innerHTML` in XHTML documents, you must specify well-formed XHTML code. If the code is not well formed, setting `innerHTML` fails silently.

Whenever you're using `innerHTML` to insert HTML from a source external to your code, it's important to sanitize the HTML before passing it through to `innerHTML`. Internet Explorer 8 added the `window.toStaticHTML()` method for this purpose. This method takes a single argument, an HTML string, and returns a sanitized version that has all script nodes and script event-handler attributes removed from the source. Following is an example:

```
var text = "<a href=\"#\" onclick=\"alert('hi')\">Click Me</a>";  
var sanitized = window.toStaticHTML(text); //Internet Explorer 8 only  
alert(sanitized); //<a href="#">Click Me</a>"
```

This example runs an HTML link string through `toStaticHTML()`. The sanitized text no longer has the `onclick` attribute present. Though Internet Explorer 8 is the only browser with this native functionality, it is still advisable to be careful when using `innerHTML` and inspect the text manually before inserting it, if possible.

The outerHTML Property

When `outerHTML` is called in read mode, it returns the HTML of the element on which it is called, as well as its child nodes. When called in write mode, `outerHTML` replaces the node on which it is called with the DOM subtree created from parsing the given HTML string. Consider the following HTML code:



Available for
download on
Wrox.com

```
<div id="content">
  <p>This is a <strong>paragraph</strong> with a list following it.</p>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
</div>
```

OuterHTMLExample01.htm

When `outerHTML` is called on the `<div>` in this example, the same code is returned, including the code for the `<div>`. Note that there may be differences based on how the browser parses and interprets the HTML code. (These are the same types of differences you'll notice when using `innerHTML`.)

Use `outerHTML` to set a value in the following manner:

```
div.outerHTML = "<p>This is a paragraph.</p>";
```

This code performs the same operation as the following DOM code:

```
var p = document.createElement("p");
p.appendChild(document.createTextNode("This is a paragraph."));
div.parentNode.replaceChild(p, div);
```

The new `<p>` element replaces the original `<div>` element in the DOM tree.

The `outerHTML` property is supported by Internet Explorer 4+, Safari 4+, Chrome, and Opera 8+. Firefox, as of version 7, still does not support `outerHTML`.

The `insertAdjacentHTML()` Method

The last addition for markup insertion is the `insertAdjacentHTML()` method. This method also originated in Internet Explorer and accepts two arguments: the position in which to insert and the HTML text to insert. The first argument must be one of the following values:

- "beforebegin" — Insert just before the element as a previous sibling.
- "afterbegin" — Insert just inside of the element as a new child or series of children before the first child.

- "beforeend" — Insert just inside of the element as a new child or series of children after the last child.
- "afterend" — Insert just after the element as a next sibling.

Note that each of these values is case insensitive. The second argument is parsed as an HTML string (the same as with `innerHTML/outerHTML`) and will throw an error if the value cannot be properly parsed. Basic usage is as follows:

```
//insert as previous sibling
element.insertAdjacentHTML("beforebegin", "<p>Hello world!</p>");

//insert as first child
element.insertAdjacentHTML("afterbegin", "<p>Hello world!</p>");

//insert as last child
element.insertAdjacentHTML("beforeend", "<p>Hello world!</p>");

//insert as next sibling
element.insertAdjacentHTML("afterend", "<p>Hello world!</p>");
```

The `insertAdjacentHTML()` method is supported in Internet Explorer, Firefox 8+, Safari, Opera, and Chrome.

Memory and Performance Issues

Replacing child nodes using the methods in this section may cause memory problems in browsers, especially Internet Explorer. The problem occurs when event handlers or other JavaScript objects are assigned to subtree elements that are removed. If an element has an event handler (or a JavaScript object as a property), and one of these properties is used in such a way that the element is removed from the document tree, the binding between the element and the event handler remains in memory. If this is repeated frequently, memory usage increases for the page. When using `innerHTML`, `outerHTML`, and `insertAdjacentHTML()`, it's best to manually remove all event handlers and JavaScript object properties on elements that are going to be removed. (Event handlers are discussed further in Chapter 13.)

Using these properties does have an upside, especially when using `innerHTML`. Generally speaking, inserting a large amount of new HTML is more efficient through `innerHTML` than through multiple DOM operations to create nodes and assign relationships between them. This is because an HTML parser is created whenever a value is set to `innerHTML` (or `outerHTML`). This parser runs in browser-level code (often written in C++), which is much faster than JavaScript. That being said, the creation and destruction of the HTML parser does have some overhead, so it's best to limit the number of times you set `innerHTML` or `outerHTML`. For example, the following creates a number of list items using `innerHTML`:

```
for (var i=0, len=values.length; i < len; i++){
    ul.innerHTML += "<li>" + values[i] + "</li>"; //avoid!!
}
```

This code is inefficient, because it sets `innerHTML` once each time through the loop. Furthermore, this code is reading `innerHTML` each time through the loop, meaning that `innerHTML` is being accessed twice each time through the loop. It's best to build up the string separately and assign it using `innerHTML` just once at the end, like this:

```
var itemsHtml = "";
for (var i=0, len=values.length; i < len; i++){
    itemsHtml += "<li>" + values[i] + "</li>";
}
ul.innerHTML = itemsHtml;
```

This example is more efficient, limiting the use of `innerHTML` to one assignment.

The `scrollIntoView()` Method

One of the issues not addressed by the DOM specification is how to scroll areas of a page. To fill this gap, browsers implemented several methods that control scrolling in different ways. Of the various proprietary methods, only `scrollIntoView()` was selected for inclusion in HTML5.

The `scrollIntoView()` method exists on all HTML elements and scrolls the browser window or container element so the element is visible in the viewport. If an argument of `true` is supplied or the argument is omitted, the window scrolls so that the top of the element is at the top of the viewport (if possible); otherwise, the element is scrolled so that it is fully visible in the viewport but may not be aligned at the top. For example:

```
//make sure this element is visible
document.forms[0].scrollIntoView();
```

This method is most useful for getting the user's attention when something has happened on the page. Note that setting focus to an element also causes the browser to scroll the element into view so that the focus can properly be displayed.

The `scrollIntoView()` method is supported in Internet Explorer, Firefox, Safari, Opera, and Chrome.

PROPRIETARY EXTENSIONS

Although all browser vendors understand the importance of adherence to standards, they all have a history of adding proprietary extensions to the DOM in order to fill perceived gaps in functionality. Though this may seem like a bad thing on the surface, proprietary extensions have given the web development community many important features that were later codified into standards such as HTML5.

There are still a large amount of DOM extensions that are proprietary in nature and haven't been incorporated into standards. This doesn't mean that they won't later be adopted as standards, just that at the time of this writing, they remain proprietary and adopted by only a subset of browsers.

Document Mode

Internet Explorer 8 introduced a new concept called *document mode*. A page's document mode determines to which features it has access. This means that there's a specific level of CSS support, a specific number of features available for scripting through JavaScript, and a specific way that doctypes are treated. As of Internet Explorer 9, there are four different document modes:

- Internet Explorer 5 — Renders the page in quirks mode (the default mode for Internet Explorer 5). The new features in Internet Explorer 8 and higher are not available.
- Internet Explorer 7 — Renders the page in Internet Explorer 7 standards mode. The new features in Internet Explorer 8 and higher are not available.
- Internet Explorer 8 — Renders the page in Internet Explorer 8 standards mode. New features in Internet Explorer 8 are available, so you can access the Selectors API, more CSS 2 selectors, some CSS3 features, and some HTML 5 features. The Internet Explorer 9 features are not available.
- Internet Explorer 9 — Renders the page in Internet Explorer 9 standards mode. New features in Internet Explorer 9 are available, such as ECMAScript 5, full CSS3 support, and more HTML5 features. This document mode is considered the most advanced.

The concept of document mode is very important for understanding how Internet Explorer 8 and higher works.

You can force a particular document mode by using the X-UA-Compatible HTTP header or by using the <meta> tag equivalent:

```
<meta http-equiv="X-UA-Compatible" content="IE=IEVersion">
```

There are several different possible values for the Internet Explorer version in this field, and they don't necessarily map to the four document modes:

- Edge — Always put the document into the most recent document mode available. Doctype is ignored. For Internet Explorer 8, this forces the document mode to Internet Explorer 8 standards all the time; for Internet Explorer 9, the document mode is forced to Internet Explorer 9 standards mode.
- EmulateIE9 — If a doctype is present, set the document mode to Internet Explorer 9 standards and otherwise set the document mode to Internet Explorer 5.
- EmulateIE8 — If a doctype is present, set the document mode to Internet Explorer 8 standards and otherwise set the document mode to Internet Explorer 5.
- EmulateIE7 — If a doctype is present, set the document mode to Internet Explorer 7 standards and otherwise set the document mode to Internet Explorer 5.
- 9 — Force document mode to be Internet Explorer 9 standards. Doctype is ignored.
- 8 — Force document mode to be Internet Explorer 8 standards. Doctype is ignored.
- 7 — Force document mode to be Internet Explorer 7 standards. Doctype is ignored.
- 5 — Force document mode to be Internet Explorer 5. Doctype is ignored.

For example, to make the document mode behave as it would in Internet Explorer 7, you can use the following:

```
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7">
```

Whereas to force Internet Explorer 7 standards mode regardless of doctype, use this:

```
<meta http-equiv="X-UA-Compatible" content="IE=7">
```

You are not required to have an `X-UA-Compatible` field set on pages. By default, the browser uses the doctype to determine if the document mode should be the best available standards mode or quirks mode.

You can determine the document mode for a given page using the `document.documentElementMode` property, a new feature in Internet Explorer 8, which returns the version of the document mode being used (in version 9, the values can be 5, 7, 8, or 9):

```
var mode = document.documentElementMode;
```

Using this property can give you a hint as to how the page is going to behave. This property is available in all document modes.

The `children` Property

The differences in how Internet Explorer prior to version 9 and other browsers interpret white space in text nodes led to the creation of the `children` property. The `children` property is an `HTMLCollection` that contains only an element's child nodes that are also elements. Otherwise, the `children` property is the same as `childNodes` and may contain the same items when an element has only elements as children. The `children` property is accessed as follows:

```
var childCount = element.children.length;
var firstChild = element.children[0];
```

The `children` collection is supported in Internet Explorer 5, Firefox 3.5, Safari 2 (buggy), Safari 3 (complete), Opera 8, and Chrome (all versions). Internet Explorer 8 and earlier also return comments in the `children` collection (while Internet Explorer 9 and later do not).

The `contains()` Method

It's often necessary to determine if a given node is a descendant of another. Internet Explorer first introduced the `contains()` method as a way of providing this information without necessitating a walk up the DOM document tree. The `contains()` method is called on the ancestor node from which the search should begin and accepts a single argument, which is the suspected descendant node. If the node exists as a descendant of the root node, the method returns `true`; otherwise it returns `false`. Here is an example:

```
alert(document.documentElement.contains(document.body)); //true
```

This example tests to see if the `<body>` element is a descendant of the `<html>` element, which returns `true` in all well-formed HTML pages. The `contains()` method is supported in Internet Explorer, Firefox 9+, Safari, Opera, and Chrome.

There is another way of determining node relationships by using the DOM Level 3 `compareDocumentPosition()` method, which is supported in Internet Explorer 9+, Firefox, Safari, Opera 9.5+, and Chrome. This method determines the relationship between two nodes and returns a bitmask indicating the relationship. The values for the bitmask are as shown in the following table.

MASK	RELATIONSHIP BETWEEN NODES
1	Disconnected (The passed-in node is not in the document.)
2	Precedes (The passed-in node appears in the DOM tree prior to the reference node.)
4	Follows (The passed-in node appears in the DOM tree after the reference node.)
8	Contains (The passed-in node is an ancestor of the reference node.)
16	Is contained by (The passed-in node is a descendant of the reference node.)

To mimic the `contains()` method, you will be interested in the 16 mask. The result of `compareDocumentPosition()` can be bitwise ANDed to determine if the reference node contains the given node. Here is an example:

```
var result = document.documentElement.compareDocumentPosition(document.body);
alert (!! (result & 16));
```

When this code is executed, the result becomes 20 (4 for “follows” plus 16 for “is contained by”). Applying a bitwise mask of 16 to the result returns a nonzero number, and the two NOT operators convert that value into a Boolean.

A generic `contains` function can be created with a little help using browser and capability detection, as shown here:



```
function contains(refNode, otherNode) {
    if (typeof refNode.contains == "function" &&
        (!client.engine.webkit || client.engine.webkit >= 522)) {
        return refNode.contains(otherNode);
    } else if (typeof refNode.compareDocumentPosition == "function") {
        return !(refNode.compareDocumentPosition(otherNode) & 16);
    } else {
        var node = otherNode.parentNode;
        do {
            if (node === refNode) {
                return true;
            } else {
                node = node.parentNode;
            }
        }
    }
}
```

```

        } while (node !== null);
        return false;
    }
}

```

[ContainsExample02.htm](#)

This function combines three methods of determining if a node is a descendant of another. The first argument is the reference node and the second argument is the node to check for. In the function body, the first check is to see if the `contains()` method exists on `refNode` (capability detection). This part of the code also checks the version of WebKit being used. If the function exists and it's not WebKit (`!client.engine.webkit`), then the code can proceed. Likewise, if the browser is WebKit and at least Safari 3 (WebKit 522 and higher), then the code can proceed. WebKit less than 522 has a `contains()` method that doesn't work properly.

Next is a check to see if the `compareDocumentPosition()` method exists, and the final part of the function walks up the DOM structure from `otherNode`, recursively getting the `parentNode` and checking to see if it's equal to `refNode`. At the very top of the document tree, `parentNode` will be `null` and the loop will end. This is the fallback strategy for older versions of Safari.

Markup Insertion

While the `innerHTML` and `outerHTML` markup insertion properties were adopted by HTML5 from Internet Explorer, there are two others that were not. The two remaining properties that are left out of HTML5 are `innerText` and `outerText`.

The `innerText` Property

The `innerText` property works with all text content contained within an element, regardless of how deep in the subtree the text exists. When used to read the value, `innerText` concatenates the values of all text nodes in the subtree in depth-first order. When used to write the value, `innerText` removes all children of the element and inserts a text node containing the given value. Consider the following HTML code:



```

<div id="content">
    <p>This is a <strong>paragraph</strong> with a list following it.</p>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
</div>

```

[InnerTextExample01.htm](#)

For the `<div>` element in this example, the `innerText` property returns the following string:

```

This is a paragraph with a list following it.
Item 1
Item 2
Item 3

```

Note that different browsers treat white space in different ways, so the formatting may or may not include the indentation in the original HTML code.

Using the `innerText` property to set the contents of the `<div>` element is as simple as this single line of code:



Available for
download on
Wrox.com

```
div.innerText = "Hello world!";
```

[InnerTextExample02.htm](#)

After executing this line of code, the HTML of the page is effectively changed to the following:

```
<div id="content">Hello world!</div>
```

Setting `innerText` removes all of the child nodes that existed before, completely changing the DOM subtree. Additionally, setting `innerText` encodes all HTML syntax characters (less-than, greater-than, quotation marks, and ampersands) that may appear in the text. Here is an example:

```
div.innerText = "Hello & welcome, <b>"reader"\!</b>";
```

[InnerTextExample03.htm](#)

The result of this operation is as follows:

```
<div id="content">Hello & welcome, &lt;b&gt;"reader"\!&lt;/b&gt;</div>
```

Setting `innerText` can never result in anything other than a single text node as the child of the container, so the HTML-encoding of the text must take place in order to keep to that single text node. The `innerText` property is also useful for stripping out HTML tags. By setting the `innerText` equal to the `innerText`, as shown here, all HTML tags are removed:

```
div.innerText = div.innerText;
```

Executing this code replaces the contents of the container with just the text that exists already.

The `innerText` property is supported in Internet Explorer 4+, Safari 3+, Opera 8+, and Chrome. Firefox does not support `innerText`, but it supports an equivalent property called `textContent`. The `textContent` property is specified in DOM Level 3 and is also supported by Internet Explorer 9+, Safari 3+, Opera 10+, and Chrome. For cross-browser compatibility, it's helpful to use functions that check which property is available, as follows:

```
function getInnerText(element){
    return (typeof element.textContent == "string") ?
        element.textContent : element.innerText;
}

function setInnerText(element, text){
    if (typeof element.textContent == "string"){
        element.textContent = text;
    }
}
```

```

    } else {
        element.innerText = text;
    }
}

```

[InnerTextExample05.htm](#)

Each of these methods expects an element to be passed in. Then the element is checked to see if it has the `textContent` property. If it does, then the `typeof element.textContent` should be "string". If `textContent` is not available, each function uses `innerText`. These can be called as follows:

```

setInnerText(div, "Hello world!");
alert(getInnerText(div)); // "Hello world!"

```

Using these functions ensures the correct property is used based on what is available in the browser.



There is a slight difference in the content returned from `innerText` and that returned from `textContent`. Whereas `innerText` skips over inline style and script blocks, `textContent` returns any inline style or script code along with other text. The best ways to avoid cross-browser differences are to use read text only on shallow DOM subtrees or in parts of the DOM where there are no inline styles or inline scripts.

The `outerText` Property

The `outerText` property works in the same way as `innerText` except that it includes the node on which it's called. For reading text values, `outerText` and `innerText` essentially behave in the exact same way. In writing mode, however, `outerText` behaves very differently. Instead of replacing just the child nodes of the element on which it's used, `outerText` actually replaces the entire element, including its child nodes. Consider the following:

```
div.outerText = "Hello world!";
```

This single line of code is equivalent to the following two lines:

```

var text = document.createTextNode("Hello world!");
div.parentNode.replaceChild(text, div);

```

Essentially, the new text node completely replaces the element on which `outerText` was set. After that point in time, the element is no longer in the document and cannot be accessed.

The `outerText` property is supported by Internet Explorer 4+, Safari 3+, Opera 8+, and Chrome. This property is typically not used since it modifies the element on which it is accessed. It is recommended to avoid it whenever possible.

Scrolling

As mentioned previously, scrolling is one area where specifications didn't exist prior to HTML5. While `scrollIntoView()` was standardized in HTML5, there are still several additional proprietary methods available in various browsers. Each of the following methods exists as an extension to the `HTMLElement` type and therefore each is available on all elements:

- `scrollIntoViewIfNeeded(alignCenter)` — Scrolls the browser window or container element so that the element is visible in the viewport only if it's not already visible; if the element is already visible in the viewport, this method does nothing. The optional `alignCenter` argument will attempt to place the element in the center of the viewport if set to `true`. This is implemented in Safari and Chrome.
- `scrollByLines(lineCount)` — Scrolls the contents of the element by the height of the given number of text lines, which may be positive or negative. This is implemented in Safari and Chrome.
- `scrollByPages(pageCount)` — Scrolls the contents of the element by the height of a page, which is determined by the height of the element. This is implemented in Safari and Chrome.

Keep in mind that `scrollIntoView()` and `scrollIntoViewIfNeeded()` act on the element's container, whereas `scrollByLines()` and `scrollByPages()` affect the element itself. Following is an example of how this may be used:

```
//scroll body by five lines
document.body.scrollByLines(5);

//make sure this element is visible only if it's not already
document.images[0].scrollIntoViewIfNeeded();

//scroll the body back up one page
document.body.scrollByPages(-1);
```

Because `scrollIntoView()` is the only method supported in all browsers, this is typically the only one used.

SUMMARY

While the DOM specifies the core API for interacting with XML and HTML documents, there are several specifications that provide extensions to the standard DOM. Many of the extensions are based on proprietary extensions that later became de facto standards as other browsers began to mimic their functionality. The three specifications covered in this chapter are:

- **Selectors API**, which defines two methods for retrieving DOM elements based on CSS selectors: `querySelector()` and `querySelectorAll()`.
- **Element Traversal**, which defines additional properties on DOM elements to allow easy traversal to the next related DOM element. The need for this arose because of the handling of white space in the DOM that creates text nodes between elements.

- HTML5, which provides a large number of extensions to the standard DOM. These include standardization of de facto standards such as `innerHTML`, as well as additional functionality for dealing with focus management, character sets, scrolling, and more.

The number of DOM extensions is currently small, but it's almost a certainty that the number will continue to grow as web technology continues to evolve. Browsers still experiment with proprietary extensions that, if successful, may end up as pseudo-standards or be incorporated into future versions' specifications.

12

DOM Levels 2 and 3

WHAT'S IN THIS CHAPTER?

- Changes to the DOM introduced in Levels 2 and 3
- The DOM API for manipulating styles
- Working with DOM traversal and ranges

The first level of the DOM focuses on defining the underlying structure of HTML and XML documents. DOM Levels 2 and 3 build on this structure to introduce more interactivity and support for more advanced XML features. As a result, DOM Levels 2 and 3 actually consist of several modules that, although related, describe very specific subsets of the DOM. These modules are as follows:

- **DOM Core** — Builds on the Level 1 core, adding methods and properties to nodes.
- **DOM Views** — Defines different views for a document based on stylistic information.
- **DOM Events** — Explains how to tie interactivity to DOM documents using events.
- **DOM Style** — Defines how to programmatically access and change CSS styling information.
- **DOM Traversal and Range** — Introduces new interfaces for traversing a DOM document and selecting specific parts of it.
- **DOM HTML** — Builds on the Level 1 HTML, adding properties, methods, and new interfaces.

This chapter explores each of these modules except for DOM events, which are covered fully in Chapter 13.



DOM Level 3 also contains the XPath module and the Load and Save module. These are discussed in Chapter 18.

DOM CHANGES

The purpose of the DOM Levels 2 and 3 Core is to expand the DOM API to encompass all of the requirements of XML and to provide for better error handling and feature detection. For the most part, this means supporting the concept of XML namespaces. DOM Level 2 Core doesn't introduce any new types; it simply augments the types defined in DOM Level 1 to include new methods and properties. DOM Level 3 Core further augments the existing types and introduces several new ones.

Similarly, DOM Views and HTML augment DOM interfaces, providing new properties and methods. These two modules are fairly small and so are grouped in with the Core to discuss changes to fundamental JavaScript objects. You can determine which browsers support these parts of the DOM using the following code:

```
var supportsDOM2Core = document.implementation.hasFeature("Core", "2.0");
var supportsDOM3Core = document.implementation.hasFeature("Core", "3.0");
var supportsDOM2HTML = document.implementation.hasFeature("HTML", "2.0");
var supportsDOM2Views = document.implementation.hasFeature("Views", "2.0");
var supportsDOM2XML = document.implementation.hasFeature("XML", "2.0");
```



This chapter covers only the parts of the DOM that have been implemented by browsers; parts that have yet to be implemented by any browser are not mentioned.

XML Namespaces

XML namespaces allow elements from different XML-based languages to be mixed together in a single, well-formed document without fear of element name clashes. Technically, XML namespaces are not supported by HTML but supported in XHTML; therefore, the examples in this section are in XHTML.

Namespaces are specified using the `xmlns` attribute. The namespace for XHTML is `http://www.w3.org/1999/xhtml` and should be included on the `<html>` element of any well-formed XHTML page, as shown in the following example:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Example XHTML page</title>
  </head>
  <body>
    Hello world!
  </body>
</html>
```

For this example, all elements are considered to be part of the XHTML namespace by default. You can explicitly create a prefix for an XML namespace using `xmlns`, followed by a colon, followed by the prefix, as in this example:

```
<xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <xhtml:head>
    <xhtml:title>Example XHTML page</xhtml:title>
  </xhtml:head>
  <xhtml:body>
    Hello world!
  </xhtml:body>
</xhtml:html>
```

Here, the namespace for XHTML is defined with a prefix of `xhtml`, requiring all XHTML elements to begin with that prefix. Attributes may also be namespaced to avoid confusion between languages, as shown in the following example:

```
<xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <xhtml:head>
    <xhtml:title>Example XHTML page</xhtml:title>
  </xhtml:head>
  <xhtml:body xhtml:class="home">
    Hello world!
  </xhtml:body>
</xhtml:html>
```

The `class` attribute in this example is prefixed with `xhtml`. Namespacing isn't really necessary when only one XML-based language is being used in a document; it is, however, very useful when mixing two languages together. Consider the following document containing both XHTML and SVG:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Example XHTML page</title>
  </head>
  <body>
    <svg xmlns="http://www.w3.org/2000/svg" version="1.1"
        viewBox="0 0 100 100" style="width:100%; height:100%">
      <rect x="0" y="0" width="100" height="100" style="fill:red" />
    </svg>
  </body>
</html>
```

In this example, the `<svg>` element is indicated as foreign to the containing document by setting its own namespace. All children of the `<svg>` element, as well as all attributes of the elements, are considered to be in the `http://www.w3.org/2000/svg` namespace. Even though the document is technically an XHTML document, the SVG code is considered valid because of the use of namespaces.

The interesting problem with a document such as this is what happens when a method is called on the document to interact with nodes in the document. When a new element is created, which namespace does it belong to? When querying for a specific tag name, what namespaces should be included in the results? DOM Level 2 Core answers these questions by providing namespace-specific versions of most DOM Level 1 methods.

Changes to Node

The `Node` type evolves in DOM Level 2 to include the following namespace-specific properties:

- `localName` — The node name without the namespace prefix.
- `namespaceURI` — The namespace URI of the node or `null` if not specified.
- `prefix` — The namespace prefix or `null` if not specified.

When a node uses a namespace prefix, the `nodeName` is equivalent to `prefix + ":" + localName`. Consider the following example:



```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Example XHTML page</title>
  </head>
  <body>
    <s:svg xmlns:s="http://www.w3.org/2000/svg" version="1.1"
      viewBox="0 0 100 100" style="width:100%; height:100%">
      <s:rect x="0" y="0" width="100" height="100" style="fill:red" />
    </s:svg>
  </body>
</html>
```

NamespaceExample.xml

For the `<html>` element, the `localName` and `tagName` is `"html"`, the `namespaceURI` is `"http://www.w3.org/1999/xhtml"`, and the `prefix` is `null`. For the `<s:svg>` element, the `localName` is `"svg"`, the `tagName` is `s:svg`, the `namespaceURI` is `"http://www.w3.org/2000/svg"`, and the `prefix` is `"s"`.

DOM Level 3 goes one step further and introduces the following methods to work with namespaces:

- `isDefaultNamespace(namespaceURI)` — Returns `true` when the specified `namespaceURI` is the default namespace for the node.
- `lookupNamespaceURI(prefix)` — Returns the namespace URI for the given `prefix`.
- `lookupPrefix(namespaceURI)` — Returns the prefix for the given `namespaceURI`.

In the previous example, the following code can be executed:

```
alert(document.body.isDefaultNamespace("http://www.w3.org/1999/xhtml")); //true

//assume svg contains a reference to <s:svg>
alert(svg.lookupPrefix("http://www.w3.org/2000/svg")); // "s"
alert(svg.lookupNamespaceURI("s")); // "http://www.w3.org/2000/svg"
```

These methods are primarily useful when you have a reference to a node without knowing its relationship to the rest of the document.

Changes to Document

The `Document` type is changed in DOM Level 2 to include the following namespace-specific methods:

- `createElementNS(namespaceURI, tagName)` — Creates a new element with the given `tagName` as part of the namespace indicated by `namespaceURI`.
- `createAttributeNS(namespaceURI, attributeName)` — Creates a new attribute node as part of the namespace indicated by `namespaceURI`.
- `getElementsByTagNameNS(namespaceURI, tagName)` — Returns a `NodeList` of elements with the given `tagName` that are also a part of the namespace indicated by `namespaceURI`.

These methods are used by passing in the namespace URI of the namespace to use (not the namespace prefix), as shown in the following example.

```
//create a new SVG element
var svg = document.createElementNS("http://www.w3.org/2000/svg", "svg");

//create new attribute for a random namespace
var att = document.createAttributeNS("http://www.somewhere.com", "random");

//get all XHTML elements
var elems = document.getElementsByTagNameNS("http://www.w3.org/1999/xhtml", "*");
```

The namespace-specific methods are necessary only when there are two or more namespaces in a given document.

Changes to Element

The changes to `Element` in DOM Level 2 Core are mostly related to attributes. The following new methods are introduced:

- `getAttributeNS(namespaceURI, localName)` — Gets the attribute from the namespace represented by `namespaceURI` and with a name of `localName`.
- `getAttributeNodeNS(namespaceURI, localName)` — Gets the attribute node from the namespace represented by `namespaceURI` and with a name of `localName`.
- `getElementsByTagNameNS(namespaceURI, tagName)` — Returns a `NodeList` of descendant elements with the given `tagName` that are also a part of the namespace indicated by `namespaceURI`.
- `hasAttributeNS(namespaceURI, localName)` — Determines if the element has an attribute from the namespace represented by `namespaceURI` and with a name of `localName`. Note: DOM Level 2 Core also adds a `hasAttribute()` method for use without namespaces.
- `removeAttributeNS(namespaceURI, localName)` — Removes the attribute from the namespace represented by `namespaceURI` and with a name of `localName`.

- `setAttributeNS(namespaceURI, qualifiedName, value)` — Sets the attribute from the namespace represented by `namespaceURI` and with a name of `qualifiedName` equal to `value`.
- `setAttributeNodeNS(attNode)` — Sets the attribute node from the namespace represented by `namespaceURI`.

These methods behave the same as their DOM Level 1 counterparts with the exception of the first argument, which is always the namespace URI except for `setAttributeNodeNS()`.

Changes to NamedNodeMap

The `NamedNodeMap` type also introduces the following methods for dealing with namespaces. Since attributes are represented by a `NamedNodeMap`, these methods mostly apply to attributes.

- `getNamedItemNS(namespaceURI, localName)` — Gets the item from the namespace represented by `namespaceURI` and with a name of `localName`.
- `removeNamedItemNS(namespaceURI, localName)` — Removes the item from the namespace represented by `namespaceURI` and with a name of `localName`.
- `setNamedItemNS(node)` — Adds `node`, which should have namespace information already applied.

These methods are rarely used, because attributes are typically accessed directly from an element.

Other Changes

There are some other minor changes made to various parts of the DOM in DOM Level 2 Core. These changes don't have to do with XML namespaces and are targeted more toward ensuring the robustness and completeness of the API.

Changes to DocumentType

The `DocumentType` type adds three new properties: `publicId`, `systemId`, and `internalSubset`. The `publicId` and `systemId` properties represent data that is readily available in a doctype but were inaccessible using DOM Level 1. Consider the following HTML doctype:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
```

In this doctype, the `publicId` is "`-//W3C//DTD HTML 4.01//EN`" and the `systemId` is "`http://www.w3.org/TR/html4/strict.dtd`". Browsers that support DOM Level 2 should be able to run the following JavaScript code:

```
alert(document.doctype.publicId);
alert(document.doctype.systemId);
```

Accessing this information is rarely, if ever, needed in web pages.

The `internalSubset` property accesses any additional definitions that are included in the doctype, as shown in the following example:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
[<!ELEMENT name (#PCDATA)] >
```

For this code, `document.doctype.internalSubset` returns "`<!ELEMENT name (#PCDATA)>`". Internal subsets are rarely used in HTML and are slightly more common in XML.

Changes to Document

The only new method on `Document` that is not related to namespaces is `importNode()`. The purpose of this method is to take a node from a different document and import it into a new document so that it can be added into the document structure. Remember, every node has an `ownerDocument` property that indicates the document it belongs to. If a method such as `appendChild()` is called and a node with a different `ownerDocument` is passed in, an error will occur. Calling `importNode()` on a node from a different document returns a new version of the node that is owned by the appropriate document.

The `importNode()` method is similar to the `cloneNode()` method on an element. It accepts two arguments: the node to clone and a Boolean value indicating if the child nodes should also be copied. The result is a duplicate of the node that is suitable for use in the document. Here is an example:

```
var newNode = document.importNode(oldNode, true); //import node and all children
document.body.appendChild(newNode);
```

This method isn't used very often with HTML documents; it is used more frequently with XML documents (discussed further in Chapter 18).

DOM Level 2 Views adds a property called `defaultView`, which is a pointer to the window (or frame) that owns the given document. The Views specification doesn't provide details about when other views may be available, so this is the only property added. The `defaultView` property is supported in all browsers (except Internet Explorer 8 and earlier). There is an equivalent property called `parentWindow` that is supported in Internet Explorer 8 and earlier, as well as Opera. Thus, to determine the owning window of a document, you can use the following code:

```
var parentWindow = document.defaultView || document.parentWindow;
```

Aside from this one method and property, there are a couple of changes to the `document.implementation` object specified in the DOM Level 2 Core in the form of two new methods: `createDocumentType()` and `createDocument()`. The `createDocumentType()` method is used to create new `DocumentType` nodes and accepts three arguments: the name of the doctype, the `publicId`, and the `systemId`. For example, the following code creates a new HTML 4.01 Strict doctype:

```
var doctype = document.implementation.createDocumentType("html",
"-//W3C//DTD HTML 4.01//EN",
"http://www.w3.org/TR/html4/strict.dtd");
```

An existing document's doctype cannot be changed, so `createDocumentType()` is useful only when creating new documents, which can be done with `createDocument()`. This method accepts three arguments: the `namespaceURI` for the document element, the tag name of the document element, and the doctype for the new document. A new blank XML document can be created, as shown in the following example:

```
var doc = document.implementation.createDocument("", "root", null);
```

This code creates a new document with no namespace and a document element of `<root>` with no doctype specified. To create an XHTML document, you can use the following code:

```
var doctype = document.implementation.createDocumentType("html",
    "-//W3C//DTD XHTML 1.0 Strict//EN",
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd");

var doc = document.implementation.createDocument("http://www.w3.org/1999/xhtml",
    "html", doctype);
```

Here, a new XHTML document is created with the appropriate namespace and doctype. The document has only the document element `<html>`; everything else must be added.

The DOM Level 2 HTML module also adds a method called `createHTMLDocument()` to `document.implementation`. The purpose of this method is to create a complete HTML document, including the `<html>`, `<head>`, `<title>`, and `<body>` elements. This method accepts a single argument, which is the title of the newly created document (the string to go in the `<title>` element), and returns the new HTML document as follows:



```
var htmdoc = document.implementation.createHTMLDocument("New Doc");
alert(htmdoc.title);           // "New Doc"
alert(typeof htmdoc.body);    // "object"
```

[CreateHTMLDocumentExample.htm](#)

The object created from a call to `createHTMLDocument()` is an instance of the `HTMLDocument` type and so contains all of the properties and methods associated with it, including the `title` and `body` properties. This method is supported in Internet Explorer 9+, Firefox 4+, Safari, Chrome, and Opera.

Changes to Node

The sole non-namespace-related change to the `Node` type is the addition of the `isSupported()` method. Like the `hasFeature()` method on `document.implementation` that was introduced in DOM Level 1, the `isSupported()` method indicates what the node is capable of doing. This method accepts the same two arguments: the feature name and the feature version. When the feature is implemented and is capable of being executed by the given node, `isSupported()` returns true. Here is an example:

```
if (document.body.isSupported("HTML", "2.0")){
    //do something only possible using DOM Level 2 HTML
}
```

This method is of limited usefulness and falls victim to the same issues surrounding `hasFeature()` in that implementations get to decide whether to return `true` or `false` for each feature. Capability detection is a better approach for detecting whether or not a particular feature is available.

DOM Level 3 introduces two methods to help compare nodes: `isSameNode()` and `isEqualNode()`. Both methods accept a single node as an argument and return `true` if that node is the same as or equal to the reference node. Two nodes are the same when they reference the same object. Two nodes are equal when they are of the same type and have properties that are equal (`nodeName`, `nodeValue`, and so on), and their `attributes` and `childNodes` properties are equivalent (containing equivalent values in the same positions). Here is an example:

```
var div1 = document.createElement("div");
div1.setAttribute("class", "box");

var div2 = document.createElement("div");
div2.setAttribute("class", "box");

alert(div1.isSameNode(div1));    //true
alert(div1.isEqualNode(div2));  //true
alert(div1.isSameNode(div2));  //false
```

Here, two `<div>` elements are created with the same attributes. The two elements are equivalent to one another but are not the same.

DOM Level 3 also introduces methods for attaching additional data to DOM nodes. The `setUserData()` method assigns data to a node and accepts three arguments: the key to set, the actual data (which may be of any data type), and a handler function. You can assign data to a node using the following code:

```
document.body.setUserData("name", "Nicholas", function(){});
```

You can then retrieve the information using `getUserData()` and passing in the same key, as shown here:

```
var value = document.body.getUserData("name");
```

The handler function for `setUserData()` is called whenever the node with the data is cloned, removed, renamed, or imported into another document and gives you the opportunity to determine what should happen to the user data in each of those cases. The handler function accepts five arguments: a number indicating the type of operation (1 for clone, 2 for import, 3 for delete, or 4 for rename), the data key, the data value, the source node, and the destination node. The source node is `null` when the node is being deleted, and the destination node is `null` unless the node is being cloned. You can then determine how to store the data. Here is an example:



```
var div = document.createElement("div");
div.setUserData("name", "Nicholas", function(operation, key, value, src, dest){
    if (operation == 1){
        dest.setUserData(key, value, function(){});    }
});

var newDiv = div.cloneNode(true);
alert(newDiv.getUserData("name"));    //"Nicholas"
```

Here, a `<div>` element is created and has some data assigned to it, including some user data. When the element is cloned via `cloneNode()`, the handler function is called and the data is automatically assigned to the clone. When `getUserData()` is called on the clone, it returns the same value that was assigned to the original.

Changes to Frames

Frames and iframes, represented by `HTMLFrameElement` and `HTMLIFrameElement`, respectively, have a new property in DOM Level 2 HTML called `contentDocument`. This property contains a pointer to the `document` object representing the contents of the frame. Prior to this, there was no way to retrieve the `document` object directly through the element; it was necessary to use the `frames` collection. This property can be used, as shown in the following example:



```
var iframe = document.getElementById("myIframe");
var iframeDoc = iframe.contentDocument;    //won't work in IE < 8
```

[IFrameElementExample.htm](#)

The `contentDocument` property is an instance of `Document` and can be used just like any other HTML document, including all properties and methods. This property is supported in Opera, Firefox, Safari, and Chrome. Internet Explorer versions prior to 8 don't support `contentDocument` on frames but do support a property called `contentWindow` that returns the `window` object for the frame, which has a `document` property. So, to access the `document` object for an iframe in all four browsers, you can use the following code:

```
var iframe = document.getElementById("myIframe");
var iframeDoc = iframe.contentDocument || iframe.contentWindow.document;
```

[IFrameElementExample2.htm](#)

The `contentWindow` property is available in all browsers.



Access to the `document` object of a frame or iframe is limited based on cross-domain security restrictions. If you are attempting to access the `document` object of a frame containing a page that is loaded from a different domain or subdomain, or with a different protocol, doing so will throw an error.

STYLES

Styles are defined in HTML in three ways: including an external style sheet via the `<link>` element, defining inline styles using the `<style>` element, and defining element-specific styles using the `style` attribute. DOM Level 2 Styles provides an API around all three of these styling mechanisms. You can determine if the browser supports the DOM Level 2 CSS capabilities using the following code:

```
var supportsDOM2CSS = document.implementation.hasFeature("CSS", "2.0");
var supportsDOM2CSS2 = document.implementation.hasFeature("CSS2", "2.0");
```

Accessing Element Styles

Any HTML element that supports the `style` attribute also has a `style` property exposed in JavaScript. The `style` object is an instance of `CSSStyleDeclaration` and contains all stylistic information specified by the HTML `style` attribute but no information about styles that have cascaded from either included or inline style sheets. Any CSS property specified in the `style` attribute are represented as properties on the `style` object. Since CSS property names use dash case (using dashes to separate words, such as `background-image`), the names must be converted into camel case in order to be used in JavaScript. The following table lists some common CSS properties and the equivalent property names on the `style` object.

CSS PROPERTY	JAVASCRIPT PROPERTY
<code>background-image</code>	<code>style.backgroundImage</code>
<code>color</code>	<code>style.color</code>
<code>display</code>	<code>style.display</code>
<code>font-family</code>	<code>style.fontFamily</code>

For the most part, property names convert directly simply by changing the format of the property name. The one CSS property that doesn't translate directly is `float`. Since `float` is a reserved word in JavaScript, it can't be used as a property name. The DOM Level 2 Style specification states that the corresponding property on the `style` object should be `cssFloat`, which is supported in Internet Explorer 9, Firefox, Safari, Opera, and Chrome. Internet Explorer 8 and earlier use `styleFloat` instead.

Styles can be set using JavaScript at any time so long as a valid DOM element reference is available. Here are some examples:

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

//set the background color
myDiv.style.backgroundColor = "red";

//change the dimensions
myDiv.style.width = "100px";
myDiv.style.height = "200px";

//assign a border
myDiv.style.border = "1px solid black";
```

When styles are changed in this manner, the `display` of the element is automatically updated.



When in standards mode, all measurements have to include a unit of measure. In quirks mode, you can set style.width to be "20" and it will assume that you mean "20px"; in standards mode, setting style.width to "20" will be ignored because it has no unit of measurement. In practice, it's best to always include the unit of measurement.

Styles specified in the `style` attribute can also be retrieved using the `style` object. Consider the following HTML:

```
<div id="myDiv" style="background-color: blue; width: 10px; height: 25px"></div>
```

The information from this element's `style` attribute can be retrieved via the following code:

```
alert(myDiv.style.backgroundColor); // "blue"
alert(myDiv.style.width); // "10px"
alert(myDiv.style.height); // "25px"
```

If no `style` attribute is specified on an element, the `style` object may contain some default values but cannot give any accurate information about the styling of the element.

DOM Style Properties and Methods

The DOM Level 2 Style specification also defines several properties and methods on the `style` object. These properties and methods provide information about the contents of the element's `style` attribute and enabling changes. They are as follows:

- `cssText` — As described previously, provides access to the CSS code of the `style` attribute.
- `length` — The number of CSS properties applied to the element.
- `parentRule` — The `CSSRule` object representing the CSS information. The `CSSRule` type is discussed in a later section.
- `getPropertyCSSValue(propertyName)` — Returns a `CSSValue` object containing the value of the given property.
- `getPropertyPriority(propertyName)` — Returns "important" if the given property is set using `!important`; otherwise it returns an empty string.
- `getPropertyValue(propertyName)` — Returns the string value of the given property.
- `item(index)` — Returns the name of the CSS property at the given position.
- `removeProperty(propertyName)` — Removes the given property from the style.
- `setProperty(propertyName, value, priority)` — Sets the given property to the given value with a priority (either "important" or an empty string).

The `cssText` property allows access to the CSS code of the style. When used in read mode, `cssText` returns the browser's internal representation of the CSS code in the `style` attribute. When used

in write mode, the value assigned to `cssText` overwrites the entire value of the `style` attribute, meaning that all previous style information specified using the attribute is lost. For instance, if the element has a border specified via the `style` attribute and you overwrite `cssText` with rules that don't include the border, it is removed from the element. The `cssText` property is used as follows:

```
myDiv.style.cssText = "width: 25px; height: 100px; background-color: green";
alert(myDiv.style.cssText);
```

Setting the `cssText` property is the fastest way to make multiple changes to an element's style because all of the changes are applied at once.

The `length` property is designed for use in conjunction with the `item()` method for iterating over the CSS properties defined on an element. With these, the `style` object effectively becomes a collection, and bracket notation can be used in place of `item()` to retrieve the CSS property name in the given position, as shown in the following example:

```
for (var i=0, len=myDiv.style.length; i < len; i++){
    alert(myDiv.style[i]);      //or myDiv.style.item(i)
}
```

Using either bracket notation or `item()`, you can retrieve the CSS property name ("background-color", not "backgroundColor"). This property name can then be used in `getPropertyValue()` to retrieve the actual value of the property, as shown in the following example:

```
var prop, value, i, len;
for (i=0, len=myDiv.style.length; i < len; i++){
    prop = myDiv.style[i];      //or myDiv.style.item(i)
    value = myDiv.style.getPropertyValue(prop);
    alert(prop + " : " + value);
}
```

The `getPropertyValue()` method always retrieves the string representation of the CSS property value. If you need more information, `getPropertyCSSValue()` returns a `CSSValue` object that has two properties: `cssText` and `cssValueType`. The `cssText` property is the same as the value returned from `getPropertyValue()`. The `cssValueType` property is a numeric constant indicating the type of value being represented: 0 for an inherited value, 1 for a primitive value, 2 for a list, or 3 for a custom value. The following code outputs the CSS property value and the value type:



Available for
download on
Wrox.com

```
var prop, value, i, len;
for (i=0, len=myDiv.style.length; i < len; i++){
    prop = myDiv.style[i];      //or myDiv.style.item(i)
    value = myDiv.style.getPropertyCSSValue(prop);
    alert(prop + " : " + value.cssText + " (" + value.cssValueType + ")");
}
```

[DOMStyleObjectExample.htm](#)

In practice, `getPropertyCSSValue()` is less useful than `getPropertyValue()`. This method is supported in Internet Explorer 9+, Safari 3+, and Chrome. Firefox through version 7 provides the method, but calls always return `null`.

The `removeProperty()` method is used to remove a specific CSS property from the element's styling. Removing a property using this method means that any default styling for that property (cascading from other style sheets) will be applied. For instance, to remove a `border` property that was set in the `style` attribute, you can use the following code:

```
myDiv.style.removeProperty("border");
```

This method is helpful when you're not sure what the default value for a given CSS property is. Simply removing the property allows the default value to be used.



Unless otherwise noted, the properties and methods in this section are supported in Internet Explorer 9+, Firefox, Safari, Opera 9+, and Chrome.

Computed Styles

The `style` object offers information about the `style` attribute on any element that supports it but contains no information about the styles that have cascaded from style sheets and affect the element. DOM Level 2 Style augments `document.defaultView` to provide a method called `getComputedStyle()`. This method accepts two arguments: the element to get the computed style for and a pseudo-element string (such as "`:after`"). The second argument can be `null` if no pseudo-element information is necessary. The `getComputedStyle()` method returns a `CSSStyleDeclaration` object (the same type as the `style` property) containing all computed styles for the element. Consider the following HTML page:



Available for download on
Wrox.com

```
<!DOCTYPE html>
<html>
<head>
    <title>Computed Styles Example</title>
    <style type="text/css">
        #myDiv {
            background-color: blue;
            width: 100px;
            height: 200px;
        }
    </style>
</head>
<body>
    <div id="myDiv" style="background-color: red; border: 1px solid black"></div>
</body>
</html>
```

[ComputedStylesExample.htm](#)

In this example, the `<div>` element has styles applied to it both from an inline style sheet (the `<style>` element) and from the `style` attribute. The `style` object has values for `backgroundColor`

and border, but nothing for width and height, which are applied through a style sheet rule. The following code retrieves the computed style for the element:



```
var myDiv = document.getElementById("myDiv");
var computedStyle = document.defaultView.getComputedStyle(myDiv, null);

alert(computedStyle.backgroundColor);    // "red"
alert(computedStyle.width);             // "100px"
alert(computedStyle.height);            // "200px"
alert(computedStyle.border);           // "1px solid black" in some browsers
```

[ComputedStylesExample.htm](#)

When retrieving the computed style of this element, the background color is reported as "red", the width as "100px", and the height as "200px". Note that the background color is not "blue", because that style is overridden on the element itself. The border property may or may not return the exact border rule from the style sheet (Opera returns it, but other browsers do not). This inconsistency is due to the way that browsers interpret rollup properties, such as border, that actually set a number of other properties. When you set border, you're actually setting rules for the border width, color, and style on all four borders (border-left-width, border-top-color, border-bottom-style, and so on). So even though computedStyle.border may not return a value in all browsers, computedStyle.borderLeftWidth does.



Note that although some browsers support this functionality, the manner in which values are represented can differ. For example, Firefox and Safari translate all colors into RGB form (such as rgb(255, 0, 0) for red), whereas Opera translates all colors into their hexadecimal representations (#ff0000 for red). It's always best to test your functionality on a number of browsers when using getComputedStyle().

Internet Explorer doesn't support getComputedStyle(), though it has a similar concept. Every element that has a style property also has a currentStyle property. The currentStyle property is an instance of CSSStyleDeclaration and contains all of the final computed styles for the element. The styles can be retrieved in a similar fashion, as shown in this example:

```
var myDiv = document.getElementById("myDiv");
var computedStyle = myDiv.currentStyle;

alert(computedStyle.backgroundColor);    // "red"
alert(computedStyle.width);             // "100px"
alert(computedStyle.height);            // "200px"
alert(computedStyle.border);           // undefined
```

[IEComputedStylesExample.htm](#)

As with the DOM version, the border style is not returned in Internet Explorer because it is considered a rollup property.

The important thing to remember about computed styles in all browsers is that they are read-only; you cannot change CSS properties on a computed style object. Also, the computed style contains styling information that is part of the browser's internal style sheet, so any CSS property that has a default value will be represented in the computed style. For instance, the `visibility` property always has a default value in all browsers, but this value differs per implementation. Some browsers set the `visibility` property to "visible" by default, whereas others have it as "inherit". You cannot depend on the default value of a CSS property to be the same across browsers. If you need elements to have a specific default value, you should manually specify it in a style sheet.

Working with Style Sheets

The `CSSStyleSheet` type represents a CSS style sheet as included using a `<link>` element or defined in a `<style>` element. Note that the elements themselves are represented by the `HTMLLinkElement` and `HTMLElement` types, respectively. The `CSSStyleSheet` type is generic enough to represent a style sheet no matter how it is defined in HTML. Furthermore, the element-specific types allow for modification of HTML attributes, whereas a `CSSStyleSheet` object is, with the exception of one property, a read-only interface. You can determine if the browser supports the DOM Level 2 style sheets using the following code:

```
var supportsDOM2StyleSheets =
    document.implementation.hasFeature("StyleSheets", "2.0");
```

The `CSSStyleSheet` type inherits from `StyleSheet`, which can be used as a base to define non-CSS style sheets. The following properties are inherited from `StyleSheet`:

- `disabled` — A Boolean value indicating if the style sheet is disabled. This property is read/write, so setting its value to `true` will disable a style sheet.
- `href` — The URL of the style sheet if it is included using `<link>`; otherwise, this is `null`.
- `media` — A collection of media types supported by this style sheet. The collection has a `length` property and `item()` method, as with all DOM collections. Like other DOM collections, you can use bracket notation to access specific items in the collection. An empty list indicates that the style sheet should be used for all media. In Internet Explorer 8 and earlier, `media` is a string reflecting the `media` attribute of the `<link>` or `<style>` element.
- `ownerNode` — Pointer to the node that owns the style sheet, which is either a `<link>` or a `<style>` element in HTML (it can be a processing instruction in XML). This property is `null` if a style sheet is included in another style sheet using `@import`. Internet Explorer 8 and earlier do not support this property.
- `parentStyleSheet` — When a style sheet is included via `@import`, this is a pointer to the style sheet that imported it.
- `title` — The value of the `title` attribute on the `ownerNode`.
- `type` — A string indicating the type of style sheet. For CSS style sheets, this is "`text/css`".

With the exception of `disabled`, the rest of these properties are read-only. The `CSSStyleSheet` type supports all of these properties and the following properties and methods:

- `cssRules` — A collection of rules contained in the style sheet. Internet Explorer 8 and earlier don't support this property but have a comparable property called `rules`. Internet Explorer 9 supports both `cssRules` and `rules`.
- `ownerRule` — If the style sheet was included using `@import`, this is a pointer to the rule representing the import; otherwise, this is `null`. Internet Explorer does not support this property.
- `deleteRule(index)` — Deletes the rule at the given location in the `cssRules` collection. Internet Explorer 8 and earlier does not support this method, but it does have a similar method called `removeRule()`. Internet Explorer 9 supports both `deleteRule()` and `removeRule()`.
- `insertRule(rule, index)` — Inserts the given string rule at the position specified in the `cssRules` collection. Internet Explorer 8 and earlier do not support this method but have a similar method called `addRule()`. Internet Explorer 9 supports both `insertRule()` and `addRule()`.

The list of style sheets available on the document is represented by the `document.styleSheets` collection. The number of style sheets on the document can be retrieved using the `length` property, and each individual style sheet can be accessed using either the `item()` method or bracket notation. Here is an example:



```
var sheet = null;
for (var i=0, len=document.styleSheets.length; i < len; i++) {
    sheet = document.styleSheets[i];
    alert(sheet.href);
}
```

[StyleSheetsExample.htm](#)

This code outputs the `href` property of each style sheet used in the document (`<style>` elements have no `href`).

The style sheets returned in `document.styleSheets` vary from browser to browser. All browsers include `<style>` elements and `<link>` elements with `rel` set to "stylesheet". Internet Explorer and Opera also include `<link>` elements where `rel` is set to "alternate stylesheet".

It's also possible to retrieve the `CSSStyleSheet` object directly from the `<link>` or `<style>` element. The DOM specifies a property called `sheet` that contains the `CSSStyleSheet` object, which all browsers except Internet Explorer support. Internet Explorer supports a property called `styleSheet` that does the same thing. To retrieve the `style sheet` object across browsers, you can use the following code:

```
function getStyleSheet(element){
    return element.sheet || element.styleSheet;
}

//get the style sheet for the first <link/> element
```

```
var link = document.getElementsByTagName("link")[0];
var sheet = getStyleSheet(link);
```

StyleSheetsExample2.htm

The object returned from `getStyleSheet()` is the same object that exists in the `document.styleSheets` collection.

CSS Rules

A `CSSRule` object represents each rule in a style sheet. The `CSSRule` type is actually a base type from which several other types inherit, but the most often used is `CSSStyleRule`, which represents styling information (other rules include `@import`, `@font-face`, `@page`, and `@charset`, although these rules rarely need to be accessed from script). The following properties are available on a `CSSStyleRule` object:

- `cssText` — Returns the text for the entire rule. This text may be different from the actual text in the style sheet because of the way that browsers handle style sheets internally; Safari always converts everything to all lowercase. This property is not supported in Internet Explorer.
- `parentRule` — If this rule is imported, this is the import rule; otherwise, this is `null`. This property is not supported in Internet Explorer.
- `parentStyleSheet` — The style sheet that this rule is a part of. This property is not supported in Internet Explorer.
- `selectorText` — Returns the selector text for the rule. This text may be different from the actual text in the style sheet because of the way that browsers handle style sheets internally. This property is read-only in Firefox, Safari, Chrome, and Internet Explorer (where it throws an error). Opera allows `selectorText` to be changed.
- `style` — A `CSSStyleDeclaration` object that allows the setting and getting of specific style values for the rule.
- `type` — A constant indicating the type of rule. For style rules, this is always 1. This property is not supported in Internet Explorer.

The three most frequently used properties are `cssText`, `selectorText`, and `style`. The `cssText` property is similar to the `style.cssText` property but not exactly the same. The former includes the selector text and the braces around the style information; the latter contains only the style information (similar to `style.cssText` on an element). Also, `cssText` is read-only, whereas `style.cssText` may be overwritten.

Most of the time, the `style` property is all that is required to manipulate style rules. This object can be used just like the one on each element to read or change the style information for a rule. Consider the following CSS rule:



```
div.box {
    background-color: blue;
    width: 100px;
```

```
        height: 200px;
    }
```

[CSSRulesExample.htm](#)

Assuming that this rule is in the first style sheet on the page and is the only style in that style sheet, the following code can be used to retrieve all of its information:



Available for
download on
Wrox.com

```
var sheet = document.styleSheets[0];
var rules = sheet.cssRules || sheet.rules;           //get rules list
var rule = rules[0];                                //get first rule
alert(rule.selectorText);                          //"div.box"
alert(rule.style.cssText);                         //complete CSS code
alert(rule.style.backgroundColor);                 //"blue"
alert(rule.style.width);                           //"100px"
alert(rule.style.height);                          //"200px"
```

[CSSRulesExample.htm](#)

Using this technique, it's possible to determine the style information related to a rule in the same way you can determine the inline style information for an element. As with elements, it's also possible to change the style information, as shown in the following example:

```
var sheet = document.styleSheets[0];
var rules = sheet.cssRules || sheet.rules;           //get rules list
var rule = rules[0];                                //get first rule
rule.style.backgroundColor = "red"
```

[CSSRulesExample.htm](#)

Note that changing a rule in this way affects all elements on the page for which the rule applies. If there are two `<div>` elements that have the `box` class, they will both be affected by this change.

Creating Rules

The DOM states that new rules are added to existing style sheets using the `insertRule()` method. This method expects two arguments: the text of the rule and the index at which to insert the rule. Here is an example:

```
sheet.insertRule("body { background-color: silver }", 0);      //DOM method
```

This example inserts a rule that changes the document's background color. The rule is inserted as the first rule in the style sheet (position 0) — the order is important in determining how the rule cascades into the document. The `insertRule()` method is supported in Internet Explorer 9+ and all modern versions of Firefox, Safari, Opera, and Chrome.

Internet Explorer 8 and earlier have a similar method called `addRule()` that expects two arguments: the selector text and the CSS style information. An optional third argument indicates

the position in which to insert the rule. The Internet Explorer equivalent of the previous example is as follows:

```
sheet.addRule("body", "background-color: silver", 0);           //IE only
```

The documentation for this method indicates that you can add up to 4,095 style rules using `addRule()`. Any additional calls result in an error.

To add a rule to a style sheet in a cross-browser way, you can use the following method. It accepts four arguments: the style sheet to add to followed by the same three arguments as `addRule()`, as shown in the following example:



```
function insertRule(sheet, selectorText, cssText, position){  
    if (sheet.insertRule){  
        sheet.insertRule(selectorText + "{" + cssText + "}", position);  
    } else if (sheet.addRule){  
        sheet.addRule(selectorText, cssText, position);  
    }  
}
```

[CSSRulesExample2.htm](#)

This function can then be called in the following way:

```
insertRule(document.styleSheets[0], "body", "background-color: silver", 0);
```

Although adding rules in this way is possible, it quickly becomes burdensome when the number of rules to add is large. In that case, it's better to use the dynamic style loading technique discussed in Chapter 10.

Deleting Rules

The DOM method for deleting rules from a style sheet is `deleteRule()`, which accepts a single argument: the index of the rule to remove. To remove the first rule in a style sheet, you can use the following code:

```
sheet.deleteRule(0);      //DOM method
```

Internet Explorer 8 and earlier support a method called `removeRule()` that is used in the same way, as shown here:

```
sheet.removeRule(0);      //IE only
```

The following function handles deleting a rule in a cross-browser way. The first argument is the style sheet to act on and the second is the index to delete, as shown in the following example:

```
function deleteRule(sheet, index){  
    if (sheet.deleteRule){  
        sheet.deleteRule(index);  
    } else if (sheet.removeRule){  
        sheet.removeRule(index);  
    }  
}
```

```

        sheet.removeRule(index);
    }
}

```

[CSSRulesExample2.htm](#)

This function can be used as follows:

```
deleteRule(document.styleSheets[0], 0);
```

As with adding rules, deleting rules is not a common practice in web development and should be used carefully, because the cascading effect of CSS can be affected.

Element Dimensions

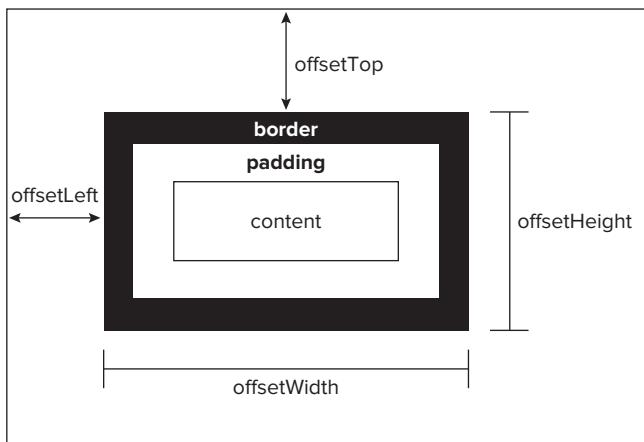
The following properties and methods are not part of the DOM Level 2 Style specification but nonetheless related to styles on HTML elements. The DOM stops short of describing ways to determine the actual dimensions of elements on a page. Internet Explorer first introduced several properties to expose dimension information to developers. These properties have now been incorporated into all of the major browsers.

Offset Dimensions

The first set of properties deals with **offset dimensions**, which incorporate all of the visual space that an element takes up on the screen. An element's visual space on the page is made up of its height and width, including all padding, scrollbars, and borders (but not including margins). The following four properties are used to retrieve offset dimensions:

- `offsetHeight` — The amount of vertical space, in pixels, taken up by the element, including its height, the height of a horizontal scrollbar (if visible), the top border height, and the bottom border height.
- `offsetLeft` — The number of pixels between the element's outside left border and the containing element's inside left border.
- `offsetTop` — The number of pixels between the element's outside top border and the containing element's inside top border.
- `offsetWidth` — The amount of horizontal space taken up by the element, including its width, the width of a vertical scrollbar (if visible), the left border width, and the right border width.

The `offsetLeft` and `offsetTop` properties are in relation to the containing element, which is stored in the `offsetParent` property. The `offsetParent` may not necessarily be the same as the `parentNode`. For example, the `offsetParent` of a `<td>` element is the `<table>` element that it's an ancestor of, because the `<table>` is the first element in the hierarchy that provides dimensions. Figure 12-1 illustrates the various dimensions these properties represent.

offsetParent**FIGURE 12-1**

The offset of an element on the page can roughly be determined by taking the `offsetLeft` and `offsetTop` properties and adding them to the same properties of the `offsetParent`, continuing up the hierarchy until you reach the root element. Here is an example:



```

function getElementLeft(element){
    var actualLeft = element.offsetLeft;
    var current = element.offsetParent;

    while (current !== null){
        actualLeft += current.offsetLeft;
        current = current.offsetParent;
    }

    return actualLeft;
}

function getElementTop(element){
    var actualTop = element.offsetTop;
    var current = element.offsetParent;

    while (current !== null){
        actualTop += current.offsetTop;
        current = current.offsetParent;
    }

    return actualTop;
}

```

[OffsetDimensionsExample.htm](#)

These two functions climb through the DOM hierarchy using the `offsetParent` property, adding up the offset properties at each level. For simple page layouts using CSS-based layouts, these functions are very accurate. For page layouts using tables and iframes, the values returned are less accurate on a cross-browser basis because of the different ways that these elements are implemented. Generally, all elements that are contained solely within `<div>` elements have `<body>` as their `offsetParent`, so `getBoundingClientRect()` and `getOffsetTop()` will return the same values as `offsetLeft` and `offsetTop`.



All of the offset dimension properties are read-only and are calculated each time they are accessed. Therefore, you should try to avoid making multiple calls to any of these properties; instead, store the values you need in local variables to avoid incurring a performance penalty.

Client Dimensions

The client dimensions of an element comprise the space occupied by the element's content and its padding. There are only two properties related to client dimensions: `clientWidth` and `clientHeight`. The `clientWidth` property is the width of the content area plus the width of both the left and the right padding. The `clientHeight` property is the height of the content area plus the height of both the top and the bottom padding. Figure 12-2 illustrates these properties.

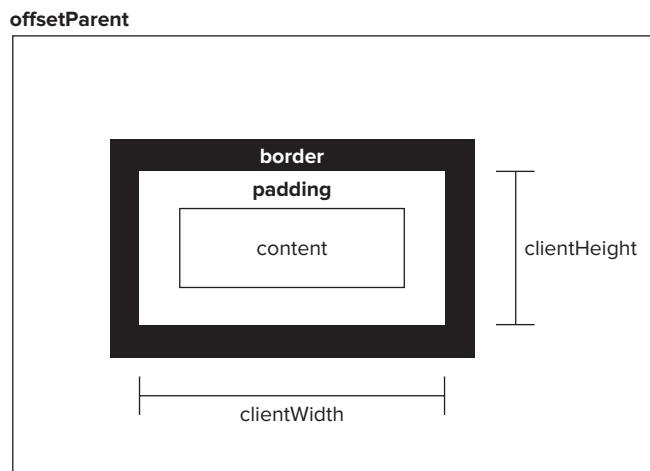


FIGURE 12-2

The client dimensions are literally the amount of space inside of the element, so the space taken up by scrollbars is not counted. The most common use of these properties is to determine the browser viewport size, as discussed in Chapter 8. This is done by using the `clientWidth` and

`clientHeight` of `document.documentElement` or `document.body` (in Internet Explorer 6 and earlier), as shown in the following example:

```
function getViewport(){
    if (document.compatMode == "BackCompat") {
        return {
            width: document.body.clientWidth,
            height: document.body.clientHeight
        };
    } else {
        return {
            width: document.documentElement.clientWidth,
            height: document.documentElement.clientHeight
        };
    }
}
```

This function determines whether or not the browser is running in quirks mode by checking the `document.compatMode` property. Internet Explorer 8+, Chrome, Safari, Opera, and Firefox run in standards mode most of the time, so they will also continue to the `else` statement. The function returns an object with two properties: `width` and `height`. These represent the dimensions of the viewport (the `<html>` or `<body>` elements).



As with offset dimensions, client dimensions are read-only and are calculated each time they are accessed.

Scroll Dimensions

The last set of dimensions is **scroll dimensions**, which provide information about an element whose content is scrolling. Some elements, such as the `<html>` element, scroll automatically without needing any additional code, whereas other elements can be made to scroll by using the CSS `overflow` property. The four scroll dimension properties are as follows:

- `scrollHeight` — The total height of the content if there were no scrollbars present.
- `scrollLeft` — The number of pixels that are hidden to the left of the content area. This property can be set to change the scroll position of the element.
- `scrollTop` — The number of pixels that are hidden in the top of the content area. This property can be set to change the scroll position of the element.
- `scrollWidth` — The total width of the content if there were no scrollbars present.

Figure 12-3 illustrates these properties.

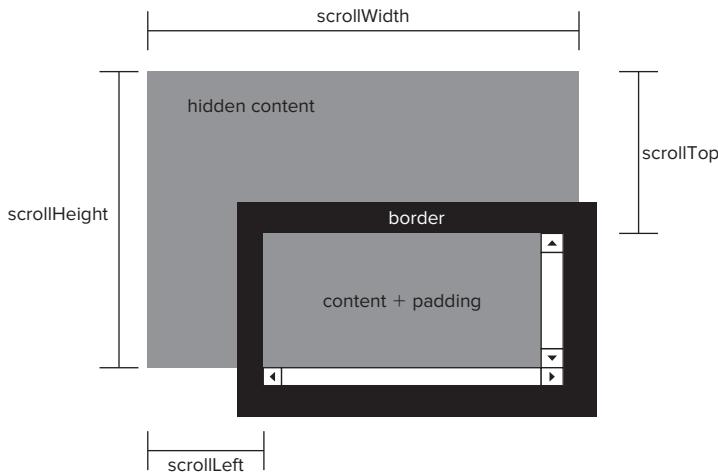


FIGURE 12-3

The `scrollWidth` and `scrollHeight` properties are useful for determining the actual dimensions of the content in a given element. For example, the `<html>` element is considered the element that scrolls the viewport in a web browser (in Internet Explorer 5.5 and earlier, the `<body>` element is the element that scrolls). Therefore, the height of an entire page that has a vertical scrollbar is `document.documentElement.scrollHeight`.

The relationship between `scrollWidth` and `scrollHeight` to `clientWidth` and `clientHeight` is not clear when it comes to documents that do not scroll. Inspecting these properties on `document.documentElement` leads to inconsistent results across browsers, as described here:

- Firefox keeps the properties equal, but the size is related to the actual size of the document content, not the size of the viewport.
- Opera, Safari 3.1 and later, and Chrome keep the properties different, with `scrollwidth` and `scrollHeight` equal to the size of the viewport and `clientWidth` and `clientHeight` equal to the document content.
- Internet Explorer (in standards mode) keeps the properties different, with `scrollWidth` and `scrollHeight` equal to the size of the document content, and `clientWidth` and `clientHeight` equal to the viewport size.

When trying to determine the total height of a document, including the minimum height based on the viewport, you must take the maximum value of `scrollWidth/clientWidth` and `scrollHeight/clientHeight` to guarantee accurate results across browsers. Here is an example:

```
var docHeight = Math.max(document.documentElement.scrollHeight,
                        document.documentElement.clientHeight);

var docWidth = Math.max(document.documentElement.scrollWidth,
                       document.documentElement.clientWidth);
```

Note that for Internet Explorer in quirks mode, you'll need to use the same measurements on `document.body` instead of `document.documentElement`.

The `scrollLeft` and `scrollTop` properties can be used either to determine the current scroll settings on an element or to set them. When an element hasn't been scrolled, both properties are equal to 0. If the element has been scrolled vertically, `scrollTop` is greater than 0, indicating the amount of content that is not visible at the top of the element. If the element has been scrolled horizontally, `scrollLeft` is greater than 0, indicating the number of pixels that are not visible on the left. Since each property can also be set, you can reset the element's scroll position by setting both `scrollLeft` and `scrollTop` to 0. The following function checks to see if the element is at the top, and if not, it scrolls it back to the top:

```
function scrollToTop(element) {
    if (element.scrollTop != 0) {
        element.scrollTop = 0;
    }
}
```

This function uses `scrollTop` both for retrieving the value and for setting it.

Determining Element Dimensions

Internet Explorer, Firefox 3+, Safari 4+, Opera 9.5+, and Chrome offer a method called `getBoundingClientRect()` on each element, which returns a rectangle object that has four properties: `left`, `top`, `right`, and `bottom`. These properties give the location of the element on the page relative to the viewport. The browser implementations are slightly different. Internet Explorer 8 and earlier consider the upper-left corner of the document to be located at (2,2), whereas the other implementations, including Internet Explorer 9, use the traditional (0,0) as the starting coordinates. This necessitates doing an initial check for the location of an element positioned at (0,0), which will return (2,2) in Internet Explorer 8 and earlier and (0,0) in other browsers. Here is an example:



```
function getBoundingClientRect(element) {
    if (typeof arguments.callee.offset != "number") {
        var scrollTop = document.documentElement.scrollTop;
        var temp = document.createElement("div");
        temp.style.cssText = "position:absolute;left:0;top:0;";
        document.body.appendChild(temp);
        arguments.callee.offset = -temp.getBoundingClientRect().top - scrollTop;
        document.body.removeChild(temp);
        temp = null;
    }

    var rect = element.getBoundingClientRect();
    var offset = arguments.callee.offset;

    return {
        left: rect.left + offset,
        right: rect.right + offset,
        top: rect.top + offset,
        bottom: rect.bottom + offset
    };
}
```

[GetBoundingClientRectExample.htm](#)

This function uses a property on itself to determine the necessary adjustment for the coordinates. The first step is to see if the property is defined and, if not, define it. The `offset` is defined as the negative value of a new element's top coordinate, essentially setting it to `-2` in Internet Explorer and `-0` in Firefox and Opera. To figure this out, you are required to create a temporary element, set its position to `(0,0)`, and then call `getBoundingClientRect()`. The `scrollTop` of the viewport is subtracted from this value just in case the window has already been scrolled when the method is called. Using this construct ensures that you don't have to call `getBoundingClientRect()` twice each time this function is called. Then, the method is called on the element and an object is created with the new calculations.

For browsers that don't support `getBoundingClientRect()`, the same information can be gained by using other means. Generally, the difference between the `right` and the `left` properties is equivalent to `offsetWidth`, and the difference between the `bottom` and the `top` properties is equivalent to `offsetHeight`. Furthermore, the `left` and `top` properties are roughly equivalent to using the `getElementLeft()` and `getElementTop()` functions defined earlier in this chapter. A cross-browser implementation of the function can be created, as shown in the following example:



Available for
download on
Wrox.com

```
function getBoundingClientRect(element) {
    var scrollTop = document.documentElement.scrollTop;
    var scrollLeft = document.documentElement.scrollLeft;

    if (element.getBoundingClientRect) {
        if (typeof arguments.callee.offset != "number") {
            var temp = document.createElement("div");
            temp.style.cssText = "position:absolute;left:0;top:0;";
            document.body.appendChild(temp);
            arguments.callee.offset = -temp.getBoundingClientRect().top -
                scrollTop;
            document.body.removeChild(temp);
            temp = null;
        }
    }

    var rect = element.getBoundingClientRect();
    var offset = arguments.callee.offset;

    return {
        left: rect.left + offset,
        right: rect.right + offset,
        top: rect.top + offset,
        bottom: rect.bottom + offset
    };
} else {

    var actualLeft = getElementLeft(element);
    var actualTop = getElementTop(element);

    return {
        left: actualLeft - scrollLeft,
        right: actualLeft + element.offsetWidth - scrollLeft,
        top: actualTop - scrollTop,
        bottom: actualTop + element.offsetHeight - scrollTop
    };
}
```

```
        top: actualTop - scrollTop,
        bottom: actualTop + element.offsetHeight - scrollTop
    }
}
```

[GetBoundingClientRectExample.htm](#)

This function uses the native `getBoundingClientRect()` method when it's available and defaults to calculating the dimensions when it is not. There are some instances where the values will vary in browsers, such as with layouts that use tables or scrolling elements.



Because of the use of arguments.callee, this method will not work in strict mode.

TRAVERSALS

The DOM Level 2 Traversal and Range module defines two types that aid in sequential traversal of a DOM structure. These types, `NodeIterator` and `Treewalker`, perform depth-first traversals of a DOM structure given a certain starting point. These object types are available in DOM-compliant browsers, including Internet Explorer 9+, Firefox, Safari, Opera, and Chrome. There is no support for DOM traversals in Internet Explorer 8 and earlier. You can test for DOM Level 2 Traversal support using the following code:

```
var supportsTraversals = document.implementation.hasFeature("Traversal", "2.0");
var supportsNodeIterator = (typeof document.createNodeIterator == "function");
var supportsTreeWalker = (typeof document.createTreeWalker == "function");
```

As stated previously, DOM traversals are a depth-first traversal of the DOM structure that allows movement in at least two directions (depending on the type being used). A traversal is rooted at a given node, and it cannot go any further up the DOM tree than that root. Consider the following HTML page:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example</title>
    </head>
    <body>
        <p><b>Hello</b> world!</p>
    </body>
</html>
```

This page evaluates to the DOM tree represented in Figure 12-4.

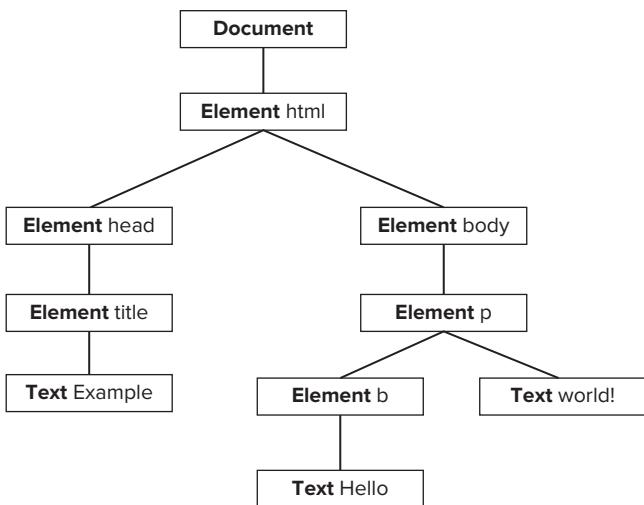


FIGURE 12-4

Any node can be the root of the traversals. Suppose, for example, that the `<body>` element is the traversal root. The traversal can then visit the `<p>` element, the `` element, and the two text nodes that are descendants of `<body>`; however, the traversal can never reach the `<html>` element, the `<head>` element, or any other node that isn't in the `<body>` element's subtree. A traversal that has its root at `document`, on the other hand, can access all of the nodes in `document`. Figure 12-5 depicts a depth-first traversal of a DOM tree rooted at `document`.

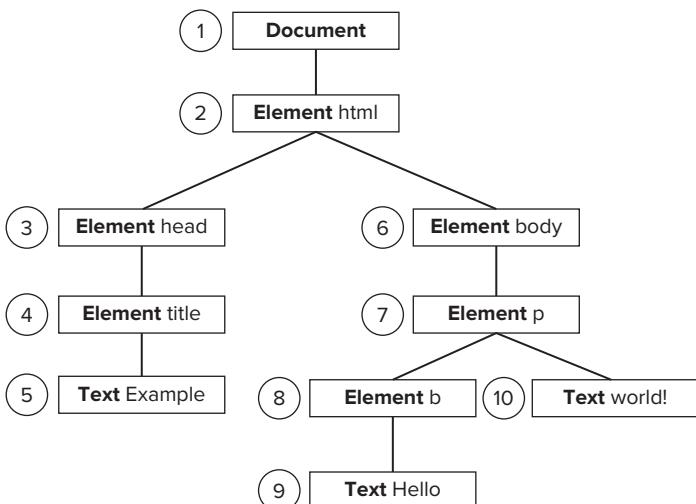


FIGURE 12-5

Starting at `document` and moving sequentially, the first node visited is `document` and the last node visited is the text node containing " world!" From the very last text node at the end of the document, the traversal can be reversed to go back up the tree. In that case, the first node visited is the text node containing " world!" and the last one visited is the `document` node itself. Both `NodeIterator` and `TreeWalker` perform traversals in this manner.

Nodelterator

The `NodeIterator` type is the simpler of the two, and a new instance can be created using the `document.createNodeIterator()` method. This method accepts the following four arguments:

- `root` — The node in the tree that you want to start searching from.
- `whatToShow` — A numerical code indicating which nodes should be visited.
- `filter` — A `NodeFilter` object or a function indicating whether a particular node should be accepted or rejected.
- `entityReferenceExpansion` — A Boolean value indicating whether entity references should be expanded. This has no effect in HTML pages, because entity references are never expanded.

The `whatToShow` argument is a bitmask that determines which nodes to visit by applying one or more filters. Possible values for this argument are included as constants on the `NodeFilter` type as follows:

- `NodeFilter.SHOW_ALL` — Show all node types.
- `NodeFilter.SHOW_ELEMENT` — Show element nodes.
- `NodeFilter.SHOW_ATTRIBUTE` — Show attribute nodes. This can't actually be used because of the DOM structure.
- `NodeFilter.SHOW_TEXT` — Show text nodes.
- `NodeFilter.SHOW_CDATA_SECTION` — Show CData section nodes. This is not used in HTML pages.
- `NodeFilter.SHOW_ENTITY_REFERENCE` — Show entity reference nodes. This is not used in HTML pages.
- `NodeFilter.SHOW_ENTITY` — Show entity nodes. This is not used in HTML pages.
- `NodeFilter.SHOW_PROCESSING_INSTRUCTION` — Show PI nodes. This is not used in HTML pages.
- `NodeFilter.SHOW_COMMENT` — Show comment nodes.
- `NodeFilter.SHOW_DOCUMENT` — Show document nodes.
- `NodeFilter.SHOW_DOCUMENT_TYPE` — Show document type nodes.
- `NodeFilter.SHOW_DOCUMENT_FRAGMENT` — Show document fragment nodes. This is not used in HTML pages.
- `NodeFilter.SHOW_NOTATION` — Show notation nodes. This is not used in HTML pages.

With the exception of `NodeFilter.SHOW_ALL`, you can combine multiple options using the bitwise OR operator, as shown in the following example:

```
var whatToShow = NodeFilter.SHOW_ELEMENT | NodeFilter.SHOW_TEXT;
```

The `filter` argument of `createNodeIterator()` can be used to specify a custom `NodeFilter` object or a function that acts as a node filter. A `NodeFilter` object has only one method, `acceptNode()`, which returns `NodeFilter.FILTER_ACCEPT` if the given node should be visited or `NodeFilter.FILTER_SKIP` if the given node should not be visited. Since `NodeFilter` is an abstract type, it's not possible to create an instance of it. Instead, just create an object with an `acceptNode()` method and pass the object into `createNodeIterator()`. The following code accepts only `<p>` elements:

```
var filter = {
    acceptNode: function(node) {
        return node.tagName.toLowerCase() == "p" ?
            NodeFilter.FILTER_ACCEPT :
            NodeFilter.FILTER_SKIP;
    }
};

var iterator = document.createNodeIterator(root, NodeFilter.SHOW_ELEMENT,
    filter, false);
```

The third argument can also be a function that takes the form of the `acceptNode()` method, as shown in this example:

```
var filter = function(node){
    return node.tagName.toLowerCase() == "p" ?
        NodeFilter.FILTER_ACCEPT :
        NodeFilter.FILTER_SKIP;
};

var iterator = document.createNodeIterator(root, NodeFilter.SHOW_ELEMENT,
    filter, false);
```

Typically, this is the form that is used in JavaScript, since it is simpler and works more like the rest of JavaScript. If no filter is required, the third argument should be set to `null`.

To create a simple `NodeIterator` that visits all node types, use the following code:

```
var iterator = document.createNodeIterator(document, NodeFilter.SHOW_ALL,
    null, false);
```

The two primary methods of `NodeIterator` are `nextNode()` and `previousNode()`. The `nextNode()` method moves one step forward in the depth-first traversal of the DOM subtree, and `previousNode()` moves one step backward in the traversal. When the `NodeIterator` is first created, an internal pointer points to the root, so the first call to `nextNode()` returns the root. When the traversal has reached the last node in the DOM subtree, `nextNode()` returns `null`. The `previousNode()` method works in a similar way. When the traversal has reached the last node in the DOM subtree, after `previousNode()` has returned the root of the traversal, it will return `null`.



Available for
download on
Wrox.com

```
<div id="div1">
    <p><b>Hello</b> world!</p>
    <ul>
        <li>List item 1</li>
        <li>List item 2</li>
        <li>List item 3</li>
    </ul>
</div>
```

NodeIteratorExample1.htm

Suppose that you would like to traverse all elements inside of the `<div>` element. This can be accomplished using the following code:

```
var div = document.getElementById("div1");
var iterator = document.createNodeIterator(div, NodeFilter.SHOW_ELEMENT,
    null, false);

var node = iterator.nextNode();
while (node !== null) {
    alert(node.tagName);           //output the tag name
    node = iterator.nextNode();
}
```

NodeIteratorExample1.htm

The first call to `nextNode()` in this example returns the `<p>` element. Since `nextNode()` returns `null` when it has reached the end of the DOM subtree, a `while` loop checks to see when `null` has been returned as it calls `nextNode()` each time through. When this code is executed, alerts are displayed with the following tag names:

```
DIV
P
B
UL
LI
LI
LI
```

Perhaps this is too much information and you really only want to return the `` elements that occur in the traversal. This can be accomplished by using a filter, as shown in the following example:

```
var div = document.getElementById("div1");
var filter = function(node){
    return node.tagName.toLowerCase() == "li" ?
        NodeFilter.FILTER_ACCEPT :
        NodeFilter.FILTER_SKIP;
```

```

};

var iterator = document.createNodeIterator(div, NodeFilter.SHOW_ELEMENT,
                                         filter, false);

var node = iterator.nextNode();
while (node !== null) {
    alert(node.tagName);           //output the tag name
    node = iterator.nextNode();
}

```

[NodeIteratorExample2.htm](#)

In this example, only `` elements will be returned from the iterator.

The `nextNode()` and `previousNode()` methods work with `NodeIterator`'s internal pointer in the DOM structure, so changes to the structure are represented appropriately in the traversal.



Firefox versions prior to 3.5 do not implement the `createNodeIterator()` method, though they do support `createTreeWalker()`, as discussed in the next section.

TreeWalker

`TreeWalker` is a more advanced version of `NodeIterator`. It has the same functionality, including `nextNode()` and `previousNode()`, and adds the following methods to traverse a DOM structure in different directions:

- ▶ `parentNode()` — Travels to the current node's parent.
- ▶ `firstChild()` — Travels to the first child of the current node.
- ▶ `lastChild()` — Travels to the last child of the current node.
- ▶ `nextSibling()` — Travels to the next sibling of the current node.
- ▶ `previousSibling()` — Travels to the previous sibling of the current node.

A `TreeWalker` object is created using the `document.createTreeWalker()` method, which accepts the same three arguments as `document.createNodeIterator()`: the root to traverse from, which node types to show, a filter, and a Boolean value indicating if entity references should be expanded. Because of these similarities, `TreeWalker` can always be used in place of `NodeIterator`, as in this example:



Available for
download on
[Wrox.com](#)

```

var div = document.getElementById("div1");
var filter = function(node){
    return node.tagName.toLowerCase() == "li" ?
        NodeFilter.FILTER_ACCEPT :
        NodeFilter.FILTER_SKIP;
};

var walker = document.createTreeWalker(div, NodeFilter.SHOW_ELEMENT,

```

```

        filter, false);

var node = iterator.nextNode();
while (node !== null) {
    alert(node.tagName);           //output the tag name
    node = iterator.nextNode();
}

```

[TreeWalkerExample1.htm](#)

One difference is in the values that the `filter` can return. In addition to `NodeFilter.FILTER_ACCEPT` and `NodeFilter.FILTER_SKIP`, there is `NodeFilter.FILTER_REJECT`. When used with a `NodeIterator` object, `NodeFilter.FILTER_SKIP` and `NodeFilter.FILTER_REJECT` do the same thing: they skip over the node. When used with a `TreeWalker` object, `NodeFilter.FILTER_SKIP` skips over the node and goes on to the next node in the subtree, whereas `NodeFilter.FILTER_REJECT` skips over that node and that node's entire subtree. For instance, changing the filter in the previous example to return `NodeFilter.FILTER_REJECT` instead of `NodeFilter.FILTER_SKIP` will result in no nodes being visited. This is because the first element returned is `<div>`, which does not have a tag name of "li", so `NodeFilter.FILTER_REJECT` is returned, indicating that the entire subtree should be skipped. Since the `<div>` element is the traversal root, this means that the traversal stops.

Of course, the true power of `TreeWalker` is its ability to move around the DOM structure. Instead of specifying `filter`, it's possible to get at the `` elements by navigating through the DOM tree using `TreeWalker`, as shown here:



```

var div = document.getElementById("div1");
var walker = document.createTreeWalker(div, NodeFilter.SHOW_ELEMENT, null, false);

walker.firstChild();    //go to <p>
walker.nextSibling(); //go to <ul>

var node = walker.firstChild(); //go to first <li>
while (node !== null) {
    alert(node.tagName);
    node = walker.nextSibling();
}

```

[TreeWalkerExample2.htm](#)

Since you know where the `` elements are located in the document structure, it's possible to navigate there, using `firstChild()` to get to the `<p>` element, `nextSibling()` to get to the `` element, and then `firstChild()` to get to the first `` element. Keep in mind that `TreeWalker` is returning only elements (because of the second argument passed in to `createTreeWalker()`). Then, `nextSibling()` can be used to visit each `` until there are no more, at which point the method returns `null`.

The `TreeWalker` type also has a property called `currentNode` that indicates the node that was last returned from the traversal via any of the traversal methods. This property can also be set to change where the traversal continues from when it resumes, as shown in this example:

```
var node = walker.nextSibling();
alert(node === walker.currentNode); //true
walker.currentNode = document.body; //change where to start from
```

Compared to `NodeIterator`, the `TreeWalker` type allows greater flexibility when traversing the DOM. There is no equivalent in Internet Explorer 8 and earlier, so cross-browser solutions using traversals are quite rare.

RANGES

To allow an even greater measure of control over a page, the DOM Level 2 Traversal and Range module defines an interface called a range. A range can be used to select a section of a document regardless of node boundaries. (This selection occurs behind the scenes and cannot be seen by the user.) Ranges are helpful when regular DOM manipulation isn't specific enough to change a document. DOM ranges are supported in Firefox, Opera, Safari, and Chrome. Internet Explorer implements ranges in a proprietary way.

Ranges in the DOM

DOM Level 2 defines a method on the `Document` type called `createRange()`. In DOM-compliant browsers, this method belongs to the `document` object. You can test for the range support by using `hasFeature()` or by checking for the method directly. Here is an example:

```
var supportsRange = document.implementation.hasFeature("Range", "2.0");
var alsoSupportsRange = (typeof document.createRange == "function");
```

If the browser supports it, a DOM range can be created using `createRange()`, as shown here:

```
var range = document.createRange();
```

Similar to nodes, the newly created range is tied directly to the document on which it was created and cannot be used on other documents. This range can then be used to select specific parts of the document behind the scenes. Once a range has been created and its position set, a number of different operations can be performed on the contents of the range, allowing more fine-grained manipulation of the underlying DOM tree.

Each range is represented by an instance of the `Range` type, which has a number of properties and methods. The following properties provide information about where the range is located in the document:

- `startContainer` — The node within which the range starts (the parent of the first node in the selection).
- `startOffset` — The offset within the `startContainer` where the range starts. If `startContainer` is a text node, comment node, or CData node, the `startOffset` is the number of characters skipped before the range starts; otherwise, the offset is the index of the first child node in the range.

- `endContainer` — The node within which the range ends (the parent of the last node in the selection).
- `endOffset` — The offset within the `endContainer` where the range ends (follows the same rules as `startOffset`).
- `commonAncestorContainer` — The deepest node in the document that has both `startContainer` and `endContainer` as descendants.

These properties are filled when the range is placed into a specific position in the document.

Simple Selection in DOM Ranges

The simplest way to select a part of the document using a range is to use either `selectNode()` or `selectNodeContents()`. These methods each accept one argument, a DOM node, and fill a range with information from that node. The `selectNode()` method selects the entire node, including its children, whereas `selectNodeContents()` selects only the node's children. For example, consider the following HTML:

```
<!DOCTYPE html>
<html>
  <body>
    <p id="p1"><b>Hello</b> world!</p>
  </body>
</html>
```

This code can be accessed using the following JavaScript:



Available for
download on
Wrox.com

```
var range1 = document.createRange(),
  range2 = document.createRange(),
  p1 = document.getElementById("p1");
range1.selectNode(p1);
range2.selectNodeContents(p1);
```

[DOMRangeExample.htm](#)

The two ranges in this example contain different sections of the document: `range1` contains the `<p>` element and all its children, whereas `range2` contains the `` element, the text node "Hello", and the text node " world!". See Figure 12-6.



FIGURE 12-6

When `selectNode()` is called, `startContainer`, `endContainer`, and `commonAncestorContainer` are all equal to the parent node of the node that was passed in; in this example, these would all be equal to `document.body`. The `startOffset` property is equal to the index of the given node within the parent's `childNodes` collection (which is 1 in this example — remember DOM-compliant browsers count white space as text nodes), whereas `endOffset` is equal to the `startOffset` plus one (because only one node is selected).

When `selectNodeContents()` is called, `startContainer`, `endContainer`, and `commonAncestorContainer` are equal to the node that was passed in, which is the `<p>` element in this example.

The `startOffset` property is always equal to 0, since the range begins with the first child of the given node, whereas `endOffset` is equal to the number of child nodes (`node.childNodes.length`), which is 2 in this example.

It's possible to get more fine-grained control over which nodes are included in the selection by using the following range methods:

- `setStartBefore(refNode)` — Sets the starting point of the range to begin before `refNode`, so `refNode` is the first node in the selection. The `startContainer` property is set to `refNode.parentNode`, and the `startOffset` property is set to the index of `refNode` within its parent's `childNodes` collection.
- `setStartAfter(refNode)` — Sets the starting point of the range to begin after `refNode`, so `refNode` is not part of the selection; rather, its next sibling is the first node in the selection. The `startContainer` property is set to `refNode.parentNode`, and the `startOffset` property is set to the index of `refNode` within its parent's `childNodes` collection plus one.
- `setEndBefore(refNode)` — Sets the ending point of the range to begin before `refNode`, so `refNode` is not part of the selection; its previous sibling is the last node in the selection. The `endContainer` property is set to `refNode.parentNode`, and the `endOffset` property is set to the index of `refNode` within its parent's `childNodes` collection.
- `setEndAfter(refNode)` — Sets the ending point of the range to begin before `refNode`, so `refNode` is the last node in the selection. The `endContainer` property is set to `refNode.parentNode`, and the `endOffset` property is set to the index of `refNode` within its parent's `childNodes` collection plus one.

Using any of these methods, all properties are assigned for you. However, it is possible to assign these values directly in order to make complex range selections.

Complex Selection in DOM Ranges

Creating complex ranges requires the use of the `setStart()` and `setEnd()` methods. Both methods accept two arguments: a reference node and an offset. For `setStart()`, the reference node becomes the `startContainer`, and the offset becomes the `startOffset`. For `setEnd()`, the reference node becomes the `endContainer`, and the offset becomes the `endOffset`.

Using these methods, it is possible to mimic `selectNode()` and `selectNodeContents()`. Here is an example:



```

var range1 = document.createRange(),
    range2 = document.createRange(),
    p1 = document.getElementById("p1"),
    p1Index = -1,
    i, len;
for (i=0, len=p1.parentNode.childNodes.length; i < len; i++) {
    if (p1.parentNode.childNodes[i] == p1) {
        p1Index = i;
        break;
    }
}
range1.setStart(p1.parentNode, p1Index);

```

```
range1.setEnd(p1.parentNode, p1Index + 1);
range2.setStart(p1, 0);
range2.setEnd(p1, p1.childNodes.length);
```

[DOMRangeExample2.htm](#)

Note that to select the node (using `range1`), you must first determine the index of the given node (`p1`) in its parent node's `childNodes` collection. To select the node contents (using `range2`), you do not need calculations; `setStart()` and `setEnd()` can be set with default values. Although mimicking `selectNode()` and `selectNodeContents()` is possible, the real power of `setStart()` and `setEnd()` is in the partial selection of nodes.

Suppose that you want to select only from the "llo" in "Hello" to the "o" in "world!" in the previous HTML code. This is quite easy to accomplish. The first step is to get references to all of the relevant nodes, as shown in the following example:



```
var p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild
```

[DOMRangeExample3.htm](#)

The "Hello" text node is actually a grandchild of `<p>` because it's a child of ``, so you can use `p1.firstChild` to get `` and `p1.firstChild.firstChild` to get the text node. The "world!" text node is the second (and the last) child of `<p>`, so you can use `p1.lastChild` to retrieve it. Next, the range must be created and its boundaries defined, as shown in the following example:

```
var range = document.createRange();
range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);
```

[DOMRangeExample3.htm](#)

Since the selection should start after the "e" in "Hello", `helloNode` is passed into `setStart()` with an offset of 2 (the position after the "e" where "H" is in position 0). To set the end of the selection, pass `worldNode` into `setEnd()` with an offset of 3, indicating the first character that should not be selected, which is "r" in position 3 (there is actually a space in position 0). See Figure 12-7.

Because both `helloNode` and `worldNode` are text nodes, they become the `startContainer` and `endContainer` for the range so that the `startOffset` and `endOffset` accurately look at the text contained within each node instead of look for child nodes (which is what happens when an element is passed in). The `commonAncestorContainer` is the `<p>` element, which is the first ancestor that contains both nodes.

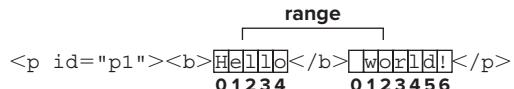


FIGURE 12-7

Of course, just selecting sections of the document isn't very useful unless you can interact with the selection.

Interacting with DOM Range Content

When a range is created, internally it creates a document fragment node onto which all of the nodes in the selection are attached. The range contents must be well formed in order for this process to take place. In the previous example, the range does not represent a well-formed DOM structure, because the selection begins inside one text node and ends in another, which cannot be represented in the DOM. Ranges, however, recognize missing opening and closing tags and are, therefore, able to reconstruct a valid DOM structure to operate on.

In the previous example, the range calculates that a `` start tag is missing inside the selection, so the range dynamically adds it behind the scenes, along with a new `` end tag to enclose "He", thus altering the DOM to the following:

```
<p><b>He</b><b>Hello</b> world!</p>
```

Additionally, the "world!" text node is split into two text nodes, one containing "wo" and the other containing "rld!". The resulting DOM tree is shown in Figure 12-8, along with the contents of the document fragment for the range.

With the range created, the contents of the range can be manipulated using a variety of methods. (Note that all nodes in the range's internal document fragment are simply pointers to nodes in the document.)

The first method is the simplest to understand and use: `deleteContents()`. This method simply deletes the contents of the range from the document. Here is an example:



Available for
download on
Wrox.com

```
var p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

range.deleteContents();
```

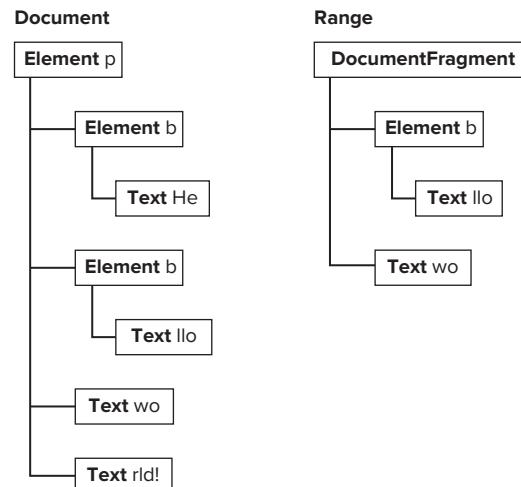


FIGURE 12-8

[DOMRangeExample4.htm](#)

Executing this code results in the following HTML being shown on the page:

```
<p><b>He</b>rld!</p>
```

Since the range selection process altered the underlying DOM structure to remain well formed, the resulting DOM structure is well formed even after removing the contents.

`extractContents()` is similar to `deleteContents()` in that it also removes the range selection from the document. The difference is that `extractContents()` returns the range's document fragment as the function value. This allows you to insert the contents of the range somewhere else. Here is an example:



```
var p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

var fragment = range.extractContents();
p1.parentNode.appendChild(fragment);
```

[DOMRangeExample5.htm](#)

In this example, the fragment is extracted and added to the end of the document's `<body>` element. (Remember, when a document fragment is passed into `appendChild()`, only the fragment's children are added, not the fragment itself.) The resulting HTML is as follows:

```
<p><b>He</b>rld!</p>
<b>llo</b> wo
```

Another option is to leave the range in place but create a clone of it that can be inserted elsewhere in the document by using `cloneContents()`, as shown in this example:

```
var p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

var fragment = range.cloneContents();
p1.parentNode.appendChild(fragment);
```

[DOMRangeExample6.htm](#)

This method is very similar to `extractContents()` because both return a document fragment. The main difference is that the document fragment returned by `cloneContents()` contains clones of the nodes contained in the range instead of the actual nodes. With this operation, the HTML in the page is as follows:

```
<p><b>Hello</b> world!</p>
<b>llo</b> wo
```

It's important to note that the splitting of nodes ensures that a well-formed document isn't produced until one of these methods is called. The original HTML remains intact right up until the point that the DOM is modified.

Inserting DOM Range Content

Ranges can be used to remove or clone content, as seen in the previous section, and to manipulate the contents inside of the range. The `insertNode()` method enables you to insert a node at the beginning of the range selection. As an example, suppose that you want to insert the following HTML prior to the HTML used in the previous example:

```
<span style="color: red">Inserted text</span>
```

The following code accomplishes this:



Available for
download on
Wrox.com

```
var p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

var span = document.createElement("span");
span.style.color = "red";
span.appendChild(document.createTextNode("Inserted text"));
range.insertNode(span);
```

DOMRangeExample7.htm

Running this JavaScript effectively creates the following HTML code:

```
<p id="p1"><b>He<span style="color: red">Inserted text</span>llo</b> world</p>
```

Note that `` is inserted just before the "llo" in "Hello", which is the first part of the range selection. Also note that the original HTML didn't add or remove `` elements, because none of the methods introduced in the previous section were used. You can use this technique to insert helpful information, such as an image next to links that open in a new window.

Along with inserting content into the range, it is possible to insert content surrounding the range by using the `surroundContents()` method. This method accepts one argument, which is the node that surrounds the range contents. Behind the scenes, the following steps are taken:

1. The contents of the range are extracted (similarly to using `extractContents()`).
2. The given node is inserted into the position in the original document where the range was.
3. The contents of the document fragment are added to the given node.



This sort of functionality is useful online to highlight certain words in a web page, as shown here:

```
var p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.selectNode(helloNode);

var span = document.createElement("span");
span.style.backgroundColor = "yellow";
range.surroundContents(span);
```

[DOMRangeExample8.htm](#)

This code highlights the range selection with a yellow background. The resulting HTML is as follows:

```
<p><b><span style="background-color:yellow">Hello</span></b> world!</p>
```

In order to insert the ``, the range has to contain a whole DOM selection. (It can't have only partially selected DOM nodes.)

Collapsing a DOM Range

When a range isn't selecting any part of a document, it is said to be **collapsed**. Collapsing a range resembles the behavior of a text box. When you have text in a text box, you can highlight an entire word using the mouse. However, if you left-click the mouse again, the selection is removed and the cursor is located between two letters. When you collapse a range, its location is set between parts of a document, either at the beginning of the range selection or at the end. Figure 12-9 illustrates what happens when a range is collapsed.

<p id="p1">Hello world!</p>
Original Range

<p id="p1">Hello world!</p>
Collapsed to beginning

<p id="p1">Hello worl</p>
Collapsed to end

FIGURE 12-9

You can collapse a range by using the `collapse()` method, which accepts a single argument: a Boolean value indicating which end of the range to collapse to. If the argument is `true`, then the range is collapsed to its starting point; if it is `false`, the range is collapsed to its ending point. To determine if a range is already collapsed, you can use the `collapsed` property as follows:

```
range.collapse(true);      //collapse to the starting point
alert(range.collapsed);   //outputs "true"
```

Testing whether a range is collapsed is helpful if you aren't sure if two nodes in the range are next to each other. For example, consider this HTML code:

```
<p id="p1">Paragraph 1</p><p id="p2">Paragraph 2</p>
```

If you don't know the exact makeup of this code (for example, if it is automatically generated), you might try creating a range like this:

```
var p1 = document.getElementById("p1"),
    p2 = document.getElementById("p2"),
    range = document.createRange();
range.setStartAfter(p1);
range.setStartBefore(p2);
alert(range.collapsed);      //outputs "true"
```

In this case, the created range is collapsed, because there is nothing between the end of `p1` and the beginning of `p2`.

Comparing DOM Ranges

If you have more than one range, you can use the `compareBoundaryPoints()` method to determine if the ranges have any boundaries (start or end) in common. The method accepts two arguments: the range to compare to and how to compare. It is one of the following constant values:

- `Range.START_TO_START` (0) — Compares the starting point of the first range to the starting point of the second.
- `Range.START_TO_END` (1) — Compares the starting point of the first range to the end point of the second.
- `Range.END_TO_END` (2) — Compares the end point of the first range to the end point of the second.
- `Range.END_TO_START` (3) — Compares the end point of the first range to the starting point of the second.

The `compareBoundaryPoints()` method returns `-1` if the point from the first range comes before the point from the second range, `0` if the points are equal, or `1` if the point from the first range comes after the point from the second range. Here is an example:



Available for
download on
Wrox.com

```
var range1 = document.createRange();
var range2 = document.createRange();
var p1 = document.getElementById("p1");

range1.selectNodeContents(p1);
range2.selectNodeContents(p1);
range2.setEndBefore(p1.lastChild);

alert(range1.compareBoundaryPoints(Range.START_TO_START, range2)); //0
alert(range1.compareBoundaryPoints(Range.END_TO_END, range2)); //1
```

[DOMRangeExample9.htm](#)

In this code, the starting points of the two ranges are exactly the same because both use the default value from `selectNodeContents()`; therefore, the method returns `0`. For `range2`, however, the

end point is changed using `setEndBefore()`, making the end point of `range1` come after the end point of `range2` (see Figure 12-10), so the method returns 1.

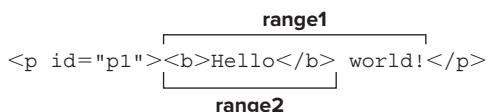


FIGURE 12-10

Cloning DOM Ranges

Ranges can be cloned by calling the `cloneRange()` method. This method creates an exact duplicate of the range on which it is called:

```
var newRange = range.cloneRange();
```

The new range contains all of the same properties as the original, and its end points can be modified without affecting the original in any way.

Clean Up

When you are done using a range, it is best to call the `detach()` method, which detaches the range from the document on which it was created. After calling `detach()`, the range can be safely dereferenced, so the memory can be reclaimed through garbage collection. Here is an example:

```
range.detach(); //detach from document
range = null; //dereferenced
```

Following these two steps is the most appropriate way to finish using a range. Once it is detached, a range can no longer be used.

Ranges in Internet Explorer 8 and Earlier

While Internet Explorer 9 supports DOM ranges, versions 8 and earlier do not. Earlier versions do, however, support a similar concept called **text ranges**. Text ranges are proprietary to Internet Explorer and so have not been implemented in any other browsers. This type of range deals specifically with text (not necessarily DOM nodes). The `createTextRange()` method can be called on a small number of elements: `<body>`, `<button>`, `<input>`, and `<textarea>`. Here is an example:

```
var range = document.body.createTextRange();
```

Creating a range in this way allows it to be used anywhere on the page (whereas creating a range on one of the other specified elements limits the range to working on that element). As with DOM ranges, there are a number of ways to use Internet Explorer text ranges.

Simple Selection in Internet Explorer Ranges

The simplest way to select an area of the page is to use a range's `findText()` method. This method finds the first instance of a given text string and moves the range to surround it. If the text isn't found, the method returns `false`; otherwise, it returns `true`. Once again, consider the following HTML code:

```
<p id="p1"><b>Hello</b> world!</p>
```

To select "Hello", you can use the following code:



Available for download on
Wrox.com

```
var range = document.body.createTextRange();
var found = range.findText("Hello");
```

[IERangeExample1.htm](#)

After the second line of code, the text "Hello" is contained within the range. You can test this by using the range's `text` property (which returns the text contained in the range) or checking the returned value of `findText()`, which is `true` if the text was found. Here is an example:

```
alert(found);           //true
alert(range.text);    //"Hello"
```

There is a second argument to `findText()`, which is a number indicating the direction in which to continue searching. A negative number indicates that the search should go backward from the current position, whereas a positive number indicates that the search should go forward from the current position. So, to find the first two instances of "Hello" in the document, you can use the following code:

```
var found = range.findText("Hello");
var foundAgain = range.findText("Hello", 1);
```

The closest thing to the DOM's `selectNode()` in Internet Explorer is `moveToElementText()`, which accepts a DOM element as an argument and selects all of the element's text, including HTML tags. Here is an example:

```
var range = document.body.createTextRange();
var p1 = document.getElementById("p1");
range.moveToElementText(p1);
```

[IERangeExample2.htm](#)

When HTML is contained in a text range, the `htmlText` property can be used to return the entire contents of the range, including HTML and text, as shown in this example:

```
alert(range.htmlText);
```

Ranges in Internet Explorer don't have any other properties that are dynamically updated as the range selection changes, although the `parentElement()` method behaves the same as the DOM's `commonAncestorContainer` property, as shown here:

```
var ancestor = range.parentElement();
```

The parent element always reflects the parent node for the text selection.

Complex Selection in Internet Explorer Ranges

Complex ranges can be created in Internet Explorer by moving the range selection around in specific increments. This can be done using four methods: `move()`, `moveStart()`, `moveEnd()`, and

`expand()`. Each of these methods accepts two arguments: the type of unit to move and the number of units to move. The type of units to move is one of the following string values:

- "character" — Moves a point by one character.
- "word" — Moves a point by one word (a sequence of non-white-space characters).
- "sentence" — Moves a point by one sentence (a sequence of characters ending with a period, question mark, or exclamation point).
- "textedit" — Moves a point to the start or end of the current range selection.

The `moveStart()` method moves the starting point of the range by the given number of units, whereas the `moveEnd()` method moves the end point of the range by the given number of units, as shown in the following example:

```
range.moveStart("word", 2);      //move the start point by two words
range.moveEnd("character", 1);   //move the ending point by one character
```

You can also use the `expand()` method to normalize the range. The `expand()` method makes sure that any partially selected units become fully selected. For example, if you selected only the middle two characters of a word, you can call `expand("word")` to ensure that the entire word is enclosed by the range.

The `move()` method first collapses the range (making the start and end points equal) and then moves the range by the specified number of units, as shown in the following example:

```
range.move("character", 5);    //move over five characters
```

After using `move()`, the start and end points are equal, so you must use either `moveStart()` or `moveEnd()` to once again make a selection.

Interacting with Internet Explorer Range Content

Interacting with a range's content in Internet Explorer is done through either the `text` property or the `pasteHTML()` method. The `text` property, used previously to retrieve the text content of the range, can also be used to set the text content of the range. Here is an example:

```
var range = document.body.createTextRange();
range.findText("Hello");
range.text = "Howdy";
```

If you run this code against the same “Hello world!” code shown earlier, the HTML result is as follows:

```
<p id="p1"><b>Howdy</b> world!</p>
```

Note that all the HTML tags remained intact when setting the `text` property.

To insert HTML code into the range, you can use the `pasteHTML()` method, as shown in the following example:



```
var range = document.body.createTextRange();
range.findText("Hello");
range.pasteHTML("<em>Howdy</em>");
```

[IERangeExample3.htm](#)

After executing this code, the following is the resulting HTML:

```
<p id="p1"><b><em>Howdy</em></b> world!</p>
```

You should not use `pasteHTML()` when the range contains HTML code, because this causes unpredictable results, and you may end up with malformed HTML.

Collapsing an Internet Explorer Range

Ranges in Internet Explorer have a `collapse()` method that works exactly the same way as the DOM method: pass in `true` to collapse the range to the beginning or `false` to collapse the range to the end. Here's an example:

```
range.collapse(true); //collapse to start
```

Unfortunately, no corresponding `collapsed` property tells you whether a range is already collapsed. Instead, you must use the `boundingWidth` property, which returns the width (in pixels) of the range. If `boundingWidth` is equal to 0, the range is collapsed as follows:

```
var isCollapsed = (range.boundingWidth == 0);
```

The `boundingHeight`, `boundingLeft`, and `boundingTop` properties also give information about the range location, although these are less helpful than `boundingWidth`.

Comparing Internet Explorer Ranges

The `compareEndPoints()` method in Internet Explorer is similar to the DOM range's `compareBoundaryPoints()` method. This method accepts two arguments: the type of comparison and the range to compare to. The type of comparison is indicated by one of the following string values: "StartToStart", "StartToEnd", "EndToEnd", and "EndToStart". These comparisons are equal to the corresponding values in DOM ranges.

Also similar to the DOM, `compareEndPoints()` returns -1 if the first range boundary occurs before the second range boundary, 0 if they are equal, and 1 if the first range boundary occurs after the second range boundary. Once again using the "Hello world!" code from the previous example, the following code creates two ranges, one that selects "Hello world!" (including the `` tags) and one that selects "Hello":

```
var range1 = document.body.createTextRange(),
    range2 = document.body.createTextRange();

range1.findText("Hello world!");
```

```
range2.findText("Hello");

alert(range1.compareEndPoints("StartToStart", range2)); //0
alert(range1.compareEndPoints("EndToEnd", range2)); //1
```

[IERangeExample5.htm](#)

The first and second ranges share the same starting point, so comparing them using `compareEndPoints()` returns 0. `range1`'s end point occurs after `range2`'s end point, so `compareEndPoints()` returns 1.

Internet Explorer also has two additional methods for comparing ranges: `isEqual()`, which determines if two ranges have the same start and end points, and `inRange()`, which determines if a range occurs inside of another range. Here is an example:



Available for
download on
Wrox.com

```
var range1 = document.body.createTextRange();
var range2 = document.body.createTextRange();
range1.findText("Hello world!");
range2.findText("Hello");
alert("range1.isEqual(range2): " + range1.isEqual(range2)); //false
alert("range1.inRange(range2): " + range1.inRange(range2)); //true
```

[IERangeExample6.htm](#)

This example uses the same ranges as in the previous example to illustrate these methods. The ranges are not equal, because the end points are different, so calling `isEqual()` returns false. However, `range2` is actually inside of `range1`, because its end point occurs before `range1`'s end point but after `range1`'s start point. For this reason, `range2` is considered to be inside of `range1`, so `inRange()` returns true.

Cloning an Internet Explorer Range

Text ranges can be cloned in Internet Explorer using the `duplicate()` method, which creates an exact clone of the range, as shown in the following example:

```
var newRange = range.duplicate();
```

All properties from the original range are carried over into the newly created one.

SUMMARY

The DOM Level 2 specifications define several modules that augment the functionality of DOM Level 1. DOM Level 2 Core introduces several new methods related to XML namespaces on various DOM types. These changes are relevant only when used in XML or XHTML documents; they have no use in HTML documents. Methods not related to XML namespaces include the ability to programmatically create new instances of `Document` and to enable the creation of `DocumentType` objects.

The DOM Level 2 Style module specifies how to interact with stylistic information about elements as follows:

- Every element has a `style` object associated with it that can be used to determine and change inline styles.
- To determine the computed style of an element, including all CSS rules that apply to it, you can use a method called `getComputedStyle()`.
- Internet Explorer doesn't support this method but offers a `currentStyle` property on all elements that returns the same information.
- It's also possible to access style sheets via the `document.styleSheets` collection.
- The interface for style sheets is supported by all browsers except Internet Explorer 8 and earlier, which offer comparable properties and methods for almost all DOM functionality.

The DOM Level 2 Traversals and Range module specifies different ways to interact with a DOM structure as follows:

- Traversals are handled using either `NodeIterator` or `TreeWalker` to perform depth-first traversals of a DOM tree.
- The `NodeIterator` interface is simple, allowing only forward and backward movement in one-step increments. The `TreeWalker` interface supports the same behavior and moves across the DOM structure in all other directions, including parents, siblings, and children.
- Ranges are a way to select specific portions of a DOM structure to augment it in some fashion.
- Selections of ranges can be used to remove portions of a document while retaining a well-formed document structure or for cloning portions of a document.
- Internet Explorer 8 and earlier don't support DOM Level 2 Traversals and Range, though they offer a proprietary text range object that can be used to do simple text-based range manipulation. Internet Explorer 9 fully supports DOM traversals.

13

Events

WHAT'S IN THIS CHAPTER?

- Understanding event flow
- Working with event handlers
- Examining the different types of events

JavaScript's interaction with HTML is handled through *events*, which indicate when particular moments of interest occur in the document or browser window. Events can be subscribed to using *listeners* (also called handlers) that execute only when an event occurs. This model, called the observer pattern in traditional software engineering, allows a loose coupling between the behavior of a page (defined in JavaScript) and the appearance of the page (defined in HTML and CSS).

Events first appeared in Internet Explorer 3 and Netscape Navigator 2 as a way to offload some form processing from the server onto the browser. By the time Internet Explorer 4 and Netscape 4 were released, each browser delivered similar but different APIs that continued for several generations. DOM Level 2 was the first attempt to standardize the DOM events API in a logical way. Internet Explorer 9, Firefox, Opera, Safari, and Chrome all have implemented the core parts of DOM Level 2 Events. Internet Explorer 8 was the last major browser to use a purely proprietary event system.

The browser event system is a complex one. Even though all major browsers have implemented DOM Level 2 Events, the specification doesn't cover all event types. The BOM also supports events, and the relationship between these and the DOM events is often confusing because of a longtime lack of documentation (something that HTML5 has tried to clarify). Further complicating matters is the augmentation of the DOM events API by DOM Level 3. Working with events can be relatively simple or very complex, depending on your requirements. Still, there are some core concepts that are important to understand.

EVENT FLOW

When development for the fourth generation of web browsers began (Internet Explorer 4 and Netscape Communicator 4), the browser development teams were met with an interesting question: what part of a page owns a specific event? To understand the issue, consider a series of concentric circles on a piece of paper. When you place your finger at the center, it is inside of not just one circle but all of the circles on the paper. Both development teams looked at browser events in the same way. When you click on a button, they concluded, you're clicking not just on the button but also on its container and on the page as a whole.

Event flow describes the order in which events are received on the page, and interestingly, the Internet Explorer and Netscape development teams came up with an almost exactly opposite concept of event flow. Internet Explorer would support an event bubbling flow, whereas Netscape Communicator would support an event capturing flow.

Event Bubbling

The Internet Explorer event flow is called *event bubbling*, because an event is said to start at the most specific element (the deepest possible point in the document tree) and then flow upward toward the least specific node (the document). Consider the following HTML page:

```
<!DOCTYPE html>
<html>
<head>
    <title>Event Bubbling Example</title>
</head>
<body>
    <div id="myDiv">Click Me</div>
</body>
</html>
```

When you click the `<div>` element in the page, the `click` event occurs in the following order:

1. `<div>`
2. `<body>`
3. `<html>`
4. `document`

The `click` event is first fired on the `<div>`, which is the element that was clicked. Then the `click` event goes up the DOM tree, firing on each node along its way until it reaches the `document` object. Figure 13-1 illustrates this effect.

All modern browsers support event bubbling, although there are some variations on how it is implemented. Internet Explorer 5.5 and earlier skip bubbling to the `<html>` element (going from `<body>` directly to `document`). Internet Explorer 9, Firefox, Chrome, and Safari continue event bubbling up to the `window` object.

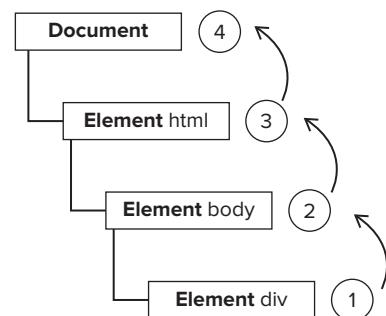


FIGURE 13-1

Event Capturing

The Netscape Communicator team came up with an alternate event flow called event capturing. The theory of event capturing is that the least specific node should receive the event first and the most specific node should receive the event last. Event capturing was really designed to intercept the event before it reached the intended target. If the previous example is used with event capturing, clicking the `<div>` element fires the `click` event in the following order:

1. document
2. `<html>`
3. `<body>`
4. `<div>`

With event capturing, the `click` event is first received by the `document` and then continues down the DOM tree to the actual target of the event, the `<div>` element. This flow is illustrated in Figure 13-2.

Although this was Netscape Communicator's only event flow model, event capturing is currently supported in Internet Explorer 9, Safari, Chrome, Opera, and Firefox. All of them actually begin event capturing at the window-level event despite the fact that the DOM Level 2 Events specification indicates that the events should begin at `document`.

Event capturing is generally not used because of a lack of support in older browsers. The general advice is to use event bubbling freely while retaining event capturing for special circumstances.

DOM Event Flow

The event flow specified by DOM Level 2 Events has three phases: the event capturing phase, at the target, and the event bubbling phase. Event capturing occurs first, providing the opportunity to intercept events if necessary. Next, the actual target receives the event. The final phase is bubbling, which allows a final response to the event. Considering the simple HTML example used previously, clicking the `<div>` fires the event in the order indicated in Figure 13-3.

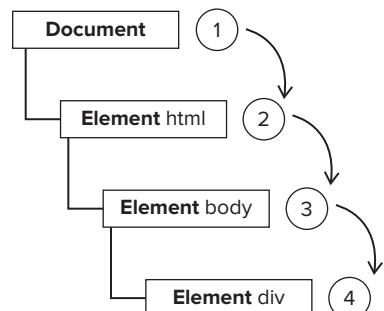


FIGURE 13-2

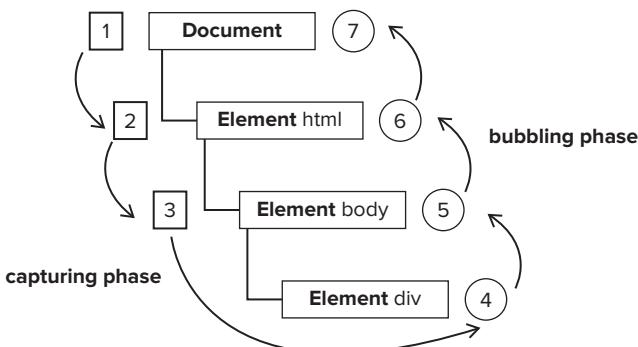


FIGURE 13-3

In the DOM event flow, the actual target (the `<div>` element) does not receive the event during the capturing phase. This means that the capturing phase moves from document to `<html>` to `<body>` and stops. The next phase is “at target,” which fires on the `<div>` and is considered to be part of the bubbling phase in terms of event handling (discussed later). Then, the bubbling phase occurs and the event travels back up to the document.

Most of the browsers that support DOM event flow have implemented a quirk. Even though the DOM Level 2 Events specification indicates that the capturing phase doesn’t hit the event target, Internet Explorer 9, Safari, Chrome, Firefox, and Opera 9.5 and later all fire an event during the capturing phase on the event target. The end result is that there are two opportunities to work with the event on the target.



Internet Explorer 9, Opera, Firefox, Chrome, and Safari all support the DOM event flow; Internet Explorer 8 and earlier do not.

EVENT HANDLERS

Events are certain actions performed either by the user or by the browser itself. These events have names like `click`, `load`, and `mouseover`. A function that is called in response to an event is called an *event handler* (or an *event listener*). Event handlers have names beginning with “on”, so an event handler for the `click` event is called `onclick` and an event handler for the `load` event is called `onload`. Assigning event handlers can be accomplished in a number of different ways.

HTML Event Handlers

Each event supported by a particular element can be assigned using an HTML attribute with the name of the event handler. The value of the attribute should be some JavaScript code to execute. For example, to execute some JavaScript when a button is clicked, you can use the following:

```
<input type="button" value="Click Me" onclick="alert('Clicked')"/>
```

When this button is clicked, an alert is displayed. This interaction is defined by specifying the `onclick` attribute and assigning some JavaScript code as the value. Note that since the JavaScript code is an attribute value, you cannot use HTML syntax characters such as the ampersand, double quotes, less-than, or greater-than without escaping them. In this case, single quotes were used instead of double quotes to avoid the need to use HTML entities. To use double quotes, you will change the code to the following:

```
<input type="button" value="Click Me" onclick="alert("Clicked")" />
```

An event handler defined in HTML may contain the precise action to take or it can call a script defined elsewhere on the page, as in this example:

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



```
</script>
<input type="button" value="Click Me" onclick="showMessage()" />
```

[HTMLEventHandlerExample01.htm](#)

In this code, the button calls `showMessage()` when it is clicked. The `showMessage()` function is defined in a separate `<script>` element and could also be included in an external file. Code executing as an event handler has access to everything in the global scope.

Event handlers assigned in this way have some unique aspects. First, a function is created that wraps the attribute value. That function has a special local variable called `event`, which is the event object (discussed later in this chapter):

```
<!-- outputs "click" -->
<input type="button" value="Click Me" onclick="alert(event.type)">
```

This gives you access to the event object without needing to define it yourself and without needing to pull it from the enclosing function's argument list.

The `this` value inside of the function is equivalent to the event's target element, for example:

```
<!-- outputs "Click Me" -->
<input type="button" value="Click Me" onclick="alert(this.value)">
```

Another interesting aspect of this dynamically created function is how it augments the scope chain. Within the function, members of both `document` and the element itself can be accessed as if they were local variables. The function accomplishes this via scope chain augmentation using `with`:

```
function(){
    with(document){
        with(this){
            //attribute value
        }
    }
}
```

This means that an event handler can access its own properties easily. The following is functionally the same as the previous example:

```
<!-- outputs "Click Me" -->
<input type="button" value="Click Me" onclick="alert(value)">
```

If the element is a form input element, then the scope chain also contains an entry for the parent form element, making the function the equivalent to the following:

```
function(){
    with(document){
        with(this.form){
            with(this){
                //attribute value
            }
        }
    }
}
```

}

Basically, this augmentation allows the event handler code to access other members of the same form without referencing the form element itself. For example:



```
<form method="post">
    <input type="text" name="username" value="">
    <input type="button" value="Echo Username" onclick="alert(username.value)">
</form>
```

HTMLEventHandlerExample04.htm

Clicking on the button in this example results in the text from the text box being displayed. Note that it just references `username` directly.

There are a few downsides to assigning event handlers in HTML. The first is a timing issue: it's possible that the HTML element appears on the page and is interacted with by the user before the event handler code is ready. In the previous example, imagine a scenario where the `showMessage()` function isn't defined until later on the page, after the code for the button. If the user were to click the button before `showMessage()` was defined, an error would occur. For this reason, most HTML event handlers are enclosed in `try-catch` blocks so that they quietly fail, as in the following example:

```
<input type="button" value="Click Me" onclick="try{showMessage();}catch(ex){}">
```

If this button is clicked before the `showMessage()` function is defined, no JavaScript error occurs because the error is caught before the browser can handle it.

Another downside is that the scope chain augmentation in the event handler function can lead to different results in different browsers. The rules being followed for identifier resolution are slightly different amongst JavaScript engines, and so the result of accessing unqualified object members may cause errors.

The last downside to assigning event handlers using HTML is that it tightly couples the HTML to the JavaScript. If the event handler needs to be changed, you may need to change code in two places: in the HTML and in the JavaScript. This is the primary reason that many developers avoid HTML event handlers in favor of using JavaScript to assign event handlers.



For more information on the disadvantages of HTML event handlers, please see Event Handler Scope by Garrett Smith (www.jibbering.com/faq/names/event_handler.html).

DOM Level 0 Event Handlers

The traditional way of assigning event handlers in JavaScript is to assign a function to an event handler property. This was the event handler assignment method introduced in the fourth generation of web browsers, and it still remains in all modern browsers because of its simplicity and cross-browser support. To assign an event handler using JavaScript, you must first retrieve a reference to the object to act on.

Each element (as well as `window` and `document`) has event handler properties that are typically all lowercase, such as `onclick`. An event handler is assigned by setting the property equal to a function, as in this example:

```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    alert("Clicked");
};
```

Here, a button is retrieved from the document and an `onclick` event handler is assigned. Note that the event handler isn't assigned until this code is run, so if the code appears after the code for the button in the page, there may be an amount of time during which the button will do nothing when clicked.

When assigning event handlers using the DOM Level 0 method, the event handler is considered to be a method of the element. The event handler, therefore, is run within the scope of element, meaning that this is equivalent to the element. Here is an example:



Available for download on
Wrox.com

```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    alert(this.id);      //"myBtn"
};
```

[DOMLevel0EventHandlerExample01.htm](#)

This code displays the element's ID when the button is clicked. The ID is retrieved using `this.id`. It's possible to use `this` to access any of the element's properties or methods from within the event handlers. Event handlers added in this way are intended for the bubbling phase of the event flow.

You can remove an event handler assigned via the DOM Level 0 approach by setting the value of the event handler property to `null`, as in the following example:

```
btn.onclick = null;      //remove event handler
```

Once the event handler is set to `null`, the button no longer has any action to take when it is clicked.



If you've assigned an event handler using HTML, the value on the `onclick` property is a function containing the code specified in the HTML attribute. These event handlers can also be removed by setting the property to `null`.

DOM Level 2 Event Handlers

DOM Level 2 Events define two methods to deal with the assignment and removal of event handlers: `addEventListener()` and `removeEventListener()`. These methods exist on all DOM nodes and accept three arguments: the event name to handle, the event handler function, and a Boolean value indicating whether to call the event handler during the capture phase (`true`) or during the bubble phase (`false`).

To add an event handler for the `click` event on a button, you can use the following code:

```
var btn = document.getElementById("myBtn");
btn.addEventListener("click", function(){
    alert(this.id);
}, false);
```

This code adds an `onclick` event handler to a button that will be fired in the bubbling phase (since the last argument is `false`). As with the DOM Level 0 approach, the event handler runs in the scope of the element on which it is attached. The major advantage to using the DOM Level 2 method for adding event handlers is that multiple event handlers can be added. Consider the following example:



Available for
download on
Wrox.com

```
var btn = document.getElementById("myBtn");
btn.addEventListener("click", function(){
    alert(this.id);
}, false);
btn.addEventListener("click", function(){
    alert("Hello world!");
}, false);
```

[DOMLevel2EventHandlerExample01.htm](#)

Here, two event handlers are added to the button. The event handlers fire in the order in which they were added, so the first alert displays the element's ID and the second displays the message "Hello world!"

Event handlers added via `addEventListener()` can be removed only by using `removeEventListener()` and passing in the same arguments as were used when the handler was added. This means that anonymous functions added using `addEventListener()` cannot be removed, as shown in this example:

```
var btn = document.getElementById("myBtn");
btn.addEventListener("click", function(){
    alert(this.id);
}, false);

//other code here

btn.removeEventListener("click", function(){ //won't work!
    alert(this.id);
}, false);
```

In this example, an anonymous function is added as an event handler using `addEventListener()`. The call to `removeEventListener()` looks like it's using the same arguments, but in reality, the second argument is a completely different function than the one used in `addEventListener()`. The event handler function passed into `removeEventListener()` must be the same one that was used in `addEventListener()`, as in this example:



Available for download on
Wrox.com

```
var btn = document.getElementById("myBtn");
var handler = function(){
    alert(this.id);
};

btn.addEventListener("click", handler, false);

//other code here

btn.removeEventListener("click", handler, false); //works!
```

[DOMLevel2EventHandlerExample02.htm](#)

This rewritten example works as expected because the same function is used for both `addEventListener()` and `removeEventListener()`.

In most cases, event handlers are added to the bubbling phase of the event flow since this offers the broadest possible cross-browser support. Attaching an event handler in the capture phase is best done if you need to intercept events before they reach their intended target. If this is not necessary, it's advisable to avoid event capturing.



DOM Level 2 event handlers are supported in Internet Explorer 9, Firefox, Safari, Chrome, and Opera.

Internet Explorer Event Handlers

Internet Explorer implements methods similar to the DOM called `attachEvent()` and `detachEvent()`. These methods accept the same two arguments: the event handler name and the event handler function. Since Internet Explorer 8 and earlier support only event bubbling, event handlers added using `attachEvent()` are attached on the bubbling phase.

To add an event handler for the `click` event on a button using `attachEvent()`, you can use the following code:

```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function(){
    alert("Clicked");
});
```

[IEEventHandlerExample01.htm](#)

Note that the first argument of `attachEvent()` is "onclick" as opposed to "click" in the DOM's `addEventListener()` method.

A major difference between using `attachEvent()` and using the DOM Level 0 approach in Internet Explorer is the scope of the event handler. When using DOM Level 0, the event handler runs with a `this` value equal to the element on which it is attached; when using `attachEvent()`, the event handler runs in the global context, so `this` is equivalent to `window`. Here is an example:

```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function(){
    alert(this === window); //true
});
```

This difference is important to understand when writing cross-browser code.

The `attachEvent()` method, similar to `addEventListener()`, can be used to add multiple event handlers to a single element. Consider the following example:



```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function(){
    alert("Clicked");
});
btn.attachEvent("onclick", function(){
    alert("Hello world!");
});
```

[IEEventHandlerExample01.htm](#)

Here, `attachEvent()` is called twice, adding two different event handlers to the same button. Unlike the DOM method, though, the event handlers fire in reverse of the order they were added. When the button in this example is clicked, the first alert says "Hello world!" and the second says "Clicked".

Events added using `attachEvent()` are removed using `detachEvent()` as long as the same arguments are provided. As with the DOM methods, this means that anonymous functions cannot be removed once they have been added. Event handlers can always be removed as long as a reference to the same function can be passed into `detachEvent()`. Here is an example:

```
var btn = document.getElementById("myBtn");
var handler = function(){
    alert("Clicked");
};
btn.attachEvent("onclick", handler);

//other code here

btn.detachEvent("onclick", handler);
```

[IEEventHandlerExample02.htm](#)

This example adds an event handler stored in the variable `handler`. That same function is later removed using `detachEvent()`.



Internet Explorer event handlers are supported in Internet Explorer and Opera.

Cross-Browser Event Handlers

To accommodate event handling in a cross-browser way, many developers end up either using a JavaScript library that abstracts away the browser differences or writing custom code to use the most appropriate event-handling approach. Writing your own code is fairly straightforward, because it relies on capability detection (covered in Chapter 9). To make sure that the event-handling code works in the most compatible way possible, you will need it to work only on the bubbling phase.

The first method to create is called `addHandler()`, and its job is to use the DOM Level 0 approach, the DOM Level 2 approach, or the Internet Explorer approach to adding events, depending on which is available. This method is attached to an object called `EventUtil` that will be used throughout this book to aid in handling cross-browser differences. The `addHandler()` method accepts three arguments: the element to act on, the name of the event, and the event handler function.

The counterpart to `addHandler()` is `removeHandler()`, which accepts the same three arguments. This method's job is to remove a previously added event handler using whichever means is available, defaulting to DOM Level 0 if no other method is available.

The full code for `EventUtil` is as follows:



Available for
download on
[Wrox.com](#)

```
var EventUtil = {

    addHandler: function(element, type, handler){
        if (element.addEventListener){
            element.addEventListener(type, handler, false);
        } else if (element.attachEvent){
            element.attachEvent("on" + type, handler);
        } else {
            element["on" + type] = handler;
        }
    },

    removeHandler: function(element, type, handler){
        if (element.removeEventListener){
            element.removeEventListener(type, handler, false);
        } else if (element.detachEvent){
            element.detachEvent("on" + type, handler);
        } else {
            element["on" + type] = null;
        }
    }
};
```

[EventUtil.js](#)

Both methods first check for the existence of the DOM Level 2 method on the element that was passed in. If the DOM Level 2 method exists, it is used, passing in the event type and the event

handler function, along with a third argument of `false` (to indicate the bubbling phase). If the Internet Explorer method is available, it is used as a second option. Note that the event type must be prefixed with "on" in order for it to work in Internet Explorer 8 and earlier. The last resort is to use the DOM Level 0 method (code should never reach here in modern browsers). Note the use of bracket notation to assign the property name to either the event handler or `null`.

This utility object can be used in the following way:



Available for
download on
Wrox.com

```
var btn = document.getElementById("myBtn");
var handler = function(){
    alert("Clicked");
};

EventUtil.addHandler(btn, "click", handler);

//other code here

EventUtil.removeHandler(btn, "click", handler);
```

CrossBrowserEventHandlerExample01.htm

The `addHandler()` and `removeHandler()` methods don't equalize all functionality across all browsers, such as the Internet Explorer scope issue, but it does allow the seamless addition and removal of event handlers. Keep in mind, also, that DOM Level 0 support is limited to just one event handler per event. Fortunately, DOM Level 0 browsers are no longer in popular use, so this shouldn't affect you.

THE EVENT OBJECT

When an event related to the DOM is fired, all of the relevant information is gathered and stored on an object called `event`. This object contains basic information such as the element that caused the event, the type of event that occurred, and any other data that may be relevant to the particular event. For example, an event caused by a mouse action generates information about the mouse's position, whereas an event caused by a keyboard action generates information about the keys that were pressed. All browsers support the `event` object, though not in the same way.

The DOM Event Object

In DOM-compliant browsers, the `event` object is passed in as the sole argument to an event handler. Regardless of the method used to assign the event handler, DOM Level 0 or DOM Level 2, the `event` object is passed in. Here is an example:

```
var btn = document.getElementById("myBtn");
btn.onclick = function(event){
    alert(event.type);      //"click"
};

btn.addEventListener("click", function(event){
    alert(event.type);      //"click"
}, false);
```

Both event handlers in this example pop up an alert indicating the type of event being fired by using the `event.type` property. This property always contains the type of event that was fired, such as "click" (it is the same value that you pass into `addEventListener()` and `removeEventListener()`).

When an event handler is assigned using HTML attributes, the `event` object is available as a variable called `event`. Here's an example:

```
<input type="button" value="Click Me" onclick="alert(event.type)">
```

Providing the `event` object in this way allows HTML attribute event handlers to perform the same as JavaScript functions.

The `event` object contains properties and methods related to the specific event that caused its creation. The available properties and methods differ based on the type of event that was fired, but all events have the members listed in the following table.

PROPERTY/METHOD	TYPE	READ/WRITE	DESCRIPTION
<code>bubbles</code>	Boolean	Read only	Indicates if the event bubbles.
<code>cancelable</code>	Boolean	Read only	Indicates if the default behavior of the event can be canceled.
<code>currentTarget</code>	Element	Read only	The element whose event handler is currently handling the event.
<code>defaultPrevented</code>	Boolean	Read only	When <code>true</code> , indicates that <code>preventDefault()</code> has been called (added in DOM Level 3 Events).
<code>detail</code>	Integer	Read only	Extra information related to the event.
<code>eventPhase</code>	Integer	Read only	The phase during which the event handler is being called: 1 for the capturing phase, 2 for "at target," and 3 for bubbling.
<code>preventDefault()</code>	Function	Read only	Cancels the default behavior for the event. If <code>cancelable</code> is <code>true</code> , this method can be used.
<code>stopImmediatePropagation()</code>	Function	Read only	Cancels any further event capturing or event bubbling and prevents any other event handlers from being called. (Added in DOM Level 3 Events.)

continues

(continued)

PROPERTY/METHOD	TYPE	READ/WRITE	DESCRIPTION
stopPropagation()	Function	Read only	Cancels any further event capturing or event bubbling. If bubbles is true, this method can be used.
target	Element	Read only	The target of the event.
trusted	Boolean	Read only	When true, indicates if the event was generated by the browser. When false, indicates the event was created using JavaScript by the developer. (Added in DOM Level 3 Events.)
type	String	Read only	The type of event that was fired.
view	AbstractView	Read only	The abstract view associated with the event. This is equal to the window object in which the event occurred.

Inside an event handler, the `this` object is always equal to the value of `currentTarget`, whereas `target` contains only the actual target of the event. If the event handler is assigned directly onto the intended target, then `this`, `currentTarget`, and `target` all have the same value. Here is an example:



```
var btn = document.getElementById("myBtn");
btn.onclick = function(event){
    alert(event.currentTarget === this);    //true
    alert(event.target === this);           //true
};
```

DOMEEventObjectExample01.htm

This code examines the values of `currentTarget` and `target` relative to `this`. Since the target of the `click` event is the button, all three are equal. If the event handler existed on a parent node of the button, such as `document.body`, the values would be different. Consider the following example:

```
document.body.onclick = function(event){
    alert(event.currentTarget === document.body);    //true
    alert(this === document.body);                   //true
    alert(event.target === document.getElementById("myBtn")); //true
};
```

DOMEEventObjectExample02.htm

When the button is clicked in this example, both `this` and `currentTarget` are equal to `document.body` because that's where the event handler was registered. The `target` property, however, is equal to the button element itself, because that's the true target of the `click` event. Since the button itself doesn't have an event handler assigned, the `click` event bubbles up to `document.body`, where the event is handled.

The `type` property is useful when you want to assign a single function to handle multiple events. Here is an example:



Available for
download on
Wrox.com

```
var btn = document.getElementById("myBtn");
var handler = function(event){
    switch(event.type){
        case "click":
            alert("Clicked");
            break;

        case "mouseover":
            event.target.style.backgroundColor = "red";
            break;

        case "mouseout":
            event.target.style.backgroundColor = "";
            break;
    }
};

btn.onclick = handler;
btn.onmouseover = handler;
btn.onmouseout = handler;
```

DOMEEventObjectExample03.htm

In this example, a single function called `handler` is defined to handle three different events: `click`, `mouseover`, and `mouseout`. When the button is clicked, it should pop up an alert, as in the previous examples. When the mouse is moved over the button, the background color should change to red, and when the mouse is moved away from the button, the background color should revert to its default. Using the `event.type` property, the function is able to determine which event occurred and then react appropriately.

The `preventDefault()` method is used to prevent the default action of a particular event. The default behavior of a link, for example, is to navigate to the URL specified in its `href` attribute when clicked. If you want to prevent that navigation from occurring, an `onclick` event handler can cancel that behavior, as in the following example:

```
var link = document.getElementById("myLink");
link.onclick = function(event) {
    event.preventDefault();
};
```

DOMEEventObjectExample04.htm

Any event that can be canceled using `preventDefault()` will have its `cancelable` property set to `true`.

The `stopPropagation()` method stops the flow of an event through the DOM structure immediately, canceling any further event capturing or bubbling before it occurs. For example, an event handler added directly to a button can call `stopPropagation()` to prevent an event handler on `document.body` from being fired, as shown in the following example:



Available for
download on
Wrox.com

```
var btn = document.getElementById("myBtn");
btn.onclick = function(event){
    alert("Clicked");
    event.stopPropagation();
};

document.body.onclick = function(event){
    alert("Body clicked");
};
```

[DOMEventObjectExample05.htm](#)

Without the call to `stopPropagation()` in this example, two alerts would be displayed when the button is clicked. However, the `click` event never reaches `document.body`, so the `onclick` event handler is never executed.

The `eventPhase` property aids in determining what phase of event flow is currently active. If the event handler is called during the capture phase, `eventPhase` is 1; if the event handler is at the target, `eventPhase` is 2; if the event handler is during the bubble phase, `eventPhase` is 3. Note that even though “at target” occurs during the bubbling phase, `eventPhase` is always 2. Here is an example:

```
var btn = document.getElementById("myBtn");
btn.onclick = function(event){
    alert(event.eventPhase); //2
};

document.body.addEventListener("click", function(event){
    alert(event.eventPhase); //1
}, true);

document.body.onclick = function(event){
    alert(event.eventPhase); //3
};
```

[DOMEventObjectExample06.htm](#)

When the button in this example is clicked, the first event handler to fire is the one on `document.body` in the capturing phase, which pops up an alert that displays 1 as the `eventPhase`. Next, event handler on the button itself is fired, at which point the `eventPhase` is 2. The last event handler to fire is during the bubbling phase on `document.body` when `eventPhase` is 3. Whenever `eventPhase` is 2, `this`, `target`, and `currentTarget` are always equal.



The `event` object exists only while event handlers are still being executed; once all event handlers have been executed, the `event` object is destroyed.

The Internet Explorer Event Object

Unlike the DOM event object, the Internet Explorer event object is accessible in different ways based on the way in which the event handler was assigned. When an event handler is assigned using the DOM Level 0 approach, the `event` object exists only as a property of the `window` object. Here is an example:

```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    var event = window.event;
    alert(event.type);      //"click"
};
```

Here, the `event` object is retrieved from `window.event` and then used to determine the type of event that was fired (the `type` property for Internet Explorer is identical to that of the DOM version). However, if the event handler is assigned using `attachEvent()`, the `event` object is passed in as the sole argument to the function, as shown here:

```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function(event){
    alert(event.type);      //"click"
});
```

When using `attachEvent()`, the `event` object is also available on the `window` object, as with the DOM Level 0 approach. It is also passed in as an argument for convenience.

If the event handler is assigned by an HTML attribute, the `event` object is available as a variable called `event` (the same as the DOM model). Here's an example:

```
<input type="button" value="Click Me" onclick="alert(event.type)">
```

The Internet Explorer event object also contains properties and methods related to the specific event that caused its creation. Many of these either map directly to or are related to DOM properties or methods. Like the DOM event object, the available properties and methods differ based on the type of event that was fired, but all events use the properties and methods defined in the following table.

PROPERTY/METHOD	TYPE	READ/WRITE	DESCRIPTION
<code>cancelBubble</code>	Boolean	Read/Write	<code>False</code> by default, but can be set to <code>true</code> to cancel event bubbling (same as the DOM <code>stopPropagation()</code> method).
<code>returnValue</code>	Boolean	Read/Write	<code>True</code> by default, but can be set to <code>false</code> to cancel the default behavior of the event (same as the DOM <code>preventDefault()</code> method).
<code>srcElement</code>	Element	Read only	The target of the event (same as the DOM <code>target</code> property).
<code>type</code>	String	Read only	The type of event that was fired.



```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    alert(window.event.srcElement === this);           //true
};

btn.attachEvent("onclick", function(event){
    alert(event.srcElement === this);                 //false
});
```

IEEventObjectExample01.htm

In the first event handler, which is assigned using the DOM Level 0 approach, the `srcElement` property is equal to `this`, but in the second event handler, the two values are different.

The `returnValue` property is the equivalent of the DOM `preventDefault()` method in that it cancels the default behavior of a given event. You need only set `returnValue` to `false` to prevent the default action. Consider the following example:

```
var link = document.getElementById("myLink");
link.onclick = function(){
    window.event.returnValue = false;
};
```

IEEventObjectExample02.htm

In this example, using `returnValue` in an `onclick` event handler stops a link's default action. Unlike the DOM, there is no way to determine whether an event can be canceled or not using JavaScript.

The `cancelBubble` property performs the same action as the DOM `stopPropagation()` method: it stops the event from bubbling. Since Internet Explorer 8 and earlier don't support the capturing phase, only bubbling is canceled, whereas `stopPropagation()` stops both capturing and bubbling. Here is an example:

```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    alert("Clicked");
    window.event.cancelBubble = true;
};

document.body.onclick = function(){
    alert("Body clicked");
};
```

IEEventObjectExample03.htm

By setting `cancelBubble` to `true` in the button's `onclick` event handler, it prevents the event from bubbling up to the `document.body` event handler. The result is that only one alert is displayed when the button is clicked.

The Cross-Browser Event Object

Although the event objects for the DOM and Internet Explorer are different, there are enough similarities to allow cross-browser solutions. All of the information and capabilities of the Internet Explorer event object are present in the DOM object, just in a different form. These parallels enable easy mapping from one event model to the other. The `EventUtil` object described earlier can be augmented with methods that equalize the differences:



Available for
download on
Wrox.com

```
var EventUtil = {

    addHandler: function(element, type, handler){
        //code removed for printing
    },

    getEvent: function(event){
        return event ? event : window.event;
    },

    getTarget: function(event){
        return event.target || event.srcElement;
    },

    preventDefault: function(event){
        if (event.preventDefault){
            event.preventDefault();
        } else {
            event.returnValue = false;
        }
    },

    removeHandler: function(element, type, handler){
        //code removed for printing
    },

    stopPropagation: function(event){
        if (event.stopPropagation){
            event.stopPropagation();
        } else {
            event.cancelBubble = true;
        }
    }
};
```

[EventUtil.js](#)

There are four new methods added to `EventUtil` in this code. The first is `getEvent()`, which returns a reference to the `event` object. Since the location of the `event` object differs in Internet Explorer, this method can be used to retrieve the `event` object regardless of the event handler assignment approach used. To use this method, you must assume that the `event` object is passed into the event handler and pass in that variable to the method. Here is an example:



```
btn.onclick = function(event){
    event = EventUtil.getEvent(event);
};
```

[CrossBrowserEventObjectExample01.htm](#)

When used in a DOM-compliant browser, the `event` variable is just passed through and returned. In Internet Explorer the `event` argument will be `undefined`, so `window.event` is returned. Adding this line to the beginning of event handlers ensures that the `event` object is always available, regardless of the browser being used.

The second method is `getTarget()`, which returns the target of the event. Inside the method, it checks the `event` object to see if the `target` property is available and returns its value if it is; otherwise, the `srcElement` property is used. This method can be used as follows:

```
btn.onclick = function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);
};
```

[CrossBrowserEventObjectExample01.htm](#)

The third method is `preventDefault()`, which stops the default behavior of an event. When the `event` object is passed in, it is checked to see if the `preventDefault()` method is available and, if so, calls it. If `preventDefault()` is not available, the method sets `returnValue` to `false`. Here is an example:

```
var link = document.getElementById("myLink");
link.onclick = function(event){
    event = EventUtil.getEvent(event);
    EventUtil.preventDefault(event);
};
```

[CrossBrowserEventObjectExample02.htm](#)

This code prevents a link click from navigating to another page in all major browsers. The `event` object is first retrieved using `EventUtil.getEvent()` and then passed into `EventUtil.preventDefault()` to stop the default behavior.

The fourth method, `stopPropagation()`, works in a similar way. It first tries to use the DOM method for stopping the event flow and uses `cancelBubble` if necessary. Here is an example:



Available for
download on
Wrox.com

```
var btn = document.getElementById("myBtn");
btn.onclick = function(event) {
    alert("Clicked");
    event = EventUtil.getEvent(event);
    EventUtil.stopPropagation(event);
};

document.body.onclick = function(event) {
    alert("Body clicked");
};
```

CrossBrowserEventObjectExample03.htm

Here, the event object is retrieved using `EventUtil.getEvent()` and then passed into `EventUtil.stopPropagation()`. Remember that this method may stop event bubbling or both event bubbling and capturing depending on the browser.

EVENT TYPES

There are numerous categories of events that can occur in a web browser. As mentioned previously, the type of event being fired determines the information that is available about the event. DOM Level 3 Events specifies the following event groups:

- **User interface (UI) events** are general browser events that may have some interaction with the BOM.
- **Focus events** are fired when an element gains or loses focus.
- Mouse events are fired when the mouse is used to perform an action on the page.
- **Wheel events** are fired when a mouse wheel (or similar device) is used.
- **Text events** are fired when text is input into the document.
- **Keyboard events** are fired when the keyboard is used to perform an action on the page.
- **Composition events** are fired when inputting characters for an Input Method Editor (IME).
- **Mutation events** are fired when a change occurs to the underlying DOM structure.
- **Mutation name events** are fired when element or attribute names are changed. These events are deprecated and not implemented by any browser, so they are intentionally omitted from this chapter.

In addition to these categories, HTML5 defines another set of events, and browsers often implement proprietary events both on the DOM and on the BOM. These proprietary events are typically driven by developer demand rather than specifications and so may be implemented differently across browsers.

DOM Level 3 Events redefines the event groupings from DOM Level 2 Events and adds additional event definitions. All major browsers support DOM Level 2 Events, including Internet Explorer 9. Internet Explorer 9 also supports DOM Level 3 Events.

UI Events

UI events are those events that aren't necessarily related to user actions. These events existed in some form or another prior to the DOM specification and were retained for backwards compatibility. The UI events are as follows:

- `DOMActivate` — Fires when an element has been activated by some user action, by either mouse or keyboard (more generic than `click` or `keydown`). This event is deprecated in DOM Level 3 Events and is supported in Firefox 2+ and Chrome. Because of cross-browser implementation differences, it's recommended not to use this event.
- `load` — Fires on a window when the page has been completely loaded, on a frameset when all frames have been completely loaded, on an `` element when it has been completely loaded, or on an `<object>` element when it has been completely loaded.
- `unload` — Fires on a window when the page has been completely unloaded, on a frameset when all frames have been completely unloaded, or on an `<object>` element when it has been completely unloaded.
- `abort` — Fires on an `<object>` element if it is not fully loaded before the user stops the download process.
- `error` — Fires on a window when a JavaScript error occurs, on an `` element if the image specified cannot be loaded, on an `<object>` element if it cannot be loaded, or on a frameset if one or more frames cannot be loaded. This event is discussed in Chapter 17.
- `select` — Fires when the user selects one or more characters in a text box (either `<input>` or `<textarea>`). This event is discussed in Chapter 14.
- `resize` — Fires on a window or frame when it is resized.
- `scroll` — Fires on any element with a scrollbar when the user scrolls it. The `<body>` element contains the scrollbar for a loaded page.

Most of the HTML events are related either to the `window` object or to form controls.

With the exception of `DOMActivate`, these events were part of the HTML Events group in DOM Level 2 Events (`DOMActivate` was still part of UI Events in DOM Level 2). To determine if a browser supports HTML events according to DOM Level 2 Events, you can use the following code:

```
var isSupported = document.implementation.hasFeature("HTMLEvents", "2.0");
```

Note that browsers should return `true` for this only if they implement these events according to the DOM Level 2 Events. Browsers may support these events in nonstandard ways and thus return `false`. To determine if the browser supports these events as defined in DOM Level 3 Events, use the following:

```
var isSupported = document.implementation.hasFeature("UIEvent", "3.0");
```

The load Event

The `load` event is perhaps the most often used event in JavaScript. For the `window` object, the `load` event fires when the entire page has been loaded, including all external resources such as images,

JavaScript files, and CSS files. You can define an `onload` event handler in two ways. The first is by using JavaScript, as shown here:



```
EventUtil.addHandler(window, "load", function(event) {
    alert("Loaded!");
});
```

[LoadEventExample01.htm](#)

This is the JavaScript-based way of assigning an event handler, using the cross-browser `EventUtil` object discussed earlier in this chapter. As with other events, the `event` object is passed into the event handler. The `event` object doesn't provide any extra information for this type of event, although it's interesting to note that DOM-compliant browsers have `event.target` set to `document`, whereas Internet Explorer prior to version 8 doesn't set the `srcElement` property for this event.

The second way to assign the `onload` event handler is to add an `onload` attribute to the `<body>` element, as in the following example:

```
<!DOCTYPE html>
<html>
<head>
    <title>Load Event Example</title>
</head>
<body onload="alert('Loaded!')">

</body>
</html>
```

[LoadEventExample02.htm](#)

Generally speaking, any events that occur on the `window` can be assigned via attributes on the `<body>` element, because there is no access to the `window` element in HTML. This really is a hack for backwards compatibility but is still well-supported in all browsers. It is recommended that you use the JavaScript approach whenever possible.



According to DOM Level 2 Events, the `load` event is supposed to fire on `document`, not on `window`. However, `load` is implemented on `window` in all browsers for backwards compatibility.

The `load` event also fires on images, both those that are in the DOM and those that are not. You can assign an `onload` event handler directly using HTML on any images in the document, using code such as this:

```

```

[LoadEventExample03.htm](#)



```
var image = document.getElementById("myImage");
EventUtil.addHandler(image, "load", function(event) {
    event = EventUtil.getEvent(event);
    alert(EventUtil.getTarget(event).src);
});
```

[LoadEventExample04.htm](#)

Here, the `onload` event handler is assigned using JavaScript. The `event` object is passed in, though it doesn't have much useful information. The target of the event is the `` element, so its `src` property can be accessed and displayed.

When creating a new `` element, an event handler can be assigned to indicate when the image has been loaded. In this case, it's important to assign the event before assigning the `src` property, as in the following example:

```
EventUtil.addHandler(window, "load", function(){
    var image = document.createElement("img");
    EventUtil.addHandler(image, "load", function(event) {
        event = EventUtil.getEvent(event);
        alert(EventUtil.getTarget(event).src);
    });
    document.body.appendChild(image);
    image.src = "smile.gif";
});
```

[LoadEventExample05.htm](#)

The first part of this example is to assign an `onload` event handler for the window. Since the example involves adding a new element to the DOM, you must be certain that the page is loaded, because trying to manipulate `document.body` prior to its being fully loaded can cause errors. A new `image` element is created and its `onload` event handler is set. Then, the image is added to the page and its `src` is assigned. Note that the element need not be added to the document for the image download to begin; it begins as soon as the `src` property is set.

This same technique can be used with the DOM Level 0 `Image` object. Prior to the DOM, the `Image` object was used to preload images on the client. It can be used the same way as an `` element with the exception that it cannot be added into the DOM tree. Consider the following example:

```
EventUtil.addHandler(window, "load", function(){
    var image = new Image();
    EventUtil.addHandler(image, "load", function(event) {
        alert("Image loaded!");
    });
    image.src = "smile.gif";
});
```

[LoadEventExample06.htm](#)

Here, the `Image` constructor is used to create a new image and the event handler is assigned. Some browsers implement the `Image` object as an `` element, but not all, so it's best to treat them as separate.



Internet Explorer 8 and earlier versions don't generate an `event` object when the `load` event fires for an image that isn't part of the DOM document. This pertains both to `` elements that are never added to the document and to the `Image` object. This was fixed in Internet Explorer 9.

There are other elements that also support the `load` event in nonstandard ways. The `<script>` element fires a `load` event in Internet Explorer 9+, Firefox, Opera, Chrome, and Safari 3+, allowing you to determine when dynamically loaded JavaScript files have been completely loaded. Unlike images, JavaScript files start downloading only after the `src` property has been assigned and the element has been added into the document, so the order in which the event handler and the `src` property are assigned is insignificant. The following illustrates how to assign an event handler for a `<script>` element:



Available for download on
Wrox.com

```
EventUtil.addHandler(window, "load", function(){
    var script = document.createElement("script");
    script.type = "text/javascript";
    EventUtil.addHandler(script, "load", function(event){
        alert("Loaded");
    });
    script.src = "example.js";
    document.body.appendChild(script);
});
```

[LoadEventExample07.htm](#)

This example uses the cross-browser `EventUtil` object to assign the `onload` event handler to a newly created `<script>` element. The event object's target is the `<script>` node in most browsers. Internet Explorer 8 and earlier versions do not support the `load` event for `<script>` elements.

Internet Explorer and Opera support the `load` event for `<link>` elements, allowing you to determine when a style sheet has been loaded. For example:

```
EventUtil.addHandler(window, "load", function(){
    var link = document.createElement("link");
    link.type = "text/css";
    link.rel= "stylesheet";
    EventUtil.addHandler(link, "load", function(event){
        alert("css loaded");
    });
    link.href = "example.css";
    document.getElementsByTagName("head")[0].appendChild(link);
});
```

[LoadEventExample07.htm](#)

As with the `<script>` node, a style sheet does not begin downloading until the `href` property has been assigned and the `<link>` element has been added to the document.

The unload Event

A companion to the `load` event, the `unload` event fires when a document has completely unloaded. The `unload` event typically fires when navigating from one page to another and is most often used to clean up references to avoid memory leaks. Similar to the `load` event, an `onunload` event handler can be assigned in two ways. The first is by using JavaScript as shown here:

```
EventUtil.addHandler(window, "unload", function(event) {
    alert("Unloaded!");
});
```

The `event` object is generated for this event but contains nothing more than the `target` (set to `document`) in DOM-compliant browsers. Internet Explorer 8 and earlier versions don't provide the `srcElement` property for this event.

The second way to assign the event handler, similar to the `load` event, is to add an attribute to the `<body>` element, as in this example:



```
<!DOCTYPE html>
<html>
<head>
    <title>Unload Event Example</title>
</head>
<body onunload="alert('Unloaded!')">

</body>
</html>
```

UnloadEventExample01.htm

Regardless of the approach you use, be careful with the code that executes inside of an `onunload` event handler. Since the `unload` event fires after everything is unloaded, not all objects that were available when the page was loaded are still available. Trying to manipulate the location of a DOM node or its appearance can result in errors.



According to DOM Level 2 Events, the `unload` event is supposed to fire on `<body>`, not on `window`. However, `unload` is implemented on `window` in all browsers for backwards compatibility.

The resize Event

When the browser window is resized to a new height or width, the `resize` event fires. This event fires on `window`, so an event handler can be assigned either via JavaScript or by using the `onresize`

attribute on the `<body>` element. As mentioned previously, it is recommended that you use the JavaScript approach as shown here:

```
EventUtil.addHandler(window, "resize", function(event) {
    alert("Resized");
});
```

Similar to other events that occur on the `window`, the `event` object is created and its `target` is `document` in DOM-compliant browsers, whereas Internet Explorer 8 and earlier provide no properties of use.

There are some important differences as to when the `resize` events fire across browsers. Internet Explorer, Safari, Chrome, and Opera fire the `resize` event as soon as the browser is resized by one pixel and then repeatedly as the user resizes the browser window. Firefox fires the `resize` event only after the user has stopped resizing the browser. Because of these differences, you should avoid computation-heavy code in the event handler for this event, because it will be executed frequently and cause a noticeable slowdown in the browser.



The `resize` event also fires when the browser window is minimized or maximized.

The scroll Event

Even though the `scroll` event occurs on the `window`, it actually refers to changes in the appropriate page-level element. In quirks mode, the changes are observable using the `scrollLeft` and `scrollTop` of the `<body>` element; in standards mode, the changes occur on the `<html>` element in all browsers except Safari (which still tracks scroll position on `<body>`). For example:



```
EventUtil.addHandler(window, "scroll", function(event) {
    if (document.compatMode == "CSS1Compat") {
        alert(document.documentElement.scrollTop);
    } else {
        alert(document.body.scrollTop);
    }
});
```

[ScrollEventExample01.htm](#)

This code assigns an event handler that outputs the vertical scroll position of the page, depending on the rendering mode. Since Safari prior to 3.1 doesn't support `document.compatMode`, older versions fall through to the second case.

- Similar to `resize`, the `scroll` event occurs repeatedly as the document is being scrolled, so it's best to keep the event handlers as simple as possible.

Focus Events

Focus events are fired when elements of a page receive or lose focus. These events work in concert with the `document.hasFocus()` and `document.activeElement` properties to give insight as to how the user is navigating the page. There are six focus events:

- `blur` — Fires when an element has lost focus. This event does not bubble and is supported in all browsers.
- `DOMFocusIn` — Fires when an element has received focus. This is a bubbling version of the `focus` HTML event. Opera is the only major browser to support this event. DOM Level 3 Events deprecates `DOMFocusIn` in favor of `focusin`.
- `DOMFocusOut` — Fires when an element has lost focus. This is a generic version of the `blur` HTML event. Opera is the only major browser to support this event. DOM Level 3 Events deprecates `DOMFocusOut` in favor of `focusout`.
- `focus` — Fires when an element has received focus. This event does not bubble and is supported in all browsers.
- `focusin` — Fires when an element has received focus. This is a bubbling version of the `focus` HTML event and is supported in Internet Explorer 5.5+, Safari 5.1+, Opera 11.5+, and Chrome.
- `focusout` — Fires when an element has lost focus. This is a generic version of the `blur` HTML event and is supported in Internet Explorer 5.5+, Safari 5.1+, Opera 11.5+, and Chrome.

The two primary events of this group are `focus` and `blur`, both of which have been supported in browsers since the early days of JavaScript. One of the biggest issues with these events is that they don't bubble. This led to the inclusion of `focusin` and `focusout` by Internet Explorer and `DOMFocusIn` and `DOMFocusOut` by Opera. Internet Explorer's approach has been standardized in DOM Level 3 Events.

When focus is moved from one element to another on the page, the following order of events is followed:

1. `focusout` fires on the element losing focus.
2. `focusin` fires on the element receiving focus.
3. `blur` fires on the element losing focus.
4. `DOMFocusOut` fires on the element losing focus.
5. `focus` fires on the element receiving focus.
6. `DOMFocusIn` fires on the element receiving focus.

The event target for `blur`, `DOMFocusOut`, and `focusout` is the element losing focus while the event target for `focus`, `DOMFocusIn`, and `focusin` is the element receiving focus.

You can determine if a browser supports these events with the following:

```
var isSupported = document.implementation.hasFeature("FocusEvent", "3.0");
```



Even though **focus** and **blur** don't bubble, they can be listened for during the capturing phase. Peter-Paul Koch has an excellent write-up on this topic at www.quirksmode.org/blog/archives/2008/04/delegating_the.html.

Mouse and Wheel Events

Mouse events are the most commonly used group of events on the Web, because the mouse is the primary navigation device used. There are nine mouse events defined in DOM Level 3 Events. They are as follows:

- `click` — Fires when the user clicks the primary mouse button (typically the left button) or when the user presses the Enter key. This is an important fact for accessibility purposes, because `onclick` event handlers can be executed using the keyboard and the mouse.
- `dblclick` — Fires when the user double-clicks the primary mouse button (typically the left button). This event was not defined in DOM Level 2 Events but is well-supported and so was standardized in DOM Level 3 Events.
- `mousedown` — Fires when the user pushes any mouse button down. This event cannot be fired via the keyboard.
- `mouseenter` — Fires when the mouse cursor is outside of an element and then the user first moves it inside of the boundaries of the element. This event does not bubble and does not fire when the cursor moves over descendant elements. The `mouseenter` event was not defined in DOM Level 2 Events but was added in DOM Level 3 Events. Internet Explorer, Firefox 9+, and Opera support this event.
- `mouseleave` — Fires when the mouse cursor is over an element and then the user moves it outside of that element's boundaries. This event does not bubble and does not fire when the cursor moves over descendant elements. The `mouseleave` event was not defined in DOM Level 2 Events but was added in DOM Level 3 Events. Internet Explorer, Firefox 9+, and Opera support this event.
- `mousemove` — Fires repeatedly as the cursor is being moved around an element. This event cannot be fired via the keyboard.
- `mouseout` — Fires when the mouse cursor is over an element and then the user moves it over another element. The element moved to may be outside of the bounds of the original element or a child of the original element. This event cannot be fired via the keyboard.
- `mouseover` — Fires when the mouse cursor is outside of an element and then the user first moves it inside of the boundaries of the element. This event cannot be fired via the keyboard.
- `mouseup` — Fires when the user releases a mouse button. This event cannot be fired via the keyboard.

All elements on a page support mouse events. All mouse events bubble except `mouseenter` and `mouseleave`, and they can all be canceled, which affects the default behavior of the browser.

Cancelling the default behavior of mouse events can affect other events as well because of the relationship that exists amongst the events.

A `click` event can be fired only if a `mousedown` event is fired and followed by a `mouseup` event on the same element; if either `mousedown` or `mouseup` is canceled, then the `click` event will not fire. Similarly, it takes two `click` events to cause the `dblclick` event to fire. If anything prevents these two `click` events from firing (either canceling one of the `click` events or canceling either `mousedown` or `mouseup`), the `dblclick` event will not fire. These four mouse events always fire in the following order:

- 1.** `mousedown`
- 2.** `mouseup`
- 3.** `click`
- 4.** `mousedown`
- 5.** `mouseup`
- 6.** `click`
- 7.** `dblclick`

Both `click` and `dblclick` rely on other events to fire before they can fire, whereas `mousedown` and `mouseup` are not affected by other events.

Internet Explorer through version 8 has a slight implementation bug that causes the second `mousedown` and `click` events to be skipped during a double click. The order is:

- 1.** `mousedown`
- 2.** `mouseup`
- 3.** `click`
- 4.** `mouseup`
- 5.** `dblclick`

Internet Explorer 9 fixes this bug so the event ordering is correct.

You can determine if the DOM Level 2 Events (those listed above excluding `dblclick`, `mouseenter`, and `mouseleave`) are supported by using this code:

```
var isSupported = document.implementation.hasFeature("MouseEvents", "2.0");
```

To determine if the browser supports all of the events listed above, use the following:

```
var isSupported = document.implementation.hasFeature("MouseEvent", "3.0")
```

Note that the DOM Level 3 feature name is just `"MouseEvent"` instead of `"MouseEvents"`.

There is also a subgroup of mouse events called *wheel events*. Wheel events are really just a single event, `mousewheel`, which monitors interactions of a mouse wheel or a similar device such as the Mac trackpad.

Client Coordinates

Mouse events all occur at a particular location within the browser viewport. This information is stored in the `clientX` and `clientY` properties of the event object. These properties indicate the location of the mouse cursor within the viewport at the time of the event and are supported in all browsers. Figure 13-4 illustrates the *client coordinates* in a viewport.

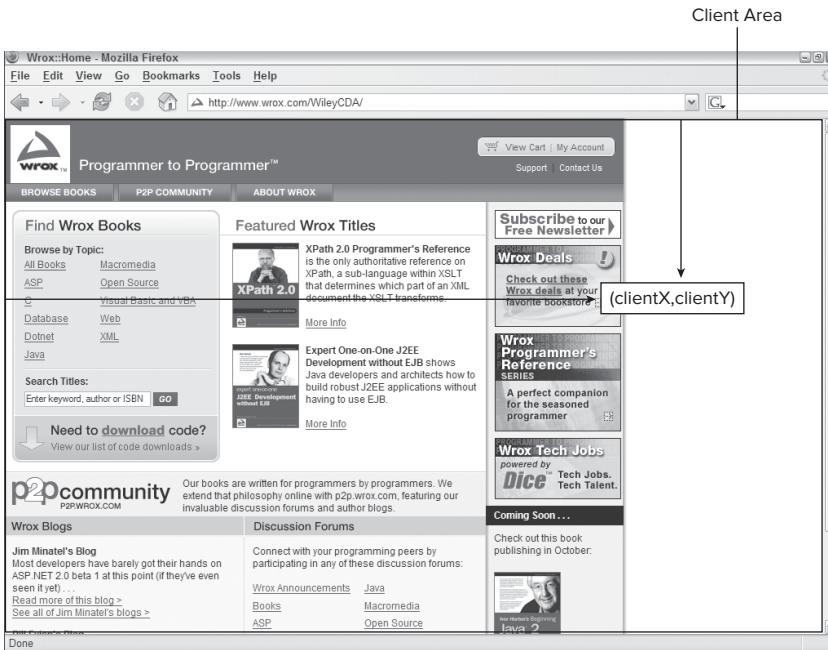


FIGURE 13-4

You can retrieve the client coordinates of a mouse event in the following way:



```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event) {
    event = EventUtil.getEvent(event);
    alert("Client coordinates: " + event.clientX + "," + event.clientY);
});
```

[ClientCoordinatesExample01.htm](#)

This example assigns an `onclick` event handler to a `<div>` element. When the element is clicked, the client coordinates of the event are displayed. Keep in mind that these coordinates do not take into account the scroll position of the page, so these numbers do not indicate the location of the cursor on the page.

Page Coordinates

Where client coordinates give you information about where an event occurred in the viewport, *page coordinates* tell you where on the page the event occurred via the `pageX` and `pageY` properties of the event object. These properties indicate the location of the mouse cursor on the page, so the coordinates are from the left and top of the page itself rather than the viewport.

You can retrieve the page coordinates of a mouse event in the following way:



```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event) {
    event = EventUtil.getEvent(event);
    alert("Page coordinates: " + event.pageX + "," + event.pageY);
});
```

[PageCoordinatesExample01.htm](#)

The values for `pageX` and `pageY` are the same as `clientX` and `clientY` when the page is not scrolled.

Internet Explorer 8 and earlier don't support page coordinates on the `event` object, but you can calculate them using client coordinates and scrolling information. You need to use the `scrollLeft` and `scrollTop` properties on either `document.body` (when in quirks mode) or `document.documentElement` (in standards mode). The calculation is done as follows:

```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event) {
    event = EventUtil.getEvent(event);
    var pageX = event.pageX,
        pageY = event.pageY;

    if (pageX === undefined){
        pageX = event.clientX + (document.body.scrollLeft ||
            document.documentElement.scrollLeft);
    }

    if (pageY === undefined){
        pageY = event.clientY + (document.body.scrollTop ||
            document.documentElement.scrollTop);
    }

    alert("Page coordinates: " + pageX + "," + pageY);
});
```

[PageCoordinatesExample01.htm](#)

Screen Coordinates

Mouse events occur not only in relation to the browser window but also in relation to the entire screen. It's possible to determine the location of the mouse in relation to the entire screen by using the `screenX` and `screenY` properties. Figure 13-5 illustrates the *screen coordinates* in a browser.



FIGURE 13-5

You can retrieve the screen coordinates of a mouse event in the following way:



Available for download on
Wrox.com

```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event){
    event = EventUtil.getEvent(event);
    alert("Screen coordinates: " + event.screenX + "," + event.screenY);
});
```

[ScreenCoordinatesExample01.htm](#)

Similar to the previous examples, this code assigns an `onclick` event handler to a `<div>` element. When the element is clicked, the screen coordinates of the event are displayed.

Modifier Keys

Even though a mouse event is primarily triggered by using the mouse, the state of certain keyboard keys may be important in determining the action to take. The *modifier keys* Shift, Ctrl, Alt, and Meta are often used to alter the behavior of a mouse event. The DOM specifies four properties to indicate the state of these modifier keys: `shiftKey`, `ctrlKey`, `altKey`, and `metaKey`. Each of these properties contains a Boolean value that is set to `true` if the key is being held down or `false` if the key is not pressed. When a mouse event occurs, you can determine the state of the various keys by inspecting these properties. Consider the following example:

```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event) {
    event = EventUtil.getEvent(event);
```

```

var keys = new Array();

if (event.shiftKey){
    keys.push("shift");
}

if (event.ctrlKey){
    keys.push("ctrl");
}

if (event.altKey){
    keys.push("alt");
}

if (event.metaKey){
    keys.push("meta");
}

alert("Keys: " + keys.join(","));
});

```

[ModifierKeysExample01.htm](#)

In this example, an `onclick` event handler checks the state of the various modifier keys. The `keys` array contains the names of the modifier keys that are being held down. For each property that is `true`, the name of the key is added to `keys`. At the end of the event handler, the `keys` are displayed in an alert.



*Internet Explorer 9, Firefox, Safari, Chrome, and Opera support all four keys.
Internet Explorer 8 and earlier versions do not support the `metaKey` property.*

Related Elements

For the `mouseover` and `mouseout` events, there are other elements related to the event. Both of these events involve moving the mouse cursor from within the boundaries of one element to within the boundaries of another element. For the `mouseover` event, the primary target of the event is the element that is gaining the cursor, and the related element is the one that is losing the cursor. Likewise, for `mouseout`, the primary target is the element that is losing the cursor, and the related element is the one that is gaining the cursor. Consider the following example:



```

<!DOCTYPE html>
<html>
<head>
    <title>Related Elements Example</title>
</head>
<body>
    <div id="myDiv" style="background-color:red;height:100px;width:100px;"></div>

```

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

RelatedElementsExample01.htm

This page renders a single `<div>` on the page. If the mouse cursor starts over the `<div>` and then moves outside of it, a `mouseout` event fires on `<div>` and the related element is the `<body>` element. Simultaneously, the `mouseover` event fires on `<body>` and the related element is the `<div>`.

The DOM provides information about related elements via the `relatedTarget` property on the event object. This property contains a value only for the `mouseover` and `mouseout` events; it is `null` for all other events. Internet Explorer 8 and earlier don't support the `relatedTarget` property but offer comparable access to the related element using other properties. When the `mouseover` event fires, Internet Explorer provides a `fromElement` property containing the related element; when the `mouseout` event fires, Internet Explorer provides a `toElement` property containing the related element (Internet Explorer 9 supports all properties). A cross-browser method to get the related element can be added to `EventUtil` like this:



Available for
download on
Wrox.com

```
var EventUtil = {
    //more code here

    getRelatedTarget: function(event) {
        if (event.relatedTarget){
            return event.relatedTarget;
        } else if (event.toElement){
            return event.toElement;
        } else if (event.fromElement){
            return event.fromElement;
        } else {
            return null;
        }
    },
    //more code here
};


```

EventUtil.js

As with the previous cross-browser methods, this one uses feature detection to determine which value to return. The `EventUtil.getRelatedTarget()` method can then be used as follows:

```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "mouseout", function(event) {
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);
    var relatedTarget = EventUtil.getRelatedTarget(event);
    alert("Moused out of " + target.tagName + " to " + relatedTarget.tagName);
});
```

RelatedElementsExample01.htm

This example registers an event handler for the `mouseout` event on the `<div>` element. When the event fires, an alert is displayed indicating the place the mouse moved from and the place the mouse moved to.

Buttons

The `click` event is fired only when the primary mouse button is clicked on an element (or when the Enter key is pressed on the keyboard), so button information isn't necessary. For the `mousedown` and `mouseup` events, there is a `button` property on the event object that indicates the button that was pressed or released. The DOM `button` property has the following three possible values: 0 for the primary mouse button, 1 for the middle mouse button (usually the scroll wheel button), and 2 for the secondary mouse button. In traditional setups, the primary mouse button is the left button and the secondary button is the right one.

Internet Explorer through version 8 also provides a `button` property, but it has completely different values, as described here:

- 0 indicates that no button has been pressed.
- 1 indicates that the primary mouse button has been pressed.
- 2 indicates that the secondary mouse button has been pressed.
- 3 indicates that the primary and secondary buttons have been pressed.
- 4 indicates that the middle button has been pressed.
- 5 indicates that the primary and middle buttons have been pressed.
- 6 indicates that the secondary and middle buttons have been pressed.
- 7 indicates that all three buttons have been pressed.

As you can tell, the DOM model for the `button` property is much simpler and arguably more useful than the Internet Explorer model since multi-button mouse usage is rare. It's typical to normalize the models to the DOM way since all browsers except Internet Explorer 8 and earlier implement it natively. The mapping of primary, middle, and secondary buttons is fairly straightforward; all of the other Internet Explorer options will translate into the pressing of one of the buttons, giving precedence to the primary button in all instances. So if Internet Explorer returns either 5 or 7, this converts to 0 in the DOM model.

Since capability detection alone can't be used to determine the difference (since both have a `button` property), you must use another method. Browsers that support the DOM version of mouse events can be detected using the `hasFeature()` method, so a normalizing `getButton()` method on `EventUtil` can be written as follows:



```
var EventUtil = {
    //more code here
    getButton: function(event){
        if (document.implementation.hasFeature("MouseEvents", "2.0")){
            return event.button;
        }
    }
}
```

```

        } else {
            switch(event.button) {
                case 0:
                case 1:
                case 3:
                case 5:
                case 7:
                    return 0;
                case 2:
                case 6:
                    return 2;
                case 4:
                    return 1;
            }
        },
    },
    //more code here
};


```

EventUtil.js

Checking for the feature "MouseEvents" determines if the `button` property that is already present on `event` contains the correct values. If that test fails, then the browser is likely Internet Explorer and the values must be normalized. This method can then be used as follows:



Available for
download on
Wrox.com

```

var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "mousedown", function(event){
    event = EventUtil.getEvent(event);
    alert(EventUtil.getButton(event));
});

```

ButtonExample01.htm

In this example, an `onmousedown` event handler is added to a `<div>` element. When a mouse button is pressed on the element, an alert displays the code for the button.



Note that when used with an `onmouseup` event handler, the value of `button` is the button that was just released.

Additional Event Information

The DOM Level 2 Events specification provides the `detail` property on the `event` object to give additional information about an event. For mouse events, `detail` contains a number indicating how many times a click has occurred at the given location. Clicks are considered to be a `mousedown` event followed by a `mouseup` event at the same pixel location. The value of `detail` starts at 1 and is incremented every time a click occurs. If the mouse is moved between `mousedown` and `mouseup`, then `detail` is set back to 0.

Internet Explorer provides the following additional information for each mouse event as well:

- `altLeft` is a Boolean value indicating if the left Alt key is pressed. If `altLeft` is true then `altKey` is also true.
- `ctrlLeft` is a Boolean value indicating if the left Ctrl key is pressed. If `ctrlLeft` is true then `ctrlKey` is also true.
- `offsetX` is the x-coordinate of the cursor relative to the boundaries of the target element.
- `offsetY` is the y-coordinate of the cursor relative to the boundaries of the target element.
- `shiftLeft` is a Boolean value indicating if the left Shift key is pressed. If `shiftLeft` is true, then `shiftKey` is also true.

These properties are of limited value because they are available only in Internet Explorer and provide information that either is not necessary or can be calculated in other ways.

The mousewheel Event

Internet Explorer 6 first implemented the `mousewheel` event. Since that time, it has been picked up by Opera, Chrome, and Safari. The `mousewheel` event fires when the user interacts with the mouse wheel, rolling it vertically in either direction. This event fires on each element and bubbles up to `document` (in Internet Explorer 8) and `window` (in Internet Explorer 9+, Opera, Chrome, and Safari). The event object for the `mousewheel` event contains all standard information about mouse events and an additional property called `wheelDelta`. When the mouse wheel is rolled toward the front of the mouse, `wheelDelta` is a positive multiple of 120; when the mouse wheel is rolled toward the rear of the mouse, `wheelDelta` is a negative multiple of 120. See Figure 13-6.

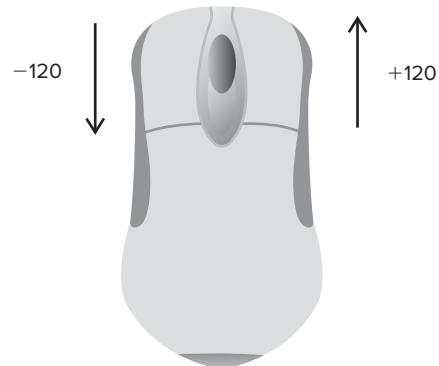


FIGURE 13-6

An `onmousewheel` event handler can be assigned to any element on the page or to the document to handle all mouse wheel interactions. Here's an example:

```
EventUtil.addHandler(document, "mousewheel", function(event) {
    event = EventUtil.getEvent(event);
    alert(event.wheelDelta);
});
```

This example simply displays the `wheelDelta` value when the event is fired. In most cases, you need only know which direction the mouse wheel was turned, which can easily be determined by the sign of the `wheelDelta` value.

One thing to be careful of: in Opera prior to version 9.5, the values for `wheelDelta` are reversed. If you plan on supporting earlier versions of Opera, you'll need to use browser detection to determine the actual value, as shown in the following example:



```
EventUtil.addHandler(document, "mousewheel", function(event) {
    event = EventUtil.getEvent(event);
    var delta = (client.engine.opera && client.engine.opera < 9.5 ?
        -event.wheelDelta : event.wheelDelta);
    alert(delta);
});
```

[MouseWheelEventExample01.htm](#)

This code uses the `client` object created in Chapter 9 to see if the browser is an earlier version of Opera.



The `mousewheel` event was added to HTML5 as a reflection of its popularity and availability in most browsers.

Firefox supports a similar event called `DOMMouseScroll`, which fires when the mouse wheel is turned. As with `mousewheel`, this event is considered a mouse event and has all of the usual mouse event properties. Information about the mouse wheel is given in the `detail` property, which is a negative multiple of three when the scroll wheel is rolled toward the front of the mouse and a positive multiple of three when it's rolled toward the back of the mouse. See Figure 13-7.

The `DOMMouseScroll` event can be attached to any element on the page and bubbles up to the `window` object. You can attach an event handler, as shown in the following example:

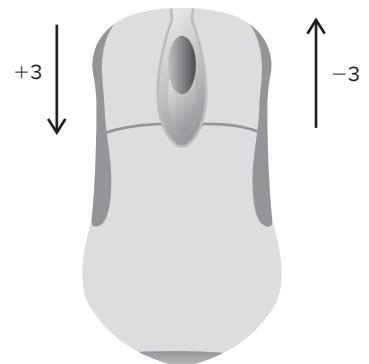


FIGURE 13-7

```
EventUtil.addHandler(window, "DOMMouseScroll", function(event) {
    event = EventUtil.getEvent(event);
    alert(event.detail);
});
```

[DOMMouseScrollEventExample01.htm](#)

This simple event handler outputs the value of the `detail` property each time the mouse wheel is scrolled.

For a cross-browser solution, the first step is to create a method that can retrieve a normalized value for the mouse wheel delta. This can be added to the `EventUtil` object as follows:

```
var EventUtil = {
    //more code here

    getWheelDelta: function(event){
        if (event.wheelDelta){
            return (client.engine.opera && client.engine.opera < 9.5 ?
                -event.wheelDelta : event.wheelDelta);
        }
    }
};
```

```

        } else {
            return -event.detail * 40;
        }
    },
    //more code here
};

```

[EventUtil.js](#)

The `getWheelDelta()` method checks to see if the `event` object has a `wheelDelta` property and, if so, uses the browser detecting code to determine the correct value. If `wheelDelta` doesn't exist, then it assumes the value is in the `detail` property. Since Firefox's value is different, it is first negated and then multiplied by 40 to be certain that its value will be the same as other browsers. With this method complete, you can assign the same event handler to both `mousewheel` and `DOMMouseScroll`, as shown here:



Available for
download on
[Wrox.com](http://www.wowebook.com)

```

(function(){

    function handleMouseWheel(event){
        event = EventUtil.getEvent(event);
        var delta = EventUtil.getWheelDelta(event);
        alert(delta);
    }

    EventUtil.addHandler(document, "mousewheel", handleMouseWheel);
    EventUtil.addHandler(document, "DOMMouseScroll", handleMouseWheel);

})();

```

[CrossBrowserMouseWheelExample01.htm](#)

This code exists within a private scope so as not to pollute the global scope with extra functions. The `handleMouseWheel()` function is the event handler for both events. (The event handler assignment quietly fails when assigned to an event that doesn't exist.) Using the `EventUtil.getWheelDelta()` method allows the event handler to work seamlessly in both cases.

Touch Device Support

Touch devices running iOS or Android have interesting implementations, because, of course, there is no mouse to interact with. When developing for touch devices, keep the following in mind:

- The `dblclick` event is not supported at all. Double-clicking on the browser window zooms in, and there is no way to override that behavior.
- Tapping on a clickable element causes the `mousemove` event to fire. If content changes as a result of this action, no further events are fired; if there are no changes to the screen, then the `mousedown`, `mouseup`, and `click` events fire in order. No events are fired when tapping on a nonclickable element. Clickable elements are defined as those that have a default action when clicked (such as links) or elements that have an `onclick` event handler assigned.

- The `mousemove` event also fires `mouseover` and `mouseout` events.
- The `mousewheel` and `scroll` events fire when two fingers are on the screen and the page is scrolled as the result of finger movement.

Accessibility Issues

If your web application or website must be accessible to users with disabilities, specifically those who are using screen readers, you should be careful when using mouse events. As mentioned previously, the `click` event can be fired using the Enter key on the keyboard, but other mouse events have no keyboard support. It's advisable not to use mouse events other than `click` to show functionality or cause code execution, as this will severely limit the usability for blind or sight-impaired users. Here are some tips for accessibility using mouse events:

- Use `click` to execute code. Some suggest that an application feels faster when code is executed using `onmousedown`, which is true for sighted users. For screen readers, however, this code is not accessible, because the `mousedown` event cannot be triggered.
- Avoid using `onmouseover` to display new options to the user. Once again, screen readers have no way to trigger this event. If you really must display new options in this manner, consider adding keyboard shortcuts to display the same information.
- Avoid using `dblclick` to execute important actions. The keyboard cannot fire this event.

Following these simple hints can greatly increase the accessibility of your web application or website to those with disabilities.



To learn more about accessibility on web pages, please visit www.webaim.org and <http://accessibility.yahoo.com>.

Keyboard and Text Events

Keyboard events are fired when the user interacts with the keyboard. DOM Level 2 Events originally specified keyboard events, but that section was removed before the specification became final. As a result, keyboard events are largely supported based on the original DOM Level 0 implementations.

DOM Level 3 Events provides a specification for keyboard events that was first completely implemented in Internet Explorer 9. Other browsers have also started work on implementing the standard, but there are still many legacy implementations.

There are three keyboard events, as described here:

- `keydown` — Fires when the user presses a key on the keyboard and fires repeatedly while the key is being held down.
- `keypress` — Fires when the user presses a key on the keyboard that results in a character and fires repeatedly while the key is being held down. This event also fires for the Esc key. DOM Level 3 Events deprecates the `keypress` event in favor of the `textInput` event.
- `keyup` — Fires when the user releases a key on the keyboard.

These events are most easily seen as the user types in a text box, though all elements support them.

There is only one text event and it is called `textInput`. This event is an augmentation of `keypress` intended to make it easier to intercept text input before being displayed to the user. The `textInput` event fires just before text is inserted into a text box.

When the user presses a character key once on the keyboard, the `keydown` event is fired first, followed by the `keypress` event, followed by the `keyup` event. Note that both `keydown` and `keypress` are fired before any change has been made to the text box, whereas the `keyup` event fires after changes have been made to the text box. If a character key is pressed and held down, `keydown` and `keypress` are fired repeatedly and don't stop until the key is released.

For noncharacter keys, a single key press on the keyboard results in the `keydown` event being fired followed by the `keyup` event. If a noncharacter key is held down, the `keydown` event fires repeatedly until the key is released, at which point the `keyup` event fires.



Keyboard events support the same set of modifier keys as mouse events. The `shiftKey`, `ctrlKey`, `altKey`, and `metaKey` properties are all available for keyboard events. Internet Explorer 8 and earlier do not support `metaKey`.

Key Codes

For `keydown` and `keyup` events, the event object's `keyCode` property is filled in with a code that maps to a specific key on the keyboard. For alphanumeric keys, the `keyCode` is the same as the ASCII value for the lowercase letter or number on that key, so the 7 key has a `keyCode` of 55 and the A key has a `keyCode` of 65, regardless of the state of the Shift key. Both the DOM and the Internet Explorer event objects support the `keyCode` property. Here's an example:



```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "keyup", function(event) {
    event = EventUtil.getEvent(event);
    alert(event.keyCode);
});
```

[KeyUpEventExample01.htm](#)

In this example, the `keyCode` is displayed every time a `keyup` event is fired. The complete list of key codes to noncharacter keys is listed in the following table.

KEY	KEY CODE	KEY	KEY CODE
Backspace	8	Numpad 8	104
Tab	9	Numpad 9	105
Enter	13	Numpad +	107

KEY	KEY CODE	KEY	KEY CODE
Shift	16	Minus (both Numpad and not)	109
Ctrl	17	Numpad .	110
Alt	18	Numpad /	111
Pause/Break	19	F1	112
Caps Lock	20	F2	113
Esc	27	F3	114
Page Up	33	F4	115
Page Down	34	F5	116
End	35	F6	117
Home	36	F7	118
Left Arrow	37	F8	119
Up Arrow	38	F9	120
Right Arrow	39	F10	121
Down Arrow	40	F11	122
Ins	45	F12	123
Del	46	Num Lock	144
Left Windows Key	91	Scroll Lock	145
Right Windows Key	92	Semicolon (IE/Safari/Chrome)	186
Context Menu Key	93	Semicolon (Opera/FF)	59
Numpad 0	96	Less-than	188
Numpad 1	97	Greater-than	190
Numpad 2	98	Forward slash	191
Numpad 3	99	Grave accent (`)	192
Numpad 4	100	Equals	61
Numpad 5	101	Left Bracket	219
Numpad 6	102	Back slash (\)	220
Numpad 7	103	Right Bracket	221
		Single Quote	222

There is one oddity regarding the keydown and keyup events. Firefox and Opera return 59 for the keyCode of the semicolon key, which is the ASCII code for a semicolon, whereas Internet Explorer, Chrome, and Safari return 186, which is the code for the keyboard key.

Character Codes

When a keypress event occurs, this means that the key affects the display of text on the screen. All browsers fire the keypress event for keys that insert or remove a character; other keys are browser-dependent. Since the DOM Level 3 Events specification has only started being implemented, there are significant implementation differences across browsers.

Internet Explorer 9+, Firefox, Chrome, and Safari support a property on the event object called charCode, which is filled in only for the keypress event and contains the ASCII code for the character related to the key that was pressed. In this case, the keyCode is typically equal to 0 or may also be equal to the key code for the key that was pressed. Internet Explorer 8 and earlier and Opera use keyCode to communicate the ASCII code for the character. To retrieve the character code in a cross-browser way, you must therefore first check to see if the charCode property is used and, if not, use keyCode instead, as shown in the following example:



Available for
download on
Wrox.com

```
var EventUtil = {
    //more code here

    getCharCode: function(event){
        if (typeof event.charCodeAt == "number"){
            return event.charCodeAt;
        } else {
            return event.keyCode;
        }
    },
    //more code here
};
```

[EventUtil.js](#)

This method checks to see if the charCode property is a number (it will be undefined for browsers that don't support it) and, if it is, returns the value. Otherwise, the keyCode value is returned. This method can be used as follows:

```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "keypress", function(event){
    event = EventUtil.getEvent(event);
    alert(EventUtil.getCharCode(event));
});
```

[KeyPressEventExample01.htm](#)

Once you have the character code, it's possible to convert it to the actual character using the String.fromCharCode() method.

DOM Level 3 Changes

Although all browsers implement some form of keyboard events, DOM Level 3 Events makes several changes. The `charCode` property, for instance, isn't part of the DOM Level 3 Events specification for keyboard events. Instead, the specification defines two additional properties: `key` and `char`.

The `key` property is intended as a replacement for `keyCode` and contains a string. When a character key is pressed, the value of `key` is equal to the text character (for example, "k" or "M"); when a noncharacter key is pressed, the value of `key` is the name of the key (for example, "Shift" or "Down"). The `char` property behaves the same as `key` when a character key is pressed and is set to `null` when a noncharacter key is pressed.

Internet Explorer 9 supports the `key` property but not the `char` property. Safari 5 and Chrome support a property called `keyIdentifier` that returns the same value that `key` would in the case of noncharacter keys (such as Shift). For character keys, `keyIdentifier` returns the character code as a string in the format "U+0000" to indicate the Unicode value.



Available for
download on
Wrox.com

```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "keypress", function(event){
    event = EventUtil.getEvent(event);
    var identifier = event.key || event.keyIdentifier;
    if (identifier){
        alert(identifier);
    }
});
```

[DOMLevel3KeyPropertyExample01.htm](#)

Because of the lack of cross-browser support, it's not recommended to use `key`, `keyIdentifier`, or `char`.

DOM Level 3 Events also adds a property called `location`, which is a numeric value indicating where the key was pressed. Possible values are 0 for default keyboard, 1 for left location (such as the left Alt key), 2 for the right location (such as the right Shift key), 3 for the numeric keypad, 4 for mobile (indicating a virtual keypad), or 5 for joystick (such as the Nintendo Wii controller). Internet Explorer 9 supports this property. Safari 5 and Chrome support an identical property called `keyLocation`, but because of a bug, the value is always 0 unless the key is on the numeric keypad (in which case it's 3); the value is never 1, 2, 4, or 5.

```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "keypress", function(event){
    event = EventUtil.getEvent(event);
    var loc = event.location || event.keyLocation;
    if (loc){
        alert(loc);
    }
});
```

[DOMLevel3LocationPropertyExample01.htm](#)

As with the `key` property, the `location` property isn't widely supported and so isn't recommended for cross-browser development.

The last addition to the event object is the `getModifierState()` method. This method accepts a single argument, a string equal to `Shift`, `Control`, `Alt`, `AltGraph`, or `Meta`, which indicates the modifier key to check. The method returns `true` if the given modifier is active (the key is being held down) or `false` if not:



```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "keypress", function(event){
    event = EventUtil.getEvent(event);
    if (event.getModifierState){
        alert(event.getModifierState("Shift"));
    }
});
```

DOMLevel3LocationGetModifierStateExample01.htm

You can retrieve some of this information already using the `shiftKey`, `altKey`, `ctrlKey`, and `metaKey` properties on the `event` object. Internet Explorer 9 is the only browser to support the `getModifierState()` method.

The `textInput` Event

The DOM Level 3 Events specification introduced an event called `textInput` that fires when a character is input to an editable area. Designed as a replacement for `keypress`, a `textInput` event behaves somewhat differently. One difference is that `keypress` fires on any element that can have focus but `textInput` fires only on editable areas. Another difference is that `textInput` fires only for keys that result in a new character being inserted, whereas `keypress` fires for keys that affect text in any way (including Backspace).

Since the `textInput` event is interested primarily in characters, it provides a `data` property on the event object that contains the character that was inserted (not the character code). The value of `data` is always the exact character that was inserted, so if the S key is pressed without Shift, `data` is "s", but if the same key is pressed holding Shift down, then `data` is "S".

The `textInput` event can be used as follows:

```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "textInput", function(event){
    event = EventUtil.getEvent(event);
    alert(event.data);
});
```

TextInputEventExample01.htm

In this example, the character that was inserted into the text box is displayed in an alert.

There is another property, on the `event` object, called `inputMethod` that indicates how the text was input into the control. The possible values are:

- 0 indicates the browser couldn't determine how the input was entered.
- 1 indicates a keyboard was used.

- 2 indicates the text was pasted in.
- 3 indicates the text was dropped in as part of a drag operation.
- 4 indicates the text was input using an IME.
- 5 indicates the text was input by selecting an option in a form.
- 6 indicates the text was input by handwriting (such as with a stylus).
- 7 indicates the text was input by voice command.
- 8 indicates the text was input by a combination of methods.
- 9 indicates the text was input by script.

Using this property, you can determine how text was input into a control in order to verify its validity.

The `textInput` event is supported in Internet Explorer 9+, Safari, and Chrome. Only Internet Explorer supports the `inputMethod` property.

Keyboard Events on Devices

The Nintendo Wii fires keyboard events when buttons are pressed on a Wii remote. Although you can't access all of the buttons on the Wii remote, there are several that fire keyboard events. Figure 13-8 illustrates the key codes that indicate particular buttons being pressed.

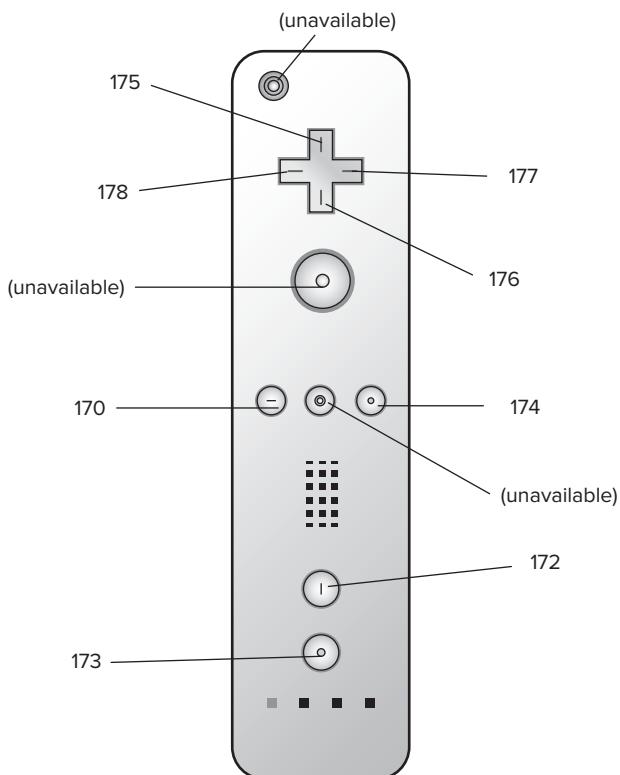


FIGURE 13-8

Keyboard events are fired when the crosspad (keycodes 175–178), minus (170), plus (174), 1 (172), or 2 (173) buttons are pressed. There is no way to tell if the power button, A, B, or Home button has been pressed.

Safari on iOS and WebKit on Android fire keyboard events when using the onscreen keyboard.

Composition Events

Composition events were first introduced in DOM Level 3 Events to handle complex input sequences typically found on IMEs. IMEs allow users to input characters not found on the physical keyboard. For example, those using a Latin keyboard can still enter Japanese characters into the computer. IMEs often require multiple keys to be pressed at once while resulting in only a single character being entered. Composition events help to detect and work with such input. There are three composition events:

- `compositionstart` — Fires when the text composition system of the IME is opened, indicating that input is about to commence.
- `compositionupdate` — Fires when a new character has been inserted into the input field.
- `compositionend` — Fires when the text composition system is closed, indicating a return to normal keyboard input.

Composition events are similar to text events in many ways. When a composition event fires, the target is the input field receiving the text. The only additional event property is `data`, which contains one of the following:

- When accessed during `compositionstart`, contains the text being edited (for instance, if text has been selected and will now be replaced).
- When accessed during `compositionupdate`, contains the new character being inserted.
- When accessed during `compositionend`, contains all of the input entered during this composition session.

As with text events, composition events can be used to filter input where necessary. These events can be used as follows:



```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "compositionstart", function(event){
    event = EventUtil.getEvent(event);
    alert(event.data);
});

EventUtil.addHandler(textbox, "compositionupdate", function(event){
    event = EventUtil.getEvent(event);
    alert(event.data);
});

EventUtil.addHandler(textbox, "compositionend", function(event){
    event = EventUtil.getEvent(event);
    alert(event.data);
});
```

[CompositionEventsExample01.htm](#)

Internet Explorer 9+ is the only browser to support composition events as of 2011. Because of this lack of support, composition events are of little use to web developers needing cross-browser support. You can determine if a browser supports composition events by using the following:

```
var isSupported = document.implementation.hasFeature("CompositionEvent", "3.0");
```

Mutation Events

The DOM Level 2 *mutation events* provide notification when a part of the DOM has been changed. Mutation events are designed to work with any XML or HTML DOM and are not specific to a particular language. The mutation events defined in DOM Level 2 are as follows:

- `DOMSubtreeModified` — Fires when any change occurs to the DOM structure. This is a catchall event that fires after any of the other events fire.
- `DOMNodeInserted` — Fires after a node is inserted as a child of another node.
- `DOMNodeRemoved` — Fires before a node is removed from its parent node.
- `DOMNodeInsertedIntoDocument` — Fires after a node has been inserted either directly or by inserting the subtree in which it exists. This event fires after `DOMNodeInserted`. This event has been deprecated in DOM Level 3 Events and should not be used.
- `DOMNodeRemovedFromDocument` — Fires before a node is removed either directly or by having the subtree in which it exists removed. This event fires after `DOMNodeRemoved`. This event has been deprecated in DOM Level 3 Events and should not be used.
- `DOMAttrModified` — Fires when an attribute has been modified. This event has been deprecated in DOM Level 3 Events and should not be used.
- `DOMCharacterDataModified` — Fires when a change is made to the value of a text node. This event has been deprecated in DOM Level 3 Events and should not be used.

You can determine if the browser supports DOM Level 2 mutation events by using the following code:

```
var isSupported = document.implementation.hasFeature("MutationEvents", "2.0");
```

Internet Explorer 8 and earlier don't support any mutation events. The following table describes browser support for the various mutation events that have not been deprecated.

EVENT	OPERA 9+	FIREFOX 3+	SAFARI 3+ AND CHROME	INTERNET EXPLORER 9+
<code>DOMSubtreeModified</code>	—	Yes	Yes	Yes
<code>DOMNodeInserted</code>	Yes	Yes	Yes	Yes
<code>DOMNodeRemoved</code>	Yes	Yes	Yes	Yes

Since many of the mutation events were deprecated in DOM Level 3 Events, this section focuses on only those events that will have continued support moving forward.

Node Removal

When a node is removed from the DOM using `removeChild()` or `replaceChild()`, the `DOMNodeRemoved` event is fired first. The target of this event is the removed node, and the event `.relatedNode` property contains a reference to the parent node. At the point that this event fires, the node has not yet been removed from its parent, so its `parentNode` property still points to the parent (same as `event.relatedNode`). This event bubbles, so the event can be handled at any level of the DOM.

If the removed node has any child nodes, the deprecated `DOMNodeRemovedFromDocument` event fires on each of those child nodes and then on the removed node. This event doesn't bubble, so an event handler is called only if it's attached directly to one of the child nodes. The target of this event is the child node or the node that was removed, and the event object provides no additional information.

After that, the `DOMSubtreeModified` event fires. The target of this event is the parent of the node that was removed. The event object provides no additional information about this event.

To understand how this works in practice, consider the following simple HTML page:

```
<!DOCTYPE html>
<html>
<head>
    <title>Node Removal Events Example</title>
</head>
<body>
    <ul id="myList">
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
</body>
</html>
```

In this example, consider removing the `` element. When that happens, the following sequence of events fire:

1. `DOMNodeRemoved` is fired on the `` element. The `relatedNode` property is `document.body`.
2. `DOMNodeRemovedFromDocument` is fired on ``.
3. `DOMNodeRemovedFromDocument` is fired on each `` element and each text node that is a child of the `` element.
4. `DOMSubtreeModified` is fired on `document.body`, since `` was an immediate child of `document.body`.

You can test this by running the following JavaScript code in the page:

```
EventUtil.addHandler(window, "load", function(event) {
    var list = document.getElementById("myList");

    EventUtil.addHandler(document, "DOMSubtreeModified", function(event) {
```

```

        alert(event.type);
        alert(event.target);
    });
    EventUtil.addHandler(document, "DOMNodeRemoved", function(event) {
        alert(event.type);
        alert(event.target);
        alert(event.relatedNode);
    });
    EventUtil.addHandler(list.firstChild, "DOMNodeRemovedFromDocument",
        function(event){
            alert(event.type);
            alert(event.target);
        });
    list.parentNode.removeChild(list);
});

```

This code adds event handlers for `DOMSubtreeModified` and `DOMNodeRemoved` to the document so they can handle all such events on the page. Since `DOMNodeRemovedFromDocument` does not bubble, its event handler is added directly to the first child of the `` element (which is a text node in DOM-compliant browsers). Once the event handlers are set up, the `` element is removed from the document.

Node Insertion

When a node is inserted into the DOM using `appendChild()`, `replaceChild()`, or `insertBefore()`, the `DOMNodeInserted` event is fired first. The target of this event is the inserted node, and the `event.relatedNode` property contains a reference to the parent node. At the point that this event fires, the node has already been added to the new parent. This event bubbles, so the event can be handled at any level of the DOM.

Next, the deprecated `DOMNodeInsertedIntoDocument` event fires on the newly inserted node. This event doesn't bubble, so the event handler must be attached to the node before it is inserted. The target of this event is the inserted node, and the `event` object provides no additional information.

The last event to fire is `DOMSubtreeModified`, which fires on the parent node of the newly inserted node.

Considering the same HTML document used in the previous section, the following JavaScript code indicates the order of events:

```

EventUtil.addHandler(window, "load", function(event) {
    var list = document.getElementById("myList");
    var item = document.createElement("li");
    item.appendChild(document.createTextNode("Item 4"));

    EventUtil.addHandler(document, "DOMSubtreeModified", function(event) {
        alert(event.type);
        alert(event.target);
    });
    EventUtil.addHandler(document, "DOMNodeInserted", function(event) {
        alert(event.type);
    });
}

```

```

        alert(event.target);
        alert(event.relatedNode);
    });
EventUtil.addHandler(item, "DOMNodeInsertedIntoDocument", function(event) {
    alert(event.type);
    alert(event.target);
});

list.appendChild(item);
});

```

This code begins by creating a new `` element containing the text "Item 4". The event handlers for `DOMSubtreeModified` and `DOMNodeInserted` are added to the document since those events bubble. Before the item is added to its parent, an event handler for `DOMNodeInsertedIntoDocument` is added to it. The last step is to use `appendChild()` to add the item, at which point the events begin to fire. The `DOMNodeInserted` event fires on the new `` item, and the `relatedNode` is the `` element. Then `DOMNodeInsertedIntoDocument` is fired on the new `` item, and lastly the `DOMSubtreeModified` event is fired on the `` element.

HTML5 Events

The DOM specification doesn't cover all events that are supported by all browsers. Many browsers have implemented custom events for various purposes based on either user need or a specific use case. HTML5 has an exhaustive list of all events that should be supported by browsers. This section discusses several events in HTML5 that are well supported by browsers. Note that this is not an exhaustive list of all events the browser supports. (Other events will be discussed throughout this book.)

The contextmenu Event

Windows 95 introduced the concept of context menus to PC users via a right mouse click. Soon, that paradigm was being mimicked on the Web. The problem developers were facing was how to detect that a context menu should be displayed (in Windows, it's a right click; on a Mac, it's a `Ctrl+click`) and then how to avoid the default context menu for the action. This resulted in the introduction of the `contextmenu` event to specifically indicate when a context menu is about to be displayed, allowing developers to cancel the default context menu and provide their own.

The `contextmenu` event bubbles, so a single event handler can be assigned to a document that handles all such events for the page. The target of the event is the element that was acted on. This event can be canceled in all browsers, using `event.preventDefault()` in DOM-compliant browsers and setting `event.returnValue` to `false` in Internet Explorer 8 and earlier. The `contextmenu` event is considered a mouse event and so has all of the properties related to the cursor position. Typically, a custom context menu is displayed using an `oncontextmenu` event handler and hidden again using the `onclick` event handler. Consider the following HTML page:



```

<!DOCTYPE html>
<html>
<head>
    <title>ContextMenu Event Example</title>
</head>
<body>

```

```

<div id="myDiv">Right click or Ctrl+click me to get a custom context menu.  

    Click anywhere else to get the default context menu.</div>  

<ul id="myMenu" style="position: absolute; visibility: hidden; background-color:  

    silver">  

    <li><a href="http://www.nczonline.net">Nicholas' site</a></li>  

    <li><a href="http://www.wrox.com">Wrox site</a></li>  

    <li><a href="http://www.yahoo.com">Yahoo!</a></li>  

</ul>  

</body>  

</html>

```

ContextMenuEventExample01.htm

In this code, a `<div>` is created that has a custom context menu. The `` element serves as the custom context menu and is initially hidden. The JavaScript to make this example work is as follows:



Available for
download on
Wrox.com

```

EventUtil.addHandler(window, "load", function(event) {
    var div = document.getElementById("myDiv");

    EventUtil.addHandler(div, "contextmenu", function(event) {
        event = EventUtil.getEvent(event);
        EventUtil.preventDefault(event);

        var menu = document.getElementById("myMenu");
        menu.style.left = event.clientX + "px";
        menu.style.top = event.clientY + "px";
        menu.style.visibility = "visible";
    });

    EventUtil.addHandler(document, "click", function(event) {
        document.getElementById("myMenu").style.visibility = "hidden";
    });
});

```

ContextMenuEventExample01.htm

Here, an `oncontextmenu` event handler is defined for the `<div>`. The event handler begins by canceling the default behavior, ensuring that the browser's context menu won't be displayed. Next, the `` element is placed into position based on the `clientX` and `clientY` properties of the event object. The last step is to show the menu by setting its `visibility` to `"visible"`. An `onclick` event handler is then added to the document to hide the menu whenever a click occurs (which is the behavior of system context menus).

Though this example is very basic, it is the basis for all custom context menus on the Web. Applying some additional CSS to the context menu in this example can yield great results.

The `contextmenu` event is supported in Internet Explorer, Firefox, Safari, Chrome, and Opera 11+.

The `beforeunload` Event

The `beforeunload` event fires on the `window` and is intended to give developers a way to prevent the page from being unloaded. This event fires before the page starts to unload from the browser,

allowing continued use of the page should it ultimately not be unloaded. You cannot cancel this event outright because that would be the equivalent of holding the user hostage on a page. Instead, the event gives you the ability to display a message to the user. The message indicates that the page is about to be unloaded, displays the message, and asks if the user would like to continue to close the page or stay (see Figure 13-9).

In order to cause this dialog box to pop up, you must set `event.returnValue` equal to the string you want displayed in the dialog (for Internet Explorer and Firefox) and also return it as the function value (for Safari and Chrome), as in this example:

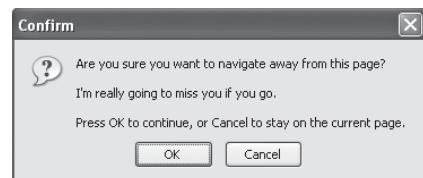


FIGURE 13-9



```
EventUtil.addHandler(window, "beforeunload", function(event){
    event = EventUtil.getEvent(event);
    var message = "I'm really going to miss you if you go.";
    event.returnValue = message;
    return message;
});
```

[BeforeUnloadEventExample01.htm](#)

Internet Explorer, Firefox, Safari, and Chrome support the `beforeunload` event and pop up the dialog box to confirm that the user wants to navigate away. Opera as of version 11 does not support `beforeunload`.

The DOMContentLoaded Event

The `window`'s `load` event fires when everything on the page has been completely loaded, which may take some time for pages with lots of external resources. The `DOMContentLoaded` event fires as soon as the DOM tree is completely formed and without regard to images, JavaScript files, CSS files, or other such resources. As compared to the `load` event, `DOMContentLoaded` allows event handlers to be attached earlier in the page download process, which means a faster time to interactivity for users.

To handle the `DOMContentLoaded` event, you can attach an event handler either on the `document` or on the `window` (the target for the event actually is `document`, although it bubbles up to `window`). Here's an example:

```
EventUtil.addHandler(document, "DOMContentLoaded", function(event){
    alert("Content loaded");
});
```

[DOMContentLoadedEventExample01.htm](#)

The event object for `DOMContentLoaded` doesn't provide any additional information (`target` is `document`).

The `DOMContentLoaded` event is supported in Internet Explorer 9+, Firefox, Chrome, Safari 3.1+, and Opera 9+ and is typically used to attach event handlers or perform other DOM manipulations. This event always fires before the `load` event.

For browsers that don't support `DOMContentLoaded`, it has been suggested that a timeout should be set during page loading with a millisecond delay of 0, as in this example:

```
setTimeout(function(){
    //attach event handlers here
}, 0);
```

This code essentially says, “Run this function as soon as the current JavaScript process is complete.” There is a single JavaScript process running as the page is being downloaded and constructed, so the timeout will fire after that. Whether or not this coincides directly with the timing of `DOMContentLoaded` relates to both the browser being used and other code on the page. To work properly, this must be the first timeout set on the page, and even then, it is not guaranteed that the timeout will run prior to the `load` event in all circumstances.

The `readystatechange` Event

Internet Explorer first defined an event called `readystatechange` on several parts of a DOM document. This somewhat mysterious event is intended to provide information about the loading state of the document or of an element, though its behavior is often erratic. Each object that supports the `readystatechange` event has a `readyState` property that can have one of the following five possible string values:

- `uninitialized` — The object exists but has not been initialized.
- `loading` — The object is loading data.
- `loaded` — The object has finished loading its data.
- `interactive` — The object can be interacted with but it's not fully loaded.
- `complete` — The object is completely loaded.

Even though this seems straightforward, not all objects go through all `readyState` phases. The documentation indicates that objects may completely skip a phase if it doesn't apply but doesn't indicate which phases apply to which objects. This means that the `readystatechange` event often fires fewer than four times and the `readyState` value doesn't always follow the same progression.

When used on `document`, a `readyState` of "interactive" fires the `readystatechange` event at a time similar to `DOMContentLoaded`. The interactive phase occurs when the entire DOM tree has been loaded and thus is safe to interact with. Images and other external resources may or may not be available at that point in time. The `readystatechange` event can be handled like this:

```
EventUtil.addHandler(document, "readystatechange", function(event) {
    if (document.readyState == "interactive") {
        alert("Content loaded");
    }
});
```

The `event` object for this event doesn't provide any additional information and has no target set.

When used in conjunction with the `load` event, the order in which these events fire is not guaranteed. In pages with numerous or large external resources, the interactive phase is reached

well before the `load` event fires; in smaller pages with few or small external resources, the `readystatechange` event may not fire until after the `load` event.

To make matters even more confusing, the interactive phase may come either before or after the complete phase; the order is not constant. In pages with more external resources, it is more likely that the interactive phase will occur before the complete phase, whereas in pages with fewer resources, it is more likely that the complete phase will occur before the interactive phase. So, to ensure that you are getting the earliest possible moment, it's necessary to check for both the interactive and the complete phases, as in this example:

```
EventUtil.addHandler(document, "readystatechange", function(event){
    if (document.readyState == "interactive" || document.readyState == "complete"){
        EventUtil.removeHandler(document, "readystatechange", arguments.callee);
        alert("Content loaded");
    }
});
```

When the `readystatechange` event fires in this code, the `document.readyState` property is checked to see if it's either the interactive or the complete phase. If so, the event handler is removed to ensure that it won't be executed for another phase. Note that because the event handler is an anonymous function, `arguments.callee` is used as the pointer to the function. After that, the alert is displayed indicating that the content is loaded. This construct allows you to get as close as possible to the `DOMContentLoaded` event.

The `readystatechange` event on the document is supported in Internet Explorer, Firefox 4+, and Opera.



Even though you can get close to mimicking `DOMContentLoaded` using `readystatechange`, they are not exactly the same. The order in which the `load` event and `readystatechange` events are fired is not consistent from page to page.

The `readystatechange` event also fires on `<script>` (Internet Explorer and Opera) and `<link>` (Internet Explorer only), allowing you to determine when external JavaScript and CSS files have been loaded. As with other browsers, dynamically created elements don't begin downloading external resources until they are added to the page. The behavior of this event for elements is similarly confusing, because the `readyState` property may be either "loaded" or "complete" to indicate that the resource is available. Sometimes the `readyState` stops at "loaded" and never makes it to "complete", and other times it skips "loaded" and goes straight to "complete". As a result, it's necessary to use the same construct used with the document. For example, the following loads an external JavaScript file:



```
EventUtil.addHandler(window, "load", function(){
    var script = document.createElement("script");

    EventUtil.addHandler(script, "readystatechange", function(event){
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        if (target.readyState == "loaded" || target.readyState == "complete") {
```

```

        EventUtil.removeHandler(target, "readystatechange", arguments.
callee);
        alert("Script Loaded");
    }
});
script.src = "example.js";
document.body.appendChild(script);
});

```

[ReadyStateChangeEventExample01.htm](#)

This example assigns an event handler to a newly created `<script>` node. The target of the event is the node itself, so when the `readystatechange` event fires, the target's `readyState` property is checked to see if it's either `"loaded"` or `"complete"`. If the phase is either of the two, then the event handler is removed (to prevent it from possibly being executed twice) and then an alert is displayed. At this time, you can start executing functions that have been loaded from the external file.

The same construct can be used to load CSS files via a `<link>` element, as shown in this example:



Available for
download on
[Wrox.com](#)

```

EventUtil.addHandler(window, "load", function(){
    var link = document.createElement("link");
    link.type = "text/css";
    link.rel= "stylesheet";

    EventUtil.addHandler(script, "readystatechange", function(event){
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        if (target.readyState == "loaded" || target.readyState == "complete"){
            EventUtil.removeHandler(target, "readystatechange", arguments.
callee);
            alert("CSS Loaded");
        }
    });

    link.href = "example.css";
    document.getElementsByTagName("head")[0].appendChild(link);
});

```

[ReadyStateChangeEventExample02.htm](#)

Once again, it's important to test for both `readyState` values and to remove the event handler after calling it once.

The `pageshow` and `pagehide` Events

Firefox and Opera introduced a feature called the *back-forward cache (bfcache)* designed to speed up page transitions when using the browser's Back and Forward buttons. The cache stores not only page data but also the DOM and JavaScript state, effectively keeping the entire page in memory. If a page is in the bfcache, the `load` event will not fire when the page is navigated to. This usually doesn't cause an issue since the entire page state is stored. However, Firefox decided to provide some events to give visibility to the bfcache behavior.

The first event is pageshow, which fires whenever a page is displayed, whether from the bfcache or not. On a newly loaded page, pageshow fires after the load event; on a page in the bfcache, pageshow fires as soon as the page's state has been completely restored. Note that even though the target of this event is document, the event handler must be attached to window. Consider the following:

```
(function(){
    var showCount = 0;

    EventUtil.addHandler(window, "load", function(){
        alert("Load fired");
    });

    EventUtil.addHandler(window, "pageshow", function(){
        showCount++;
        alert("Show has been fired " + showCount + " times.");
    });
})();
```

This example uses a private scope to protect the showCount variable from being introduced into the global scope. When the page is first loaded, showCount has a value of 0. Every time the pageshow event fires, showCount is incremented and an alert is displayed. If you navigate away from the page containing this code and then click the Back button to restore it, you will see that the value of showCount is incremented each time. That's because the variable state, along with the entire page state, is stored in memory and then retrieved when you navigate back to the page. If you were to click the Reload button on the browser, the value of showCount would be reset to 0 because the page would be completely reloaded.

Besides the usual properties, the event object for pageshow includes a property called persisted. This is a Boolean value that is set to true if the page is stored in the bfcache or false if the page is not. The property can be checked in the event handler as follows:



```
(function(){
    var showCount = 0;

    EventUtil.addHandler(window, "load", function(){
        alert("Load fired");
    });

    EventUtil.addHandler(window, "pageshow", function(){
        showCount++;
        alert("Show has been fired " + showCount +
            " times. Persisted? " + event.persisted);
    });
})();
```

PageShowEventExample01.htm

The persisted property lets you determine if a different action must be taken depending on the state of the page in the bfcache.

The pagehide event is a companion to pageshow and fires whenever a page is unloaded from the browser, firing immediately before the unload event. As with the pageshow event, pagehide fires

on the document even though the event handler must be attached to the window. The event object also includes the `persisted` property, though there is a slight difference in its usage. Consider the following:



Available for download on
Wrox.com

```
EventUtil.addHandler(window, "pagehide", function(event) {
    alert("Hiding. Persisted? " + event.persisted);
});
```

PageShowEventExample01.htm

You may decide to take a different action based on the value of `persisted` when `pagehide` fires. For the `pageshow` event, `persisted` is set to `true` if the page has been loaded from the bfcache; for the `pagehide` event, `persisted` is set to `true` if the page will be stored in the bfcache once unloaded. So the first time `pageshow` is fired, `persisted` is always `false`, whereas the first time `pagehide` is fired, `persisted` will be `true` (unless the page won't be stored in the bfcache).

The `pageshow` and `pagehide` events are supported in Firefox, Safari 5+, Chrome, and Opera. Internet Explorer through version 9 does not support these events.



Pages that have an `onunload` event handler assigned are automatically excluded from the bfcache, even if the event handler is empty. The reasoning is that `onunload` is typically used to undo what was done using `onload`, and so skipping `onload` the next time the page is displayed could cause it to break.

The hashchange Event

HTML5 introduced the `hashchange` event as a way to notify developers when the URL hash (everything following a pound sign (#) in a URL) changed. This came about as developers frequently used the URL hash to store state information or navigational information in Ajax applications.

The `onhashchange` event handler must be attached to the `window`, and it is called whenever the URL hash changes. The event object should have two additional properties: `oldURL` and `newURL`. These properties hold the complete URL including the hash before the change and after the change. For example:

```
EventUtil.addHandler(window, "hashchange", function(event) {
    alert("Old URL: " + event.oldURL + "\nNew URL: " + event.newURL);
});
```

HashChangeEventExample01.htm

The `hashchange` event is supported in Internet Explorer 8+, Firefox 3.6+, Safari 5+, Chrome, and Opera 10.6+. Of those browsers, only Firefox 6+, Chrome, and Opera support the `oldURL` and `newURL` properties. For that reason, it's best to use the `location` object to determine the current hash:

```
EventUtil.addHandler(window, "hashchange", function(event) {
    alert("Current hash: " + location.hash);
});
```

You can detect if the hashchange event is supported using the following code:

```
var isSupported = ("onhashchange" in window); //buggy
```

Internet Explorer 8 has a quirk where this code returns `true` even when running in Internet Explorer 7 document mode, even though it doesn't work. To get around this, use the following as a more bulletproof detection:

```
var isSupported = ("onhashchange" in window) && (document.documentElement === undefined || document.documentElement > 7);
```

Device Events

With the introduction of smartphones and tablet devices came a new set of ways for users to interact with a browser. As such, a new class of events was invented. *Device events* allow you to determine how a device is being used. A new draft for device events was started in 2011 at the W3C (<http://dev.w3.org/geo/api/spec-source-orientation.html>) to cover the growing number of devices looking to implement device-related events. This section covers both the API defined in the draft and vendor-specific events.

The orientationchange Event

Apple created the `orientationchange` event on mobile Safari so that developers could determine when the user switched the device from landscape to portrait mode. There is a `window.orientation` property on mobile Safari that contains one of three values: `0` for portrait mode, `90` for landscape mode when rotated to the left (the Home button on the right), and `-90` for landscape mode when rotated to the right (the Home button on the left). The documentation also mentions a value of `180` if the device is upside down, but that configuration is not supported to date. Figure 13-10 illustrates the various values for `window.orientation`.

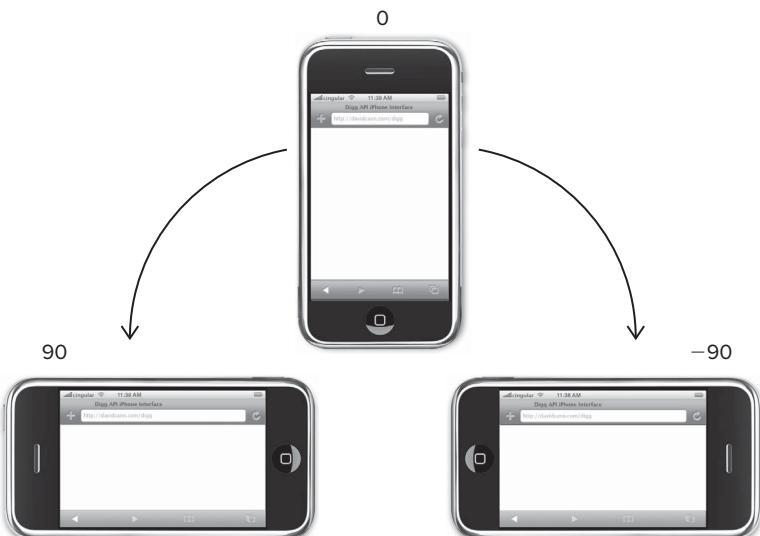


FIGURE 13-10

Whenever the user changes from one mode to another, the `orientationchange` event fires. The event object doesn't contain any useful information, since the only relevant information is accessible via `window.orientation`. Typical usage of this event is as follows:



```
EventUtil.addHandler(window, "load", function(event) {
    var div = document.getElementById("myDiv");
    div.innerHTML = "Current orientation is " + window.orientation;

    EventUtil.addHandler(window, "orientationchange", function(event) {
        div.innerHTML = "Current orientation is " + window.orientation;
    });
});
```

[OrientationChangeEventExample01.htm](#)

In this example, the initial orientation is displayed when the `load` event fires. Then, the event handler for `orientationchange` is assigned. Whenever the event fires, the message on the page is updated to indicate the new orientation.

All iOS devices support both the `orientationchange` event and the `window.orientation` property.



Since `orientationchange` is considered a window event, you can also assign an event handler by adding the `onorientationchange` attribute to the `<body>` element.

The MozOrientation Event

Firefox 3.6 introduced a new event called `MozOrientation` to detect device orientation. (The prefix `Moz` indicates that it's a vendor-specific event instead of a standard event.) This event fires periodically as the device accelerometer detects changes in how the device is oriented. Note that this is different from `orientationchange` in iOS, which provides only one dimension of movement. The `MozOrientation` event fires on the `window` object and so can be handled with the following code:

```
EventUtil.addHandler(window, "MozOrientation", function(event) {
    //respond to event
});
```

The event object has three properties with accelerometer data: `x`, `y`, and `z`. Each value is a number between 1 and -1 and represents a different axis. When at rest, `x` is 0, `y` is 0, and `z` is 1 (indicating the device is upright). Tilting to the right decreases the value of `x`, while tilting to the left increases the value. Likewise, tilting away from you decreases the value of `y`, while tilting toward you (as if to read a paper) increases it. The `z` value is the vertical acceleration, and so is 1 at rest, and decreases when the device is in motion. (It would be 0 with no gravity.) Here's a simple example that outputs the three values:

```
EventUtil.addHandler(window, "MozOrientation", function(event) {
    var output = document.getElementById("output");
    output.innerHTML = "X=" + event.x + ", Y=" + event.y + ", Z=" + event.z +
        "<br>";
});
```

[MozOrientationEventExample01.htm](#)

The `MozOrientation` event is supported only on devices with accelerometers, including Macbook laptops, Lenovo Thinkpad laptops, and both Windows Mobile and Android devices. It should be noted that this is an experimental API and may or may not change in the future. (It is likely that it may be superseded by another event.)

The `deviceorientation` Event

The `deviceorientation` event is defined in the `DeviceOrientation` Event specification and is similar in nature to the `MozOrientation` event. The event is fired on `window` when accelerometer information is available and changes and, as such, has the same support limitations as `MozOrientation`. Keep in mind that the purpose of `deviceorientation` is to inform you of how the device is oriented in space and not of movement.

A device is said to exist in three-dimensional space along an `x`-axis, a `y`-axis, and a `z`-axis. These all start at zero when the device is at rest on a horizontal surface. The `x`-axis goes from the left of the device to the right, the `y`-axis goes from the bottom of the device to the top, and the `z`-axis goes from the back of the device to the front (see Figure 13-11).

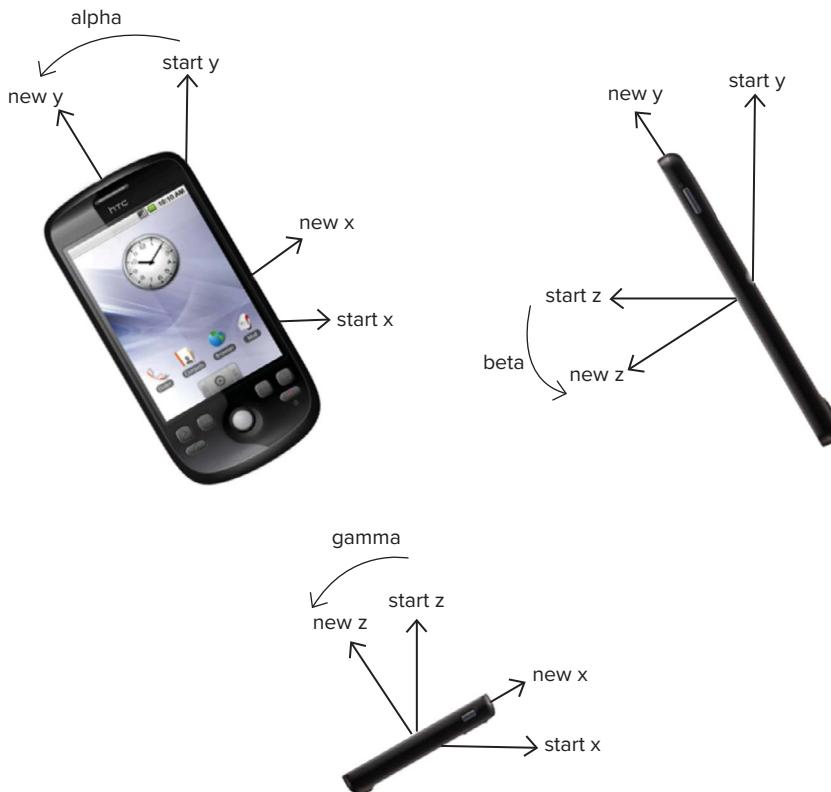
When `deviceorientation` fires, it returns information about how the values of each axis have changed relative to the device at rest. The event object has five properties:

- `alpha` — The difference in `y`-axis degrees as you rotate around the `z`-axis (side-to-side tilt); a floating point number between 0 and 360.
- `beta` — The difference in `z`-axis degrees as you rotate around the `x`-axis (front-to-back tilt); a floating point number between -180 and 180.
- `gamma` — The difference in the `z`-axis degrees as you rotate around the `y`-axis (twisting tilt); a floating point number between -90 and 90.
- `absolute` — A Boolean value indicating if the device is returning absolute values or not.
- `compassCalibrated` — A Boolean value indicating if the device's compass is properly calibrated or not.

Figure 13-12 shows how the values of `alpha`, `beta`, and `gamma` are calculated.



FIGURE 13-11

**FIGURE 13-12**

Here's a simple example that outputs the values for alpha, beta, and gamma:



Available for download on
Wrox.com

```
EventUtil.addHandler(window, "deviceorientation", function(event){
    var output = document.getElementById("output");
    output.innerHTML = "Alpha=" + event.alpha + ", Beta=" + event.beta +
                      ", Gamma=" + event.gamma + "<br>";
});
```

[DeviceOrientationEventExample01.htm](#)

You can use this information to rearrange or otherwise alter elements on the screen in reaction to the device changing its orientation. For example, this code rotates an element in reaction to the device orientation:

```
EventUtil.addHandler(window, "deviceorientation", function(event){
    var arrow = document.getElementById("arrow");
    arrow.style.webkitTransform = "rotate(" + Math.round(event.alpha) + "deg)";
});
```

[DeviceOrientationEventExample01.htm](#)

This example works only on mobile WebKit browsers because of the use of the proprietary `webkitTransform` property (the temporary version of the CSS standard `transform` property). The element “arrow” is rotated along with the value of `event.alpha`, giving it a compass-like feel. The CSS3 rotation transformation is used with a rounded version of the value to ensure smoothness.

As of 2011, Safari for iOS 4.2+, Chrome, and WebKit for Android are the only implementations of the `deviceorientation` event.

The `devicemotion` Event

The DeviceOrientation Event specification also includes a `devicemotion` event. This event is designed to inform you when the device is actually moving, not just when it has changed orientation. For instance, `devicemotion` is useful to determine that the device is falling or is being held by someone who is walking.

When the `devicemotion` event fires, the `event` object contains the following additional properties:

- `acceleration` — An object containing `x`, `y`, and `z` properties that tells you the acceleration in each dimension without considering gravity.
- `accelerationIncludingGravity` — An object containing `x`, `y`, and `z` properties that tells you the acceleration in each dimension, including the natural acceleration of gravity in the `z`-axis.
- `interval` — The amount of time, in milliseconds, that will pass before another `devicemotion` event is fired. This value should be constant from event to event.
- `rotationRate` — An object containing the `alpha`, `beta`, and `gamma` properties that indicate device orientation.

If `acceleration`, `accelerationIncludingGravity`, or `rotationRate` cannot be provided, then the property value is `null`. Because of that, you should always check that the value is not `null` before using any of these properties. For example:



```
EventUtil.addHandler(window, "devicemotion", function(event){
    var output = document.getElementById("output");
    if (event.rotationRate !== null){
        output.innerHTML += "Alpha=" + event.rotationRate.alpha + ", Beta=" +
                           event.rotationRate.beta + ", Gamma=" +
                           event.rotationRate.gamma;
    }
});
```

[DeviceMotionEventExample01.htm](#)

As with `deviceorientation`, Safari for iOS 4.2+, Chrome, and WebKit for Android are the only implementations of the `devicemotion` event.

Touch and Gesture Events

Safari for iOS introduced several proprietary events designed to inform developers when specific events occur. Since iOS devices are mouseless and keyboardless, the regular mouse and keyboard

events simply aren't enough to create a completely interactive web page designed with mobile Safari in mind. With the introduction of WebKit for Android, many of the proprietary events became de facto standards and led to the beginning of a Touch Events specification from the W3C (found at <https://dvcs.w3.org/hg/webevents/raw-file/tip/touchevents.html>). The following events work only on touch-based devices.

Touch Events

When the iPhone 3G was released with iOS 2.0 software, a new version of Safari was included. This new mobile Safari exposed several new events relating to touch interactions. The Android browser later implemented these same events. *Touch events* are fired when a finger is placed on the screen, dragged across the screen, or removed from the screen. The touch events are as follows:

- `touchstart` — Fires when a finger touches the screen even if another finger is already touching the screen.
- `touchmove` — Fires continuously as a finger is moved across the screen. Calling `preventDefault()` during this event prevents scrolling.
- `touchend` — Fires when a finger is removed from the screen.
- `touchcancel` — Fires when the system has stopped tracking the touch. It's unclear in the documentation as to when this can occur.

Each of these events bubbles and can be canceled. Even though touch events aren't part of the DOM specification, they are implemented in a DOM-compatible way. So the `event` object for each touch event provides properties that are common to mouse events: `bubbles`, `cancelable`, `view`, `clientX`, `clientY`, `screenX`, `screenY`, `detail`, `altKey`, `shiftKey`, `ctrlKey`, and `metaKey`.

In addition to these common DOM properties, touch events have the following three properties to track touches:

- `touches` — An array of `Touch` objects that indicate the currently tracked touches.
- `targetTouches` — An array of `Touch` objects specific to the event's target.
- `changedTouches` — An array of `Touch` objects that have been changed in the last user action.

Each `Touch` object, in turn, has the following properties:

- `clientX` — The x-coordinate of the touch target in the viewport.
- `clientY` — The y-coordinate of the touch target in the viewport.
- `identifier` — A unique ID for the touch.
- `pageX` — The x-coordinate of the touch target on the page.
- `pageY` — The y-coordinate of the touch target on the page.
- `screenX` — The x-coordinate of the touch target on the screen.
- `screenY` — The y-coordinate of the touch target on the screen.
- `target` — The DOM node target for the touch.



Available for
download on
Wrox.com

```
function handleTouchEvent(event) {
    //only for one touch
    if (event.touches.length == 1)

        var output = document.getElementById("output");
        switch(event.type){
            case "touchstart":
                output.innerHTML = "Touch started (" + event.touches[0].clientX +
                    ", " + event.touches[0].clientY + ")";
                break;
            case "touchend":
                output.innerHTML += "<br>Touch ended (" +
                    event.changedTouches[0].clientX + "," +
                    event.changedTouches[0].clientY + ")";
                break;
            case "touchmove":
                event.preventDefault(); //prevent scrolling
                output.innerHTML += "<br>Touch moved (" +
                    event.changedTouches[0].clientX + "," +
                    event.changedTouches[0].clientY + ")";
                break;
        }
    }

    EventUtil.addHandler(document, "touchstart", handleTouchEvent);
    EventUtil.addHandler(document, "touchend", handleTouchEvent);
    EventUtil.addHandler(document, "touchmove", handleTouchEvent);
}
```

TouchEventsExample01.htm

This code tracks a single touch around the screen. To keep things simple, it outputs information only when there's a single active touch. When the touchstart event occurs, it outputs the location of the touch into a `<div>`. When a touchmove event fires, its default behavior is canceled to prevent scrolling (moving touches typically scroll the page) and then it outputs information about the changed touch. The touchend event outputs the last information about the touch. Note that there is nothing in the touches collection during the touchend event, because there is no longer an active touch; the changedTouches collection must be used instead.

These events fire on all elements of the document, so you can manipulate different parts of the page individually. The order of events (including mouse events) when you tap on an element are:

1. touchstart
2. mouseover
3. mousemove (once)
4. mousedown

- 5.** mouseup
- 6.** click
- 7.** touchend

Safari for iOS, WebKit for Android, Dolfin for bada, BlackBerry WebKit for OS6+, Opera Mobile 10.1+, and Phantom browser for LG-proprietary OSs all support touch events. Only Safari on iOS can support multiple touches at once. Both Firefox 6+ and Chrome on the desktop also support touch events.

Gesture Events

The iOS 2.0 version of Safari also introduced a class of events for gestures. A *gesture* occurs when two fingers are touching the screen and typically causes a change in the scale of the displayed item or the rotation. There are three gesture events, as described here:

- `gesturestart` — Fires when a finger is already on the screen and another finger is placed on the screen.
- `gesturechange` — Fires when the position of either finger on the screen has changed.
- `gestureend` — Fires when one of the fingers has been removed from the screen.

These events fire only if the two fingers are touching the recipient of the event. Setting event handlers on a single element means that both fingers must be within the bounds of the element in order for gesture events to fire (this will be the target). Since these events bubble, you can also place event handlers at the document level to handle all gesture events. When you are using this approach, the target of the event will be the element that has both fingers within its boundaries.

There is a relationship between the touch and the gesture events. When a finger is placed on the screen, the `touchstart` event fires. When another finger is placed on the screen, the `gesturestart` event fires first and is followed by the `touchstart` event for that finger. If one or both of the fingers are moved, a `gesturechange` event is fired. As soon as one of the fingers is removed, the `gestureend` event fires, followed by `touchend` for that finger.

As with touch events, each gesture event object contains all of the standard mouse event properties: `bubbles`, `cancelable`, `view`, `clientX`, `clientY`, `screenX`, `screenY`, `detail`, `altKey`, `shiftKey`, `ctrlKey`, and `metaKey`. The two additions to the event object are `rotation` and `scale`. The `rotation` property indicates the degrees of rotation that the fingers have changed, where negative numbers indicate a counterclockwise rotation and positive numbers indicate clockwise rotation (the value begins as 0). The `scale` property indicates how much of a distance change occurred between the fingers (making a pinch motion). This starts out as 1 and will either increase as the distance increases or decrease as the distance decreases.

These events can be used as follows:



```
function handleGestureEvent(event) {
    var output = document.getElementById("output");
    switch(event.type) {
        case "gesturestart":
            output.innerHTML = "Gesture started (rotation=" + event.rotation +
                ",scale=" + event.scale + ")";
```

```

        break;
    case "gestureend":
        output.innerHTML += "<br>Gesture ended (rotation=" + event.rotation +
                           ",scale=" + event.scale + ")";
        break;
    case "gesturechange":
        output.innerHTML += "<br>Gesture changed (rotation=" + event.rotation +
                           ",scale=" + event.scale + ")";
        break;
    }
}

document.addEventListener("gesturestart", handleGestureEvent, false);
document.addEventListener("gestureend", handleGestureEvent, false);
document.addEventListener("gesturechange", handleGestureEvent, false);

```

GestureEventsExample01.htm

As with the touch events example, this code simply wires up each event to a single function and then outputs information about each event.



*Touch events also return **rotation** and **scale** properties, but they change only when two fingers are in contact with the screen. Generally, it is easier to use gesture events with two fingers than to manage all interactions with touch events.*

MEMORY AND PERFORMANCE

Since event handlers provide the interaction on modern web applications, many developers mistakenly add a large number of them to the page. In languages that create GUIs, such as C#, it's customary to add an `onclick` event handler to each button in the GUI, and there is no real penalty for doing so. In JavaScript, the number of event handlers on the page directly relates to the overall performance of the page. This happens for a number of reasons. The first is that each function is an object and takes up memory; the more objects in memory, the slower the performance. Second, the amount of DOM access needed to assign all of the event handlers up front delays the interactivity of the entire page. There are a number of ways that you can improve performance by minding your use of event handlers.

Event Delegation

The solution to the “too many event handlers” issue is called *event delegation*. Event delegation takes advantage of event bubbling to assign a single event handler to manage all events of a particular type. The `click` event, for example, bubbles all the way up to the document level. This means that it's possible to assign one `onclick` event handler for an entire page instead of one for each clickable element. Consider the following HTML:



```
<ul id="myLinks">
<li id="goSomewhere">Go somewhere</li>
<li id="doSomething">Do something</li>
```

```
<li id="sayHi">Say hi</li>
</ul>
```

EventDelegationExample01.htm

The HTML in this example contains three items that should perform actions when clicked. Traditional thinking simply attaches three event handlers like this:

```
var item1 = document.getElementById("goSomewhere");
var item2 = document.getElementById("doSomething");
var item3 = document.getElementById("sayHi");

EventUtil.addHandler(item1, "click", function(event){
    location.href = "http://www.wrox.com";
});

EventUtil.addHandler(item2, "click", function(event){
    document.title = "I changed the document's title";
});

EventUtil.addHandler(item3, "click", function(event){
    alert("hi");
});
```

If this scenario is repeated for all of the clickable elements in a complex web application, the result is an incredibly long section of code that simply attaches event handlers. Event delegation approaches this problem by attaching a single event handler to the highest possible point in the DOM tree, as in this example:



Available for
download on
Wrox.com

```
var list = document.getElementById("myLinks");

EventUtil.addHandler(list, "click", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    switch(target.id){
        case "doSomething":
            document.title = "I changed the document's title";
            break;

        case "goSomewhere":
            location.href = "http://www.wrox.com";
            break;

        case "sayHi":
            alert("hi");
            break;
    }
});
```

EventDelegationExample01.htm

In this code, event delegation is used to attach a single `onclick` event handler to the `` element. Since all of the list items are children of this element, their events bubble up and are handled by this function. The event target is the list item that was clicked, so you can check the `id` property to determine the appropriate action. In comparison with the previous code that didn't use event delegation, this code has less of an up-front cost, because it just retrieves one DOM element and attaches one event handler. The end result is the same for the user, but this approach requires much less memory. All events that use buttons (most mouse events and keyboard events) are candidates for this technique.

If it's practical, you may want to consider attaching a single event handler on document that can handle all of the page events of a particular type. This has the following advantages compared to traditional techniques:

- The `document` object is immediately available and can have event handlers assigned at any point during the page's life cycle (no need to wait for `DOMContentLoaded` or `load` events). This means that as soon as a clickable element is rendered, it can function appropriately without delay.
- Less time is spent setting up event handlers on the page. Assigning one event handler takes fewer DOM references and less time.
- Lower memory usage is required for the entire page, improving overall performance.

The best candidates for event delegation are `click`, `mousedown`, `mouseup`, `keydown`, `keyup`, and `keypress`. The `mouseover` and `mouseout` events bubble but are complicated to handle properly and often require calculating element position to appropriately handle (since `mouseout` fires when moving from an element to one of its child nodes and when moving outside of the element).

Removing Event Handlers

When event handlers are assigned to elements, a connection is formed between code that is running the browser and JavaScript code interacting with the page. The more of these connections that exist, the slower a page performs. One way to handle this issue is through event delegation to limit the number of connections that are set up. Another way to manage the issue is to remove event handlers when they are no longer needed. Dangling event handlers, those that remain in memory after they are necessary, are a major source of memory and performance issues in web applications.

This problem occurs at two specific points during a page's life cycle. The first is when an element is removed from the document while it has event handlers attached. This can be due to a true DOM manipulation involving `removeChild()` or `replaceChild()`, but it happens most often when using `innerHTML` to replace a section of the page. Any event handlers assigned to an element that was eliminated by the call to `innerHTML` may not be properly garbage collected. Consider the following example:

```
<div id="myDiv">
    <input type="button" value="Click Me" id="myBtn">
</div>
<script type="text/javascript">
    var btn = document.getElementById("myBtn");
```

```

btn.onclick = function(){

    //do something

    document.getElementById("myDiv").innerHTML = "Processing..."; //Bad!!!
};

</script>

```

Here, a button exists inside of a `<div>` element. When the button is clicked, it is removed and replaced with a message to prevent double-clicking, which is a very common paradigm on websites. The issue is that the button still had an event handler attached when it was removed from the page. Setting `innerHTML` on the `<div>` removed the button completely, but the event handler remains attached. Some browsers, especially Internet Explorer 8 and earlier, will have trouble in this situation, and most likely, references to both the element and the event handler will remain in memory. If you know that a given element is going to be removed, it's best to manually remove the event handlers yourself, as in this example:

```

<div id="myDiv">
    <input type="button" value="Click Me" id="myBtn">
</div>
<script type="text/javascript">
    var btn = document.getElementById("myBtn");
    btn.onclick = function(){

        //do something

        btn.onclick = null; //remove event handler

        document.getElementById("myDiv").innerHTML = "Processing...";
    };
</script>

```

In this rewritten code, the button's event handler is removed before setting the `<div>` element's `innerHTML`. This ensures that the memory will be reclaimed and the button can safely be removed from the DOM.

Note also that removing the button in the event handler prevents bubbling of the event. An event will bubble only if its target is still present in the document.



Event delegation also helps solve this problem. If you know that a particular part of the page is going to be replaced using `innerHTML`, do not attach event handlers directly to elements within that part. Instead, attach event handlers at a higher level that can handle events in that area.

The other time that dangling event handlers are a problem is when the page is unloaded. Once again, Internet Explorer 8 and earlier have a lot of problems with this situation, though it seems to affect all browsers in some way. If event handlers aren't cleaned up before the page is unloaded,

they remain in memory. Each time the browser loads and unloads the page after that (as a result of navigating away and back or clicking the Reload button), the number of objects in memory increases, since the event handler memory is not being reclaimed.

Generally speaking, it's a good idea to remove all event handlers before the page is unloaded by using an `onunload` event handler. This is another area where event delegation helps, because it is easier to keep track of the event handlers to remove when there are fewer of them. A good way to think about this technique is that anything done using an `onload` event handler must be reversed using `onunload`.



Keep in mind that assigning an `onunload` event handler means that your page will not be stored in the bfcache. If this is of concern, you may want to use `onunload` to remove event handlers only in Internet Explorer.

SIMULATING EVENTS

Events are designed to indicate particular moments of interest in a web page. These events are often fired based on user interaction or other browser functionality. It's a little-known fact that JavaScript can be used to fire specific events at any time, and those events are treated the same as events that are created by the browser. This means that the events bubble appropriately and cause the browser to execute event handlers assigned to deal with the event. This capability can be extremely useful in testing web applications. The DOM Level 3 specification indicates ways to simulate specific types of events, and Internet Explorer 9, Opera, Firefox, Chrome, and Safari all support it. Internet Explorer 8 and earlier versions have their own way to simulate events.

DOM Event Simulation

An event object can be created at any time by using the `createEvent()` method on `document`. This method accepts a single argument, which is a string indicating the type of event to create. In DOM Level 2, all of these strings were plural, while DOM Level 3 changed them to singular. The string may be one of the following:

- `UIEvents` — Generic UI event. Mouse events and keyboard events inherit from UI events. For DOM Level 3, use `UIEvent`.
- `MouseEvents` — Generic mouse event. For DOM Level 3, use `MouseEvent`.
- `MutationEvents` — Generic DOM mutation event. For DOM Level 3, use `MutationEvent`.
- `HTMLEvents` — Generic HTML event. There is no equivalent DOM Level 3 Event (HTML events were dispersed into other groupings).

Note that keyboard events are not specifically described in DOM Level 2 Events and were only later introduced in DOM Level 3 Events. Internet Explorer 9 is currently the only browser to support DOM Level 3 keyboard events. There are, however, ways to simulate keyboard events using methods that are available in other browsers.

Once an event object is created, it needs to be initialized with information about the event. Each type of event object has a specific method that is used to initialize it with the appropriate data. The name of the method is different, depending on the argument that was used with `createEvent()`.

The final step in event simulation is to fire the event. This is done by using the `dispatchEvent()` method that is present on all DOM nodes that support events. The `dispatchEvent()` method accepts a single argument, which is the `event` object representing the event to fire. After that point, the event becomes “official,” bubbling and causing event handlers to execute.

Simulating Mouse Events

Mouse events can be simulated by creating a new mouse event object and assigning the necessary information. A mouse event object is created by passing "MouseEvents" into the `createEvent()` method. The returned object has a method called `initMouseEvent()` that is used to assign mouse-related information. This method accepts 15 arguments, one for each property typically available on a mouse event. The arguments are as follows:

- `type` (string) — The type of event to fire, such as "click".
- `bubbles` (Boolean) — Indicates if the event should bubble. This should be set to `true` for accurate mouse event simulation.
- `cancelable` (Boolean) — Indicates if the event can be canceled. This should be set to `true` for accurate mouse event simulation.
- `view` (`AbstractView`) — The view associated with the event. This is almost always `document.defaultView`.
- `detail` (integer) — Additional information for the event. This is used only by event handlers, though it's typically set to 0.
- `screenX` (integer) — The x-coordinate of the event relative to the screen.
- `screenY` (integer) — The y-coordinate of the event relative to the screen.
- `clientX` (integer) — The x-coordinate of the event relative to the viewport.
- `clientY` (integer) — The y-coordinate of the event relative to the viewport.
- `ctrlKey` (Boolean) — Indicates if the Ctrl key is pressed. The default is `false`.
- `altKey` (Boolean) — Indicates if the Alt key is pressed. The default is `false`.
- `shiftKey` (Boolean) — Indicates if the Shift key is pressed. The default is `false`.
- `metaKey` (Boolean) — Indicates if the Meta key is pressed. The default is `false`.
- `button` (integer) — Indicates the button that was pressed. The default is 0.
- `relatedTarget` (object) — An object related to the event. This is used only when simulating `mouseover` or `mouseout`.

As should be obvious, the arguments for `initMouseEvent()` map directly to the `event` object properties for a mouse event. The first four arguments are the only ones that are critical for the proper execution of the event, because they are used by the browser; only event handlers use the



Available for
download on
Wrox.com

```
var btn = document.getElementById("myBtn");

//create event object
var event = document.createEvent("MouseEvents");

//initialize the event object
event.initMouseEvent("click", true, true, document.defaultView, 0, 0, 0, 0,
                     false, false, false, false, 0, null);

//fire the event
btn.dispatchEvent(event);
```

[SimulateDOMClickExample01.htm](#)

All other mouse events, including `dblclick`, can be simulated using this same technique in DOM-compliant browsers.

Simulating Keyboard Events

As mentioned previously, keyboard events were left out of DOM Level 2 Events, so simulating keyboard events is not straightforward. Keyboard events were included in draft versions of DOM Level 2 Events and were removed before finalization. Firefox implements the draft version of keyboard events. It's worth noting that keyboard events in DOM Level 3 are drastically different from the draft version originally included in DOM Level 2.

The DOM Level 3 way to create a keyboard event is by passing `"KeyboardEvent"` into the `createEvent()` method. Doing so creates an event object with a method called `initKeyboardEvent()`. This method has the following parameters:

- `type (string)` — The type of event to fire, such as `"keydown"`.
- `bubbles (Boolean)` — Indicates if the event should bubble. This should be set to `true` for accurate mouse event simulation.
- `cancelable (Boolean)` — Indicates if the event can be canceled. This should be set to `true` for accurate mouse event simulation.
- `view (AbstractView)` — The view associated with the event. This is almost always `document.defaultView`.
- `key (string)` — String code for the key that was pressed.
- `location (integer)` — The location of the key that was pressed. 0 for default keyboard, 1 for the left location, 2 for the right location, 3 for the numeric keypad, 4 for mobile (indicating a virtual keypad), or 5 for joystick.
- `modifiers (string)` — A space-separated list of modifiers such as `"Shift"`.
- `repeat (integer)` — The number of times this key has been pressed in a row.

Keep in mind that DOM Level 3 Events deprecates the `keypress` event, so you can simulate only the `keydown` and `keyup` events using this technique:



```
var textbox = document.getElementById("myTextbox"),
    event;

//create event object the DOM Level 3 way
if (document.implementation.hasFeature("KeyboardEvents", "3.0")) {
    event = document.createEvent("KeyboardEvent");

    //initialize the event object
    event.initKeyboardEvent("keydown", true, true, document.defaultView, "a",
                           0, "Shift", 0);
}

//fire the event
textbox.dispatchEvent(event);
```

[SimulateDOMKeyEventExample01.htm](#)

This example simulates `keydown` of the A key while Shift is being held. You should always check for DOM Level 3 keyboard events support before attempting to use `document.createEvent("KeyboardEvent")`; other browsers return a nonstandard `KeyboardEvent` object.

Firefox allows you to create a keyboard event by passing `"KeyEvents"` into the `createEvent()` method. This returns an event object with a method called `initKeyEvent()`, which accepts the following 10 arguments:

- `type` (string) — The type of event to fire, such as `"keydown"`.
- `bubbles` (Boolean) — Indicates if the event should bubble. This should be set to `true` for accurate mouse event simulation.
- `cancelable` (Boolean) — Indicates if the event can be canceled. This should be set to `true` for accurate mouse event simulation.
- `view` (`AbstractView`) — The view associated with the event. This is almost always `document.defaultView`.
- `ctrlKey` (Boolean) — Indicates if the Ctrl key is pressed. The default is `false`.
- `altKey` (Boolean) — Indicates if the Alt key is pressed. The default is `false`.
- `shiftKey` (Boolean) — Indicates if the Shift key is pressed. The default is `false`.
- `metaKey` (Boolean) — Indicates if the Meta key is pressed. The default is `false`.
- `keyCode` (integer) — The key code of the key that was pressed or released. This is used for `keydown` and `keyup`. The default is 0.
- `charCode` (integer) — The ASCII code of the character generated from the key press. This is used for `keypress`. The default is 0.

Available for
download on
Wrox.com

```
//for Firefox only
var textbox = document.getElementById("myTextbox");

//create event object
var event = document.createEvent("KeyEvents");

//initialize the event object
event.initKeyEvent("keydown", true, true, document.defaultView, false, false,
                   true, false, 65, 65);

//fire the event
textbox.dispatchEvent(event);
```

[SimulateFFKeyEventExample01.htm](#)

This example simulates keydown for the A key with the Shift key held down. You can also simulate keyup and keypress events using this technique.

For other browsers, you'll need to create a generic event and assign keyboard-specific information to it. Here is an example:

```
var textbox = document.getElementById("myTextbox");

//create event object
var event = document.createEvent("Events");

//initialize the event object
event.initEvent(type, bubbles, cancelable);
event.view = document.defaultView;
event.altKey = false;
event.ctrlKey = false;
event.shiftKey = false;
event.metaKey = false;
event.keyCode = 65;
event.charCode = 65;

//fire the event
textbox.dispatchEvent(event);
```

This code creates a generic event, initializes it by using `initEvent()`, and then assigns keyboard event information. It's necessary to use a generic event instead of a UI event because the UI event prevents new properties from being added to the event object (except in Safari). Simulating an event in this way causes the keyboard event to fire, but no text will be placed into the text box because this doesn't accurately simulate a keyboard event.

Simulating Other Events

Mouse events and keyboard events are the ones most often simulated in the browser, though it is possible to simulate mutation and HTML events as well. To simulate a mutation event, use `create`

`Event ("MutationEvents")` to create a new mutation event object with an `initMutationEvent()` method. The arguments of this event are `type`, `bubbles`, `cancelable`, `relatedNode`, `prevValue`, `newValue`, `attrName`, and `attrChange`. Simulating a mutation event takes the following form:

```
var event = document.createEvent("MutationEvents");
event.initMutationEvent("DOMNodeInserted", true, false, someNode, "", "", "", 0);
target.dispatchEvent(event);
```

This code simulates a `DOMNodeInserted` event. All other mutation events can be simulated using the same basic code and changing the arguments.

HTML events are simulated by creating an event object, using `createEvent ("HTMLEvents")`, and then initializing the `event` object using `initEvent()`. Here's an example:

```
var event = document.createEvent("HTMLEvents");
event.initEvent("focus", true, false);
target.dispatchEvent(event);
```

This example fires the `focus` event on a given target. Other HTML events may be simulated the same way.



Mutation events and HTML events are rarely used in browsers because they are of limited utility.

Custom DOM Events

DOM Level 3 specifies a class of events called *custom events*. Custom events don't get fired natively by the DOM but are provided so that developers can create their own events. You create a new custom event by calling `createEvent ("CustomEvent")`. The returned object has a method called `initCustomEvent()`, which takes four arguments:

- `type` (string) — The type of event to fire, such as "keydown".
- `bubbles` (Boolean) — Indicates if the event should bubble.
- `cancelable` (Boolean) — Indicates if the event can be canceled.
- `detail` (object) — Any value. This fills in the `detail` property of the `event` object.

The created event can then be dispatched in the DOM just like any other event. For example:



Available for
download on
Wrox.com

```
var div = document.getElementById("myDiv"),
    event;
EventUtil.addHandler(div, "myevent", function(event) {
    alert("DIV: " + event.detail);
});
EventUtil.addHandler(document, "myevent", function(event) {
    alert("DOCUMENT: " + event.detail);
});

if (document.implementation.hasFeature("CustomEvents", "3.0")) {
```

```

        event = document.createEvent("CustomEvent");
        event.initCustomEvent("myevent", true, false, "Hello world!");
        div.dispatchEvent(event);
    }

```

[SimulateDOMCustomEventExample01.htm](#)

This example creates a bubbling event called "myevent". The value of `event.detail` is set to a simple string and is then listened for both on a `<div>` element and at the `document` level. Because the event is specified as bubbling using `initCustomEvent()`, the browser takes care of bubbling the event up to the `document`.

Custom DOM events are supported only in Internet Explorer 9+ and Firefox 6+.

Internet Explorer Event Simulation

Event simulation in Internet Explorer 8 and earlier follows a similar pattern as event simulation in the DOM: you create an `event` object, assign the appropriate information, and then fire the event using the object. Of course, Internet Explorer has different ways of doing each step.

The `createEventObject()` method of `document` creates an `event` object. Unlike the DOM, this method accepts no arguments and returns a generic `event` object. After that, you must manually assign all of the properties that you want to have on the object. (There is no method to do this.) The last step is to call `fireEvent()` on the target, which accepts two arguments: the name of the event handler and the `event` object. When `fireEvent()` is called, the `srcElement` and `type` properties are automatically assigned to the `event` object; all other properties must be manually assigned. This means that all events that Internet Explorer supports are simulated using the same algorithm. For example, the following fires a `click` event on a button:



Available for
download on
Wrox.com

```

var btn = document.getElementById("myBtn");

//create event object
var event = document.createEventObject();

//initialize the event object
event.screenX = 100;
event.screenY = 0;
event.clientX = 0;
event.clientY = 0;
event.ctrlKey = false;
event.altKey = false;
event.shiftKey = false;
event.button = 0;

//fire the event
btn.fireEvent("onclick", event);

```

[SimulateIEClickExample01.htm](#)

This example creates an `event` object and then initializes it with some information. Note that property assignment is free-form, so you can assign any properties you'd like, including those not

normally supported by Internet Explorer 8 and earlier. The property values are of no consequence to the event, because only event handlers use them.

The same algorithm can be used to fire a `keypress` event as well, as shown in this example:



Available for
download on
Wrox.com

```
var textbox = document.getElementById("myTextbox");

//create event object
var event = document.createEventObject();

//initialize the event object
event.altKey = false;
event.ctrlKey = false;
event.shiftKey = false;
event.keyCode = 65;

//fire the event
textbox.fireEvent("onkeypress", event);
```

SimulateIEKeyEventExample01.htm

Since there is no difference between event objects for mouse, keyboard, or other events, a generic event object can be used to fire any type of event. Note that, as with DOM keyboard event simulation, no characters will appear in a text box as the result of a simulated `keypress` event even though the event handler will fire.

SUMMARY

Events are the primary way that JavaScript is tied to web pages. Most common events are defined in the DOM Level 3 Events specification or in HTML5. Even though there is a specification for basic events, many browsers have gone beyond the specification and implemented proprietary events to give developers greater insight into user interactions. Some proprietary events are directly related to specific devices, such as the mobile Safari `orientationchange` event that is specific to iOS devices.

There are some memory and performance considerations surrounding events. For example:

- It's best to limit the number of event handlers on a page, since they can take up more memory and make the page feel less responsive to the user.
- Event delegation can be used to limit the number of event handlers by taking advantage of event bubbling.
- It's a good idea to remove all event handlers that were added before the page is unloaded.

It's possible to simulate events in the browser using JavaScript. The DOM Level 2 and 3 Events specifications provide for the simulation of all events, making it easy to simulate all defined events. It's also possible to simulate keyboard events to a point by using a combination of other techniques. Internet Explorer 8 and earlier also support event simulation, albeit through a different interface.

Events are one of the most important topics in JavaScript, and a good understanding of how they work and their performance implications is critical.

14

Scripting Forms

WHAT'S IN THIS CHAPTER?

- Understanding form basics
- Text box validation and interaction
- Working with other form controls

One of the original uses of JavaScript was to offload some form-processing responsibilities onto the browser instead of relying on the server to do it all. Although the Web and JavaScript have evolved since that time, web forms remain more or less unchanged. The failure of web forms to provide out-of-the-box solutions for common problems led developers to use JavaScript not just for form validation but also to augment the default behavior of standard form controls.

FORM BASICS

Web forms are represented by the `<form>` element in HTML and by the `HTMLFormElement` type in JavaScript. The `HTMLFormElement` type inherits from `HTMLElement` and therefore has all of the same default properties as other HTML elements. However, `HTMLFormElement` also has the following additional properties and methods:

- `acceptCharset` — The character sets that the server can process; equivalent to the HTML `accept-charset` attribute.
- `action` — The URL to send the request to; equivalent to the HTML `action` attribute.
- `elements` — An `HTMLCollection` of all controls in the form.
- `enctype` — The encoding type of the request; equivalent to the HTML `enctype` attribute.
- `length` — The number of controls in the form.

- `method` — The type of HTTP request to send, typically "get" or "post"; equivalent to the HTML `method` attribute.
- `name` — The name of the form; equivalent to the HTML `name` attribute.
- `reset()` — Resets all form fields to their default values.
- `submit()` — Submits the form.
- `target` — The name of the window to use for sending the request and receiving the response; equivalent to the HTML `target` attribute.

References to `<form>` elements can be retrieved in a number of different ways. The most common way is to treat them as any other elements and assign the `id` attribute, allowing the use of `getElementById()`, as in the following example:

```
var form = document.getElementById("form1");
```

All forms on the page can also be retrieved from `document.forms` collection. Each form can be accessed in this collection by numeric index and by name, as shown in the following examples:

```
var firstForm = document.forms[0];           //get the first form in the page
var myForm = document.forms["form2"];         //get the form with a name of "form2"
```

Older browsers, or those with strict backwards compatibility, also add each form with a name as a property of the `document` object. For instance, a form named "form2" could be accessed via `document.form2`. This approach is not recommended, because it is error-prone and may be removed from browsers in the future.

Note that forms can have both an `id` and a `name` and that these values need not be the same.

Submitting Forms

Forms are submitted when a user interacts with a submit button or an image button. Submit buttons are defined using either the `<input>` element or the `<button>` element with a `type` attribute of "submit", and image buttons are defined using the `<input>` element with a `type` attribute of "image". All of the following, when clicked, will submit a form in which the button resides:

```
<!-- generic submit button -->
<input type="submit" value="Submit Form">

<!-- custom submit button -->
<button type="submit">Submit Form</button>

<!-- image button -->
<input type="image" src="graphic.gif">
```

If any one of these types of buttons is within a form that has a submit button, pressing Enter on the keyboard while a form control has focus will also submit the form. (The one exception is a `textarea`, within which Enter creates a new line of text.) Note that forms without a submit button will not be submitted when Enter is pressed.

When a form is submitted in this manner, the `submit` event fires right before the request is sent to the server. This gives you the opportunity to validate the form data and decide whether to allow the form submission to occur. Preventing the event's default behavior cancels the form submission. For example, the following prevents a form from being submitted:

```
var form = document.getElementById("myForm");
EventUtil.addHandler(form, "submit", function(event) {

    //get event object
    event = EventUtil.getEvent(event);

    //prevent form submission
    EventUtil.preventDefault(event);
});
```

This code uses the `EventUtil` object from the previous chapter to provide cross-browser event handling. The `preventDefault()` method stops the form from being submitted. Typically, this functionality is used when data in the form is invalid and should not be sent to the server.

It's possible to submit a form programmatically by calling the `submit()` method from JavaScript. This method can be called at any time to submit a form and does not require a submit button to be present in the form to function appropriately. Here's an example:

```
var form = document.getElementById("myForm");

//submit the form
form.submit();
```

When a form is submitted via `submit()`, the `submit` event does not fire, so be sure to do data validation before calling the method.

One of the biggest issues with form submission is the possibility of submitting the form twice. Users sometimes get impatient when it seems like nothing is happening and may click a submit button multiple times. The results can be annoying (because the server processes duplicate requests) or damaging (if the user is attempting a purchase and ends up placing multiple orders). There are essentially two ways to solve this problem: disable the submit button once the form is submitted, or use the `onsubmit` event handler to cancel any further form submissions.

Resetting Forms

Forms are reset when the user clicks a reset button. Reset buttons are created using either the `<input>` or the `<button>` element with a `type` attribute of `"reset"`, as in these examples:

```
<!-- generic reset button -->
<input type="reset" value="Reset Form">

<!-- custom reset button -->
<button type="reset">Reset Form</button>
```

Either of these buttons will reset a form. When a form is reset, all of the form fields are set back to the values they had when the page was first rendered. If a field was originally blank, it becomes blank again, whereas a field with a default value reverts to that value.

When a form is reset by the user clicking a reset button, the `reset` event fires. This event gives you the opportunity to cancel the reset if necessary. For example, the following prevents a form from being reset:

```
var form = document.getElementById("myForm");
EventUtil.addHandler(form, "reset", function(event) {

    //get event object
    event = EventUtil.getEvent(event);

    //prevent form reset
    EventUtil.preventDefault(event);
});
```

As with form submission, resetting a form can be accomplished via JavaScript using the `reset()` method, as in this example:

```
var form = document.getElementById("myForm");

//reset the form
form.reset();
```

Unlike the `submit()` method's functionality, `reset()` fires the `reset` event the same as if a reset button were clicked.



Form resetting is typically a frowned-upon approach to web form design. It's often disorienting to the user and, when triggered accidentally, can be quite frustrating. There's almost never a need to reset a form. It's often enough to provide a cancel button that takes the user back to the previous page rather than explicitly to revert all values in the form.

Form Fields

Form elements can be accessed in the same ways as any other elements on the page using native DOM methods. Additionally, all form elements are parts of an `elements` collection that is a property of each form. The `elements` collection is an ordered list of references to all form fields in the form and includes all `<input>`, `<textarea>`, `<button>`, `<select>`, and `<fieldset>` elements. Each form field appears in the `elements` collection in the order in which it appears in the markup, indexed by both position and name. Here are some examples:

```

var form = document.getElementById("form1");

//get the first field in the form
var field1 = form.elements[0];

//get the field named "textbox1"
var field2 = form.elements["textbox1"];

//get the number of fields
var fieldCount = form.elements.length;

```

If a name is in use by multiple form controls, as is the case with radio buttons, then an `HTMLCollection` is returned containing all of the elements with the name. For example, consider the following HTML snippet:



Available for
download on
Wrox.com

```

<form method="post" id="myForm">
    <ul>
        <li><input type="radio" name="color" value="red">Red</li>
        <li><input type="radio" name="color" value="green">Green</li>
        <li><input type="radio" name="color" value="blue">Blue</li>
    </ul>
</form>

```

[FormFieldsExample01.htm](#)

The form in this HTML has three radio controls that have "color" as their name, which ties the fields together. When accessing `elements["color"]`, a `NodeList` is returned, containing all three elements; when accessing `elements[0]`, however, only the first element is returned. Consider this example:

```

var form = document.getElementById("myForm");

var colorFields = form.elements["color"];
alert(colorFields.length); //3

var firstColorField = colorFields[0];
var firstFormField = form.elements[0];
alert(firstColorField === firstFormField); //true

```

[FormFieldsExample01.htm](#)

This code shows that the first form field, accessed via `form.elements[0]`, is the same as the first element contained in `form.elements["color"]`.



It's possible to access elements as properties of a form as well, such as `form[0]` to get the first form field and `form["color"]` to get a named field. These properties always return the same thing as their equivalent in the `elements` collection. This approach is provided for backwards compatibility with older browsers and should be avoided when possible in favor of using `elements`.

Common Form-Field Properties

With the exception of the `<fieldset>` element, all form fields share a common set of properties. Since the `<input>` type represents many form fields, some properties are used only with certain field types, whereas others are used regardless of the field type. The common form-field properties and methods are as follows:

- `disabled` — A Boolean indicating if the field is disabled.
- `form` — A pointer to the form that the field belongs to. This property is read only.
- `name` — The name of the field.
- `readOnly` — A Boolean indicating if the field is read only.
- `tabIndex` — Indicates the tab order for the field.
- `type` — The type of the field: "checkbox", "radio", and so on.
- `value` — The value of the field that will be submitted to the server. For file-input fields, this property is read only and simply contains the file's path on the computer.

With the exception of the `form` property, JavaScript can change all other properties dynamically. Consider this example:

```
var form = document.getElementById("myForm");
var field = form.elements[0];

//change the value
field.value = "Another value";

//check the value of form
alert(field.form === form);    //true

//set focus to the field
field.focus();

//disable the field
field.disabled = true;

//change the type of field (not recommended, but possible for <input>)
field.type = "checkbox";
```

The ability to change form-field properties dynamically allows you to change the form at any time and in almost any way. For example, a common problem with web forms is users' tendency to click the submit button twice. This is a major problem when credit-card orders are involved, because it may result in duplicate charges. A very common solution to this problem is to disable the submit button once it's been clicked, which is possible by listening for the `submit` event and disabling the submit button when it occurs. The following code accomplishes this:



```
//Code to prevent multiple form submissions
EventUtil.addHandler(form, "submit", function(event) {
    event = EventUtil.getEvent(event);
```

```

var target = EventUtil.getTarget(event);

//get the submit button
var btn = target.elements["submit-btn"];

//disable it
btn.disabled = true;

});

```

[FormFieldsExample02.htm](#)

This code attaches an event handler on the form for the `submit` event. When the event fires, the submit button is retrieved and its `disabled` property is set to `true`. Note that you cannot attach an `onclick` event handler to the submit button to do this because of a timing issue across browsers: some browsers fire the `click` event before the form's `submit` event, some after. For browsers that fire `click` first, the button will be disabled before the submission occurs, meaning that the form will never be submitted. Therefore it's better to disable the submit button using the `submit` event. This approach won't work if you are submitting the form without using a submit button, because, as stated before, the `submit` event is fired only by a submit button.

The `type` property exists for all form fields except `<fieldset>`. For `<input>` elements, this value is equal to the HTML `type` attribute. For other elements, the value of `type` is set as described in the following table.

DESCRIPTION	SAMPLE HTML	VALUE OF TYPE
Single-select list	<code><select>...</select></code>	"select-one"
Multi-select list	<code><select multiple>...</select></code>	"select-multiple"
Custom button	<code><button>...</button></code>	"submit"
Custom nonsubmit button	<code><button type="button">...</button></code>	"button"
Custom reset button	<code><button type="reset">...</button></code>	"reset"
Custom submit button	<code><button type="submit">...</button></code>	"submit"

For `<input>` and `<button>` elements, the `type` property can be changed dynamically, whereas the `<select>` element's `type` property is read only.

Common Form-Field Methods

Each form field has two methods in common: `focus()` and `blur()`. The `focus()` method sets the browser's focus to the form field, meaning that the field becomes active and will respond to keyboard events. For example, a text box that receives focus displays its caret and is ready to accept input. The `focus()` method is most often employed to call the user's attention to some part of the page. It's quite common, for instance, to have the focus moved to the first field in a form when

the page is loaded. This can be accomplished by listening for the `load` event and then calling `focus()` on the first field, as in the following example:

```
EventUtil.addHandler(window, "load", function(event){
    document.forms[0].elements[0].focus();
});
```

Note that this code will cause an error if the first form field is an `<input>` element with a `type` of "hidden" or if the field is being hidden using the `display` or `visibility` CSS property.

HTML5 introduces an `autofocus` attribute for form fields that causes supporting browsers to automatically set the focus to that element without the use of JavaScript. For example:

```
<input type="text" autofocus>
```

In order for the previous code to work correctly with `autofocus`, you must first detect if it has been set and, if so, not call `focus()`:



```
EventUtil.addHandler(window, "load", function(event){
    var element = document.forms[0].elements[0];
    if (element.autofocus !== true){
        element.focus(); console.log("JS focus");
    }
});
```

FocusExample01.htm

Because `autofocus` is a Boolean attribute, the value of the `autofocus` property will be `true` in supporting browsers. (It will be the empty string in browsers without support.) So this code calls `focus()` only if the `autofocus` property is not equal to `true`, ensuring forwards compatibility. The `autofocus` property is supported in Firefox 4+, Safari 5+, Chrome, and Opera 9.6+.



By default, only form elements can have focus set to them. It's possible to allow any element to have focus by setting its `tabIndex` property to -1 and then calling `focus()`. The only browser that doesn't support this technique is Opera.

The opposite of `focus()` is `blur()`, which removes focus from the element. When `blur()` is called, focus isn't moved to any element in particular; it's just removed from the field on which it was called. This method was used early in web development to create read-only fields before the `readonly` attribute was introduced. There's rarely a need to call `blur()`, but it's available if necessary. Here's an example:

```
document.forms[0].elements[0].blur();
```

Common Form-Field Events

All form fields support the following three events in addition to mouse, keyboard, mutation, and HTML events:

- `blur` — Fires when the field loses focus.
- `change` — Fires when the field loses focus and the `value` has changed for `<input>` and `<textarea>` elements; also fires when the selected option changes for `<select>` elements.
- `focus` — Fires when the field gets focus.

Both the `blur` and the `focus` events fire because of users manually changing the field's focus, as well as by calling the `blur()` and `focus()` methods, respectively. These two events work the same way for all form fields. The `change` event, however, fires at different times for different controls. For `<input>` and `<textarea>` elements, the `change` event fires when the field loses focus and the `value` has changed since the time the control got focus. For `<select>` elements, however, the `change` event fires whenever the user changes the selected option; the control need not lose focus for `change` to fire.

The `focus` and `blur` events are typically used to change the user interface in some way, to provide either visual cues or additional functionality (such as showing a drop-down menu of options for a text box). The `change` event is typically used to validate data that was entered into a field. For example, consider a text box that expects only numbers to be entered. The `focus` event may be used to change the background color to more clearly indicate that the field has focus, the `blur` event can be used to remove that background color, and the `change` event can change the background color to red if nonnumeric characters are entered. The following code accomplishes this:



Available for
download on
[Wrox.com](#)

```
var textbox = document.forms[0].elements[0];

EventUtil.addHandler(textbox, "focus", function(event) {
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    if (target.style.backgroundColor != "red"){
        target.style.backgroundColor = "yellow";
    }
});

EventUtil.addHandler(textbox, "blur", function(event) {
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    if (/[^\\d]/.test(target.value)){
        target.style.backgroundColor = "red";
    } else {
        target.style.backgroundColor = "";
    }
});

EventUtil.addHandler(textbox, "change", function(event) {
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    if (/[^\\d]/.test(target.value)){
```

```

        target.style.backgroundColor = "red";
    } else {
        target.style.backgroundColor = "";
    }
});

```

FormFieldEventsExample01.htm

The `onfocus` event handler simply changes the background color of the text box to yellow, more clearly indicating that it's the active field. The `onblur` and `onchange` event handlers turn the background color red if any nonnumeric character is found. To test for a nonnumeric character, use a simple regular expression against the text box's value. This functionality has to be in both the `onblur` and `onchange` event handlers to ensure that the behavior remains consistent regardless of text box changes.



The relationship between the blur and the change events is not strictly defined. In some browsers, the blur event fires before change; in others, it's the opposite. You can't depend on the order in which these events fire, so use care whenever they are required.

SCRIPTING TEXT BOXES

There are two ways to represent text boxes in HTML: a single-line version using the `<input>` element and a multiline version using `<textarea>`. These two controls are very similar and behave in similar ways most of the time. There are, however, some important differences.

By default, the `<input>` element displays a text box, even when the `type` attribute is omitted (the default value is "text"). The `size` attribute can then be used to specify how wide the text box should be in terms of visible characters. The `value` attribute specifies the initial value of the text box, and the `maxlength` attribute specifies the maximum number of characters allowed in the text box. So to create a text box that can display 25 characters at a time but has a maximum length of 50, you can use the following code:

```
<input type="text" size="25" maxlength="50" value="initial value">
```

The `<textarea>` element always renders a multiline text box. To specify how large the text box should be, you can use the `rows` attribute, which specifies the height of the text box in number of characters, and the `cols` attribute, which specifies the width in number of characters, similar to `size` for an `<input>` element. Unlike `<input>`, the initial value of a `<textarea>` must be enclosed between `<textarea>` and `</textarea>`, as shown here:

```
<textarea rows="25" cols="5">initial value</textarea>
```

Also unlike the `<input>` element, a `<textarea>` cannot specify the maximum number of characters allowed using HTML.

Despite the differences in markup, both types of text boxes store their contents in the `value` property. The value can be used to read the text box value and to set the text box value, as in this example:

```
var textbox = document.forms[0].elements["textbox1"];
alert(textbox.value);

textbox.value = "Some new value";
```

It's recommended to use the `value` property to read or write text box values rather than to use standard DOM methods. For instance, don't use `setAttribute()` to set the `value` attribute on an `<input>` element, and don't try to modify the first child node of a `<textarea>` element. Changes to the `value` property aren't always reflected in the DOM either, so it's best to avoid using DOM methods when dealing with text box values.

Text Selection

Both types of text boxes support a method called `select()`, which selects all of the text in a text box. Most browsers automatically set focus to the text box when the `select()` method is called (Opera does not). The method accepts no arguments and can be called at any time. Here's an example:

```
var textbox = document.forms[0].elements["textbox1"];
textbox.select();
```

It's quite common to select all of the text in a text box when it gets focus, especially if the text box has a default value. The thinking is that it makes life easier for users when they don't have to delete text separately. This pattern is accomplished with the following code:



Available for
download on
[Wrox.com](#)

```
EventUtil.addHandler(textbox, "focus", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    target.select();
});
```

[TextboxSelectExample01.htm](#)

With this code applied to a text box, all of the text will be selected as soon as the text box gets focus. This can greatly aid the usability of forms.

The `select` Event

To accompany the `select()` method, there is a `select` event. The `select` event fires when text is selected in the text box. Exactly when the event fires differs from browser to browser. In Internet Explorer 9+, Opera, Firefox, Chrome, and Safari, the `select` event fires once the user has finished



```
var textbox = document.forms[0].elements["textbox1"];
EventUtil.addHandler(textbox, "select", function(event){
    var      alert("Text selected: " + textbox.value);
});
```

[SelectEventExample01.htm](#)

Retrieving Selected Text

Although useful for understanding when text is selected, the `select` event provides no information about what text has been selected. HTML5 solved this issue by introducing some extensions to allow for better retrieval of selected text. The specification approach adds two properties to text boxes: `selectionStart` and `selectionEnd`. These properties contain zero-based numbers indicating the text-selection boundaries (the offset of the beginning of text selection and the offset of end of text selection, respectively). So, to get the selected text in a text box, you can use the following code:

```
function getSelectedText(textbox) {
    return textbox.value.substring(textbox.selectionStart, textbox.selectionEnd);
}
```

Since the `substring()` method works on string offsets, the values from `selectionStart` and `selectionEnd` can be passed in directly to retrieve the selected text.

This solution works for Internet Explorer 9+, Firefox, Safari, Chrome, and Opera. Internet Explorer 8 and earlier don't support these properties, so a different approach is necessary.

Older versions of Internet Explorer have a `document.selection` object that contains text-selection information for the entire document, which means you can't be sure where the selected text is on the page. When used in conjunction with the `select` event, however, you can be assured that the selection is inside the text box that fired the event. To get the selected text, you must first create a range (discussed in Chapter 12) and then extract the text from it, as in the following:

```
function getSelectedText(textbox) {
    if (typeof textbox.selectionStart == "number"){
        return textbox.value.substring(textbox.selectionStart,
                                       textbox.selectionEnd);
    } else if (document.selection){
        return document.selection.createRange().text;
    }
}
```

[TextboxGetSelectedTextExample01.htm](#)

This function has been modified to determine whether to use the Internet Explorer approach to selected text. Note that `document.selection` doesn't need the `textbox` argument at all.

Partial Text Selection

HTML5 also specifies an addition to aid in partially selecting text in a text box. The `setSelectionRange()` method, originally implemented by Firefox, is now available on all text boxes in addition to the `select()` method. This method takes two arguments: the index of the first character to select and the index at which to stop the selection (the same as the string's `substring()` method). Here are some examples:

```
textbox.value = "Hello world!"  
  
//select all text  
textbox.setSelectionRange(0, textbox.value.length);      // "Hello world!"  
  
//select first three characters  
textbox.setSelectionRange(0, 3);      // "Hel"  
  
//select characters 4 through 6  
textbox.setSelectionRange(4, 7);      // "o w"
```

To see the selection, you must set focus to the text box either immediately before or after a call to `setSelectionRange()`. This approach works for Internet Explorer 9, Firefox, Safari, Chrome, and Opera.

Internet Explorer 8 and earlier allow partial text selection through the use of ranges (discussed in Chapter 12). To select part of the text in a text box, you must first create a range and place it in the correct position by using the `createTextRange()` method that Internet Explorer provides on text boxes and using the `moveStart()` and `moveEnd()` range methods to move the range into position. Before calling these methods, however, you need to collapse the range to the start of the text box using `collapse()`. After that, `moveStart()` moves both the starting and the end points of the range to the same position. You can then pass in the total number of characters to select as the argument to `moveEnd()`. The last step is to use the range's `select()` method to select the text, as shown in these examples:

```
textbox.value = "Hello world!";  
  
var range = textbox.createTextRange();  
  
//select all text  
range.collapse(true);  
range.moveStart("character", 0);  
range.moveEnd("character", textbox.value.length);      // "Hello world!"  
range.select();  
  
//select first three characters  
range.collapse(true);  
range.moveStart("character", 0);  
range.moveEnd("character", 3);  
range.select();      // "Hel"  
  
//select characters 4 through 6  
range.collapse(true);  
range.moveStart("character", 4);  
range.moveEnd("character", 3);  
range.select();      // "o w"
```

As with the other browsers, the text box must have focus in order for the selection to be visible.

These two techniques can be combined into a single function for cross-browser usage, as in the following example:



```
function selectText(textbox, startIndex, stopIndex) {
    if (textbox.setSelectionRange) {
        textbox.setSelectionRange(startIndex, stopIndex);
    } else if (textbox.createTextRange) {
        var range = textbox.createTextRange();
        range.collapse(true);
        range.moveStart("character", startIndex);
        range.moveEnd("character", stopIndex - startIndex);
        range.select();
    }
    textbox.focus();
}
```

TextboxPartialSelectionExample01.htm

The `selectText()` function accepts three arguments: the text box to act on, the index at which to begin the selection, and the index before which to end the selection. First, the text box is tested to determine if it has the `setSelectionRange()` method. If so, that method is used.

If `setSelectionRange()` is not available, then the text box is checked to see if it supports `createTextRange()`. If `createTextRange()` is supported, then a range is created to accomplish the text selection. The last step in the method is to set the focus to the text box so that the selection will be visible. The `selectText()` method can be used as follows:

```
textbox.value = "Hello world!"

//select all text
selectText(textbox, 0, textbox.value.length);      // "Hello world!"

//select first three characters
selectText(textbox, 0, 3);      // "Hel"

//select characters 4 through 6
selectText(textbox, 4, 7);      // "o w"
```

Partial text selection is useful for implementing advanced text input boxes such as those that provide autocomplete suggestions.

Input Filtering

It's common for text boxes to expect a certain type of data or data format. Perhaps the data needs to contain certain characters or must match a particular pattern. Since text boxes don't offer much in the way of validation by default, JavaScript must be used to accomplish such *input filtering*. Using a combination of events and other DOM capabilities, you can turn a regular text box into one that understands the data it is dealing with.

Blocking Characters

Certain types of input require that specific characters be present or absent. For example, a text box for the user's phone number should not allow non-numeric values to be inserted. The `keypress` event is responsible for inserting characters into a text box. Characters can be blocked by preventing this event's default behavior. For example, the following code blocks all key presses:

```
EventUtil.addHandler(textbox, "keypress", function(event) {
    event = EventUtil.getEvent(event);
    EventUtil.preventDefault(event);
});
```

Running this code causes the text box to effectively become read only, because all key presses are blocked. To block only specific characters, you need to inspect the character code for the event and determine the correct response. For example, the following code allows only numbers:

```
EventUtil.addHandler(textbox, "keypress", function(event) {
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);
    var charCode = EventUtil.getCharCode(event);

    if (!/\d/.test(String.fromCharCode(charCode))) {
        EventUtil.preventDefault(event);
    }
});
```

In this example, the character code is retrieved using `EventUtil.getCharCode()` for cross-browser compatibility. The character code is converted to a string using `String.fromCharCode()`, and the result is tested against the regular expression `/\d/`, which matches all numeric characters. If that test fails, then the event is blocked using `EventUtil.preventDefault()`. This ensures that the text box ignores nonnumeric keys.

Even though the `keypress` event should be fired only when a character key is pressed, some browsers fire it for other keys as well. Firefox and Safari (versions prior to 3.1) fire `keypress` for keys like up, down, Backspace, and Delete; Safari versions 3.1 and later do not fire `keypress` events for these keys. This means that simply blocking all characters that aren't numbers isn't good enough, because you'll also be blocking these very useful and necessary keys. Fortunately, you can easily detect when one of these keys is pressed. In Firefox, all noncharacter keys that fire the `keypress` event have a character code of 0, whereas Safari versions prior to 3 give them all a character code of 8. To generalize the case, you don't want to block any character codes lower than 10. The function can then be updated as follows:

```
EventUtil.addHandler(textbox, "keypress", function(event) {
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);
    var charCode = EventUtil.getCharCode(event);

    if (!/\d/.test(String.fromCharCode(charCode)) && charCode > 9) {
        EventUtil.preventDefault(event);
    }
});
```

The event handler now behaves appropriately in all browsers, blocking nonnumeric characters but allowing all basic keys that also fire `keypress`.

There is still one more issue to handle: copying, pasting, and any other functions that involve the Ctrl key. In all browsers but Internet Explorer, the preceding code disallows the shortcut keystrokes of `Ctrl+C`, `Ctrl+V`, and any other combinations using the Ctrl key. The last check, therefore, is to make sure the Ctrl key is not pressed, as shown in the following example:



```
EventUtil.addHandler(textBox, "keypress", function(event) {
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);
    var charCode = EventUtil.getCharCode(event);

    if (!/\d/.test(String.fromCharCode(charCode)) && charCode > 9 &&
        !event.ctrlKey) {
        EventUtil.preventDefault(event);
    }
});
```

TextboxInputFilteringExample01.htm

This final change ensures that all of the default text box behaviors work. This technique can be customized to allow or disallow any characters in a text box.

Dealing with the Clipboard

Internet Explorer was the first browser to support events related to the clipboard and access to clipboard data from JavaScript. The Internet Explorer implementation became a de facto standard as Safari 2, Chrome, and Firefox 3 implemented similar events and clipboard access (Opera as of version 11 still doesn't have JavaScript clipboard support), and clipboard events were later added to HTML5. The following six events are related to the clipboard:

- `beforecopy` — Fires just before the copy operation takes place.
- `copy` — Fires when the copy operation takes place.
- `beforecut` — Fires just before the cut operation takes place.
- `cut` — Fires when the cut operation takes place.
- `beforepaste` — Fires just before the paste operation takes place.
- `paste` — Fires when the paste operation takes place.

Since this is a fairly new standard governing clipboard access, the behavior of the events and related objects differs from browser to browser. In Safari, Chrome, and Firefox, the `beforecopy`, `beforecut`, and `beforepaste` events fire only when the context menu for the text box is displayed (in anticipation of a clipboard event), but Internet Explorer fires them in that case and immediately before firing the `copy`, `cut`, and `paste` events. The `copy`, `cut`, and `paste` events all fire when you would expect them to in all browsers, both when the selection is made from a context menu and when using keyboard shortcuts.

The `beforecopy`, `beforecut`, and `beforepaste` events give you the opportunity to change the data being sent to or retrieved from the clipboard before the actual event occurs. However, canceling these events does not cancel the clipboard operation — you must cancel the `copy`, `cut`, or `paste` event to prevent the operation from occurring.

Clipboard data is accessible via the `clipboardData` object that exists either on the `window` object (in Internet Explorer) or on the `event` object (in Firefox 4+, Safari, and Chrome). In Firefox, Safari, and Chrome, the `clipboardData` object is available only during clipboard events to prevent unauthorized clipboard access; Internet Explorer exposes the `clipboardData` object all the time. For cross-browser compatibility, it's best to use this object only during clipboard events.

There are three methods on the `clipboardData` object: `getData()`, `setData()`, and `clearData()`. The `getData()` method retrieves string data from the clipboard and accepts a single argument, which is the format for the data to retrieve. Internet Explorer specifies two options: "text" and "URL". Firefox, Safari, and Chrome expect a MIME type but will accept "text" as equivalent to "text/plain".

The `setData()` method is similar: its first argument is the data type, and its second argument is the text to place on the clipboard. Once again, Internet Explorer supports "text" and "URL" whereas Safari and Chrome expect a MIME type. Unlike `getData()`, however, Safari and Chrome won't recognize the "text" type. Only Internet Explorer 8 and earlier honor `setData()`; other browsers simply ignore the call. To even out the differences, you can add the following cross-browser methods to `EventUtil`:



Available for download on Wrox.com

```
var EventUtil = {
    //more code here

    getClipboardText: function(event){
        var clipboardData = (event.clipboardData || window.clipboardData);
        return clipboardData.getData("text");
    },
    //more code here

    setClipboardText: function(event, value){
        if (event.clipboardData){
            return event.clipboardData.setData("text/plain", value);
        } else if (window.clipboardData){
            return window.clipboardData.setData("text", value);
        }
    },
    //more code here
};
```

[EventUtil.js](#)

The `getClipboardText()` method is relatively simple. It needs only to identify the location of the `clipboardData` object and then call `getData()` with a type of "text". Its companion method,

`setClipboardText()`, is slightly more involved. Once the `clipboardData` object is located, `setData()` is called with the appropriate type for each implementation ("text/plain" for Firefox, Safari, and Chrome; "text" for Internet Explorer).

Reading text from the clipboard is helpful when you have a text box that expects only certain characters or a certain format of text. For example, if a text box allows only numbers, then pasted values must also be inspected to ensure that the value is valid. In the `paste` event, you can determine if the text on the clipboard is invalid and, if so, cancel the default behavior, as shown in the following example:



```
EventUtil.addHandler(textbox, "paste", function(event) {
    event = EventUtil.getEvent(event);
    var text = EventUtil.getClipboardText(event);

    if (!/^\\d*$/.test(text)){
        EventUtil.preventDefault(event);
    }
});
```

TextboxClipboardExample01.htm

This `onpaste` handler ensures that only numeric values can be pasted into the text box. If the clipboard value doesn't match the pattern, then the paste is canceled. Firefox, Safari, and Chrome allow access to the `getData()` method only in an `onpaste` event handler.

Since not all browsers support clipboard access, it's often easier to block one or more of the clipboard operations. In browsers that support the `copy`, `cut`, and `paste` events (Internet Explorer, Safari, Chrome, and Firefox 3+), it's easy to prevent the events' default behavior. For Opera, you need to block the keystrokes that cause the events and block the context menu from being displayed.

Automatic Tab Forward

JavaScript can be used to increase the usability of form fields in a number of ways. One of the most common is to automatically move the focus to the next field when the current field is complete.

This is frequently done when entering data whose appropriate length is already known, such as for telephone numbers. In the United States, telephone numbers are typically split into three parts: the area code, the exchange, and then four more digits. It's quite common for web pages to represent this as three text boxes, such as the following:

```
<input type="text" name="tel1" id="txtTel1" maxlength="3">
<input type="text" name="tel2" id="txtTel2" maxlength="3">
<input type="text" name="tel3" id="txtTel3" maxlength="4">
```

TextboxTabForwardExample01.htm

To aid in usability and speed up the data-entry process, you can automatically move focus to the next element as soon as the maximum number of characters has been entered. So once the user

types three characters in the first text box, the focus moves to the second, and once the user types three characters in the second text box, the focus moves to the third. This “tab forward” behavior can be accomplished using the following code:



Available for
download on
Wrox.com

```
(function(){

    function tabForward(event){
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        if (target.value.length == target.maxLength) {
            var form = target.form;

            for (var i=0, len=form.elements.length; i < len; i++) {
                if (form.elements[i] == target) {
                    if (form.elements[i+1]){
                        form.elements[i+1].focus();
                    }
                    return;
                }
            }
        }
    }

    var textbox1 = document.getElementById("txtTel1");
    var textbox2 = document.getElementById("txtTel2");
    var textbox3 = document.getElementById("txtTel3");

    EventUtil.addHandler(textbox1, "keyup", tabForward);
    EventUtil.addHandler(textbox2, "keyup", tabForward);
    EventUtil.addHandler(textbox3, "keyup", tabForward);

})();
```

[TextboxTabForwardExample01.htm](#)

The `tabForward()` function is the key to this functionality. It checks to see if the text box’s maximum length has been reached by comparing the value to the `maxLength` attribute. If they’re equal (since the browser enforces the maximum, there’s no way it could be more), then the next form element needs to be found by looping through the elements collection until the text box is found and then setting focus to the element in the next position. This function is then assigned as the `onkeyup` handler for each text box. Since the `keyup` event fires after a new character has been inserted into the text box, this is the ideal time to check the length of the text box contents. When filling out this simple form, the user will never have to press the Tab key to move between fields and submit the form.

Keep in mind that this code is specific to the markup mentioned previously and doesn’t take into account possible hidden fields.

HTML5 Constraint Validation API

HTML5 introduces the ability for browsers to validate data in forms before submitting to the server. This capability enables basic validation even when JavaScript is unavailable or fails to load. The browser itself handles performing the validation based on rules in the code and then displays appropriate error messages on its own (without needing additional JavaScript). Of course, this functionality works only in browsers that support this part of HTML5, including Firefox 4+, Safari 5+, Chrome, and Opera 10+.

Validation is applied to a form field only under certain conditions. You can use HTML markup to specify constraints on a particular field that will result in the browser automatically performing form validation.

Required Fields

The first condition is when a form field has a `required` attribute, as in this example:

```
<input type="text" name="username" required>
```

Any field marked as `required` must have a value in order for the form to be submitted. This attribute applies to `<input>`, `<textarea>`, and `<select>` fields (Opera through version 11 doesn't support `required` on `<select>`). You can check to see if a form field is required in JavaScript by using the corresponding `required` property on the element:

```
var isUsernameRequired = document.forms[0].elements["username"].required;
```

You can also test to see if the browser supports the `required` attribute using this code snippet:

```
var isRequiredSupported = "required" in document.createElement("input");
```

This code uses simple feature detection to determine if the property `required` exists on a newly created `<input>` element.

Keep in mind that different browsers behave differently when a form field is required. Firefox 4 and Opera 11 prevent the form from submitting and pop up a help box beneath the field, while Safari (as of version 5) and Chrome (as of version 9) do nothing and don't prevent the form from submitting.

Alternate Input Types

HTML5 specifies several additional values for the `type` attribute on an `<input>` element. These `type` attributes not only provide additional information about the type of data expected but also provide some default validation. The two new input types that are mostly widely supported are `"email"` and `"url"`, and each comes with a custom validation that the browser applies. For example:

```
<input type="email" name="email">
<input type="url" name="homepage">
```

The `"email"` type ensures that the input text matches the pattern for an e-mail address, while the `"url"` type ensures that the input text matches the pattern for a URL. Note that the browsers

mentioned earlier in this section all have some issues with proper pattern matching. Most notably, the text "-@-" is considered a valid e-mail address. Such issues are still being addressed with browser vendors.

You can detect if a browser supports these new types by creating an element in JavaScript and setting the type property to "email" or "url" and then reading the value back. Older browsers automatically set unknown values back to "text", while supporting browsers echo the correct value back. For example:

```
var input = document.createElement("input");
input.type = "email";

var isEmailSupported = (input.type == "email");
```

Keep in mind that an empty field is also considered valid unless the `required` attribute is applied. Also, specifying a special input type doesn't prevent the user from entering an invalid value; it only applies some default validation.

Numeric Ranges

In addition to "email" and "url", there are several other new input element types defined in HTML5. These are all numeric types that expect some sort of numbers-based input: "number", "range", "datetime", "datetime-local", "date", "month", "week", and "time". These types are not well supported in browsers and as such should be used carefully, if at all. Browser vendors are working toward better cross-compatibility and more logical functionality at this time. Therefore, the information in this section is more forward looking rather than explanatory of existing functionality.

For each of these numeric types, you can specify a `min` attribute (the smallest possible value), a `max` attribute (the largest possible value), and a `step` attribute (the difference between individual steps along the scale from `min` to `max`). For instance, to allow only multiples of 5 between 0 and 100, you could use:

```
<input type="number" min="0" max="100" step="5" name="count">
```

Depending on the browser, you may or may not see a spin control (up and down buttons) to automatically increment or decrement the browser.

Each of the attributes have corresponding properties on the element that are accessible (and changeable) using JavaScript. Additionally, there are two methods: `stepUp()` and `stepDown()`. These methods each accept an optional argument: the number to either subtract or add from the current value. (By default, they increment or decrement by one.) The methods have not yet been implemented by browsers but will be usable as in this example:

```
input.stepUp();      //increment by one
input.stepUp(5);    //increment by five
input.stepDown();   //decrement by one
input.stepDown(10); //decrement by ten
```

Input Patterns

The `pattern` attribute was introduced for text fields in HTML5. This attribute specifies a regular expression with which the input value must match. For example, to allow only numbers in a text field, the following code applies this constraint:

```
<input type="text" pattern="\d+" name="count">
```

Note that `^` and `$` are assumed at the beginning and end of the pattern, respectively. That means the input must exactly match the pattern from beginning to end.

As with the alternate input types, specifying a `pattern` does not prevent the user from entering invalid text. The pattern is applied to the value, and the browser then knows if the value is valid or not. You can read the pattern by accessing the `pattern` property:

```
var pattern = document.forms[0].elements["count"].pattern;
```

You can also test to see if the browser supports the `pattern` attribute using this code snippet:

```
var isPatternSupported = "pattern" in document.createElement("input");
```

Checking Validity

You can check if any given field on the form is valid by using the `checkValidity()` method. This method is provided on all elements and returns `true` if the field's value is valid or `false` if not. Whether or not a field is valid is based on the conditions previously mentioned in this section, so a required field without a value is considered invalid, and a field whose value does not match the `pattern` attribute is considered invalid. For example:

```
if (document.forms[0].elements[0].checkValidity()) {
    //field is valid, proceed
} else {
    //field is invalid
}
```

To check if the entire form is valid, you can use the `checkValidity()` method on the form itself. This method returns `true` if all form fields are valid and `false` if even one is not:

```
if(document.forms[0].checkValidity()){
    //form is valid, proceed
} else {
    //form field is invalid
}
```

While `checkValidity()` simply tells you if a field is valid or not, the `validity` property indicates exactly why the field is valid or invalid. This object has a series of properties that return a Boolean value:

- `customError` — `true` if `setCustomValidity()` was set, `false` if not.
- `patternMismatch` — `true` if the value doesn't match the specified `pattern` attribute.

- `rangeOverflow` — `true` if the value is larger than the `max` value.
- `rangeUnderflow` — `true` if the value is smaller than the `min` value.
- `stepMismatch` — `true` if the value isn't correct given the `step` attribute in combination with `min` and `max`.
- `tooLong` — `true` if the value has more characters than allowed by the `maxlength` property. Some browsers, such as Firefox 4, automatically constrain the character count, and so this value may always be `false`.
- `typeMismatch` — `value` is not in the required format of either `"email"` or `"url"`.
- `valid` — `true` if every other property is `false`. Same value that is required by `checkValidity()`.
- `valueMissing` — `true` if the field is marked as required and there is no value.

Therefore, you may wish to check the validity of a form field using `validity` to get more specific information, as in the following code:

```
if (input.validity && !input.validity.valid) {
    if (input.validity.valueMissing) {
        alert("Please specify a value.");
    } else if (input.validity.typeMismatch) {
        alert("Please enter an email address.");
    } else {
        alert("Value is invalid.");
    }
}
```

Disabling Validation

You can instruct a form not to apply any validation to a form by specifying the `novalidate` attribute:

```
<form method="post" action="signup.php" novalidate

```

This value can also be retrieved or set by using the JavaScript property `noValidate`, which is set to `true` if the attribute is present and `false` if the attribute is omitted:

```
document.forms[0].noValidate = true; //turn off validation
```

If there are multiple submit buttons in a form, you can specify that the form not validate when a particular submit button is used by adding the `formnovalidate` attribute to the button itself:

```
<form method="post" action="foo.php">
    <!-- form elements here -->
    <input type="submit" value="Regular Submit">
    <input type="submit" formnovalidate name="btnNoValidate"
           value="Non-validating Submit">
</form>
```

In this example, the first submit button will cause the form to validate as usual while the second disables validation when submitting. You can also set this property using JavaScript:

```
//turn off validation
document.forms[0].elements["btnNoValidate"].formNoValidate = true;
```

SCRIPTING SELECT BOXES

Select boxes are created using the `<select>` and `<option>` elements. To allow for easier interaction with the control, the `HTMLSelectElement` type provides the following properties and methods in addition to those that are available on all form fields:

- `add(newOption, relOption)` — Adds a new `<option>` element to the control before the related option.
- `multiple` — A Boolean value indicating if multiple selections are allowed; equivalent to the HTML `multiple` attribute.
- `options` — An `HTMLCollection` of `<option>` elements in the control.
- `remove(index)` — Removes the option in the given position.
- `selectedIndex` — The zero-based index of the selected option or `-1` if no options are selected. For select boxes that allow multiple selections, this is always the first option in the selection.
- `size` — The number of rows visible in the select box; equivalent to the HTML `size` attribute.

The `type` property for a select box is either `"select-one"` or `"select-multiple"`, depending on the absence or presence of the `multiple` attribute. The option that is currently selected determines a select box's `value` property according to the following rules:

- If there is no option selected, the value of a select box is an empty string.
- If an option is selected and it has a `value` attribute specified, then the select box's value is the `value` attribute of the selected option. This is true even if the `value` attribute is an empty string.
- If an option is selected and it doesn't have a `value` attribute specified, then the select box's value is the text of the option.
- If multiple options are selected, then the select box's value is taken from the first selected option according to the previous two rules.

Consider the following select box:

```
<select name="location" id="selLocation">
  <option value="Sunnyvale, CA">Sunnyvale</option>
  <option value="Los Angeles, CA">Los Angeles</option>
  <option value="Mountain View, CA">Mountain View</option>
  <option value="">China</option>
  <option>Australia</option>
</select>
```

If the first option in this select box is selected, the value of the field is "Sunnyvale, CA". If the option with the text "China" is selected, then the field's value is an empty string because the `value` attribute is empty. If the last option is selected, then the value is "Australia" because there is no `value` attribute specified on the `<option>`.

Each `<option>` element is represented in the DOM by an `HTMLOptionElement` object. The `HTMLOptionElement` type adds the following properties for easier data access:

- `index` — The option's index inside the `options` collection.
- `label` — The option's label; equivalent to the HTML `label` attribute.
- `selected` — A Boolean value used to indicate if the option is selected. Set this property to `true` to select an option.
- `text` — The option's text.
- `value` — The option's value (equivalent to the HTML `value` attribute).

Most of the `<option>` properties are used for faster access to the option data. Normal DOM functionality can be used to access this information, but it's quite inefficient, as this example shows:

```
var selectbox = document.forms[0].elements["location"];

//not recommended
var text = selectbox.options[0].firstChild.nodeValue;      //option text
var value = selectbox.options[0].getAttribute("value");     //option value
```

This code gets the text and value of the first option in the select box using standard DOM techniques. Compare this to using the special option properties:

```
var selectbox = document.forms[0].elements["location"];

//preferred
var text = selectbox.options[0].text;      //option text
var value = selectbox.options[0].value;     //option value
```

When dealing with options, it's best to use the option-specific properties because they are well supported across all browsers. The exact interactions of form controls may vary from browser to browser when manipulating DOM nodes. It is not recommended to change the text or values of `<option>` elements by using standard DOM techniques.

As a final note, there is a difference in the way the `change` event is used for select boxes. As opposed to other form fields, which fire the `change` event after the value has changed and the field loses focus, the `change` event fires on select boxes as soon as an option is selected.



There are differences in what the `value` property returns across browsers. The `value` property is always equal to the `value` attribute in all browsers. When the `value` attribute is not specified, Internet Explorer 8 and earlier versions return an empty string, whereas Internet Explorer 9+, Safari, Firefox, Chrome, and Opera return the same value as `text`.

Options Selection

For a select box that allows only one option to be selected, the easiest way to access the selected option is by using the select box's `selectedIndex` property to retrieve the option, as shown in the following example:

```
var selectedOption = selectbox.options[selectbox.selectedIndex];
```

This can be used to display all of the information about the selected option, as in this example:



```
var selectedIndex = selectbox.selectedIndex;
var selectedOption = selectbox.options[selectedIndex];
alert("Selected index: " + selectedIndex + "\nSelected text: " +
      selectedOption.text + "\nSelected value: " + selectedOption.value);
```

SelectboxExample01.htm

Here, an alert is displayed showing the selected index along with the text and value of the selected option.

When used in a select box that allows multiple selections, the `selectedIndex` property acts as if only one selection was allowed. Setting `selectedIndex` removes all selections and selects just the single option specified, whereas getting `selectedIndex` returns only the index of the first option that was selected.

Options can also be selected by getting a reference to the option and setting its `selected` property to `true`. For example, the following selects the first option in a select box:

```
selectbox.options[0].selected = true;
```

Unlike `selectedIndex`, setting the option's `selected` property does not remove other selections when used in a multiselect select box, allowing you to dynamically select any number of options. If an option's `selected` property is changed in a single-select select box, then all other selections are removed. It's worth noting that setting the `selected` property to `false` has no effect in a single-select select box.

The `selected` property is helpful in determining which options in a select box are selected. To get all of the selected options, you can loop over the options collection and test the `selected` property. Consider this example:

```
function getSelectedOptions(selectbox) {
    var result = new Array();
    var option = null;

    for (var i=0, len=selectbox.options.length; i < len; i++) {
        option = selectbox.options[i];
        if (option.selected){
            result.push(option);
        }
    }
    return result;
}
```

SelectboxExample03.htm

This function returns an array of options that are selected in a given select box. First an array to contain the results is created. Then a `for` loop iterates over the options, checking each option's `selected` property. If the option is selected, it is added to the result array. The last step is to return the array of selected options. The `getSelectedOptions()` function can then be used to get information about the selected options, like this:



Available for download on
Wrox.com

```
var selectbox = document.getElementById("selLocation");
var selectedOptions = getSelectedOptions(selectbox);
var message = "";

for (var i=0, len=selectedOptions.length; i < len; i++){
    message += "Selected index: " + selectedOptions[i].index +
    "\nSelected text: " + selectedOptions[i].text +
    "\nSelected value: " + selectedOptions[i].value + "\n\n";
}

alert(message);
```

[SelectboxExample03.htm](#)

In this example, the selected options are retrieved from a select box. A `for` loop is used to construct a message containing information about all of the selected options, including each option's index, text, and value. This can be used for select boxes that allow single or multiple selection.

Adding Options

There are several ways to create options dynamically and add them to select boxes using JavaScript. The first way is to use the DOM as follows:

```
var newOption = document.createElement("option");
newOption.appendChild(document.createTextNode("Option text"));
newOption.setAttribute("value", "Option value");

selectbox.appendChild(newOption);
```

[SelectboxExample04.htm](#)

This code creates a new `<option>` element, adds some text using a text node, sets its `value` attribute, and then adds it to a select box. The new option shows up immediately after being created.

New options can also be created using the `Option` constructor, which is a holdover from pre-DOM browsers. The `Option` constructor accepts two arguments, the `text` and the `value`, though the second argument is optional. Even though this constructor is used to create an instance of `Object`, DOM-compliant browsers return an `<option>` element. This means you can still use `appendChild()` to add the option to the select box. Consider the following:

```
var newOption = new Option("Option text", "Option value");
selectbox.appendChild(newOption); //problems in IE <= 8
```

[SelectboxExample04.htm](#)

This approach works as expected in all browsers except Internet Explorer 8 and earlier. Because of a bug, the browser doesn't correctly set the text of the new option when using this approach.

Another way to add a new option is to use the select box's `add()` method. The DOM specifies that this method accepts two arguments: the new option to add and the option before which the new option should be inserted. To add an option at the end of the list, the second argument should be `null`. The Internet Explorer 8 and earlier implementation of `add()` is slightly different in that the second argument is optional, and it must be the index of the option before which to insert the new option. DOM-compliant browsers require the second argument, so you can't use just one argument for a cross-browser approach (Internet Explorer 9 is DOM-compliant). Instead, passing `undefined` as the second argument ensures that the option is added at the end of the list in all browsers. Here's an example:



```
var newOption = new Option("Option text", "Option value");
selectbox.add(newOption, undefined); //best solution
```

SelectboxExample04.htm

This code works appropriately in all versions of Internet Explorer and DOM-compliant browsers. If you need to insert a new option into a position other than last, you should use the DOM technique and `insertBefore()`.



As in HTML, you are not required to assign a value for an option. The `Option` constructor works with just one argument (the option text).

Removing Options

As with adding options, there are multiple ways to remove options. You can use the DOM `removeChild()` method and pass in the option to remove, as shown here:

```
selectbox.removeChild(selectbox.options[0]); //remove first option
```

The second way is to use the select box's `remove()` method. This method accepts a single argument, the index of the option to remove, as shown here:

```
selectbox.remove(0); //remove first option
```

The last way is to simply set the option equal to `null`. This is also a holdover from pre-DOM browsers. Here's an example:

```
selectbox.options[0] = null; //remove first option
```

To clear a select box of all options, you need to iterate over the options and remove each one, as in this example:

```
function clearSelectbox(selectbox){
    for(var i=0, len=selectbox.options.length; i < len; i++) {
```

```

        selectbox.remove(0);
    }
}

```

This function simply removes the first option in a select box repeatedly. Since removing the first option automatically moves all of the options up one spot, this removes all options.

Moving and Reordering Options

Before the DOM, moving options from one select box to another was a rather arduous process that involved removing the option from the first select box, creating a new option with the same name and value, and then adding that new option to the second select box. Using DOM methods, it's possible to literally move an option from the first select box into the second select box by using the `appendChild()` method. If you pass an element that is already in the document into this method, the element is removed from its parent and put into the position specified. For example, the following code moves the first option from one select box into another select box.



Available for
download on
Wrox.com

```

var selectbox1 = document.getElementById("selLocations1");
var selectbox2 = document.getElementById("selLocations2");
selectbox2.appendChild(selectbox1.options[0]);

```

[SelectboxExample05.htm](#)

Moving options is the same as removing them in that the `index` property of each option is reset.

Reordering options is very similar, and DOM methods are the best way to accomplish this. To move an option to a particular location in the select box, the `insertBefore()` method is most appropriate, though the `appendChild()` method can be used to move any option to the last position. To move an option up one spot in the select box, you can use the following code:

```

var optionToMove = selectbox.options[1];
selectbox.insertBefore(optionToMove, selectbox.options[optionToMove.index-1]);

```

[SelectboxExample06.htm](#)

In this code, an option is selected to move and then inserted before the option that is in the previous index. The second line of code is generic enough to work with any option in the select box except the first. The following similar code can be used to move an option down one spot:

```

var optionToMove = selectbox.options[1];
selectbox.insertBefore(optionToMove, selectbox.options[optionToMove.index+2]);

```

[SelectboxExample06.htm](#)

This code works for all options in a select box, including the last one.



There is a repainting issue in Internet Explorer 7 that sometimes causes options that are reordered using DOM methods to take a few seconds to display correctly.

FORM SERIALIZATION

With the emergence of Ajax (discussed further in Chapter 21), *form serialization* has become a common requirement. A form can be serialized in JavaScript using the `type` property of form fields in conjunction with the `name` and `value` properties. Before writing the code, you need to understand how the browser determines what gets sent to the server during a form submission:

- Field names and values are URL-encoded and delimited using an ampersand.
- Disabled fields aren't sent at all.
- A check box or radio field is sent only if it is checked.
- Buttons of type "reset" or "button" are never sent.
- Multiselect fields have an entry for each value selected.
- When the form is submitted by clicking a submit button, that submit button is sent; otherwise no submit buttons are sent. Any `<input>` elements with a `type` of "image" are treated the same as submit buttons.
- The value of a `<select>` element is the `value` attribute of the selected `<option>` element. If the `<option>` element doesn't have a `value` attribute, then the value is the text of the `<option>` element.

Form serialization typically doesn't include any button fields, because the resulting string will most likely be submitted in another way. All of the other rules should be followed. The code to accomplish form serialization is as follows:



```
function serialize(form) {
    var parts = [],
        field = null,
        i,
        len,
        j,
        optLen,
        option,
        optValue;

    for (i=0, len=form.elements.length; i < len; i++){
        field = form.elements[i];

        switch(field.type){
            case "select-one":
            case "select-multiple":

                if (field.name.length){
```

```

        for (j=0, optLen = field.options.length; j < optLen; j++){
            option = field.options[j];
            if (option.selected){
                optValue = "";
                if (option.hasAttribute){
                    optValue = (option.hasAttribute("value") ?
                                option.value : option.text);
                } else {
                    optValue = (option.attributes["value"].specified ?
                                option.value : option.text);
                }
                parts.push(encodeURIComponent(field.name) + "=" +
                           encodeURIComponent(optValue));
            }
        }
        break;
    }

    case undefined: //fieldset
    case "file": //file input
    case "submit": //submit button
    case "reset": //reset button
    case "button": //custom button
        break;

    case "radio": //radio button
    case "checkbox": //checkbox
        if (!field.checked){
            break;
        }
        /* falls through */

    default:
        //don't include form fields without names
        if (field.name.length){
            parts.push(encodeURIComponent(field.name) + "=" +
                       encodeURIComponent(field.value));
        }
    }
}

return parts.join("&");
}

```

[FormSerializationExample01.htm](#)

The `serialize()` function begins by defining an array called `parts` to hold the parts of the string that will be created. Next, a `for` loop iterates over each form field, storing it in the `field` variable. Once a field reference is obtained, its `type` is checked using a `switch` statement. The most involved field to serialize is the `<select>` element, in either single-select or multiselect mode. Serialization is done by looping over all of the options in the control and adding a value if the option is selected. For single-select controls, there will be only one option selected, whereas multiselect controls may

have zero or more options selected. The same code can be used for both select types, because the restriction on the number of selections is enforced by the browser. When an option is selected, you need to determine which value to use. If the `value` attribute is not present, the text should be used instead, although a `value` attribute with an empty string is completely valid. To check this, you'll need to use `hasAttribute()` in DOM-compliant browsers and the attribute's `specified` property in Internet Explorer 8 and earlier.

If a `<fieldset>` element is in the form, it appears in the elements collection but has no `type` property. So if `type` is `undefined`, no serialization is necessary. The same is true for all types of buttons and file input fields. (File input fields contain the content of the file in form submissions; however, these fields can't be mimicked, so they are typically omitted in serialization.) For radio and check box controls, the `checked` property is inspected and if it is set to `false`, the `switch` statement is exited. If `checked` is `true`, then the code continues executing in the `default` statement, which encodes the name and value of the field and adds it to the `parts` array. Note that in all cases form fields without names are not included as part of the serialization to mimic browser form submission behavior. The last part of the function uses `join()` to format the string correctly with ampersands between fields.

The `serialize()` function outputs the string in query string format, though it can easily be adapted to serialize the form into another format.

RICH TEXT EDITING

One of the most requested features for web applications was the ability to edit rich text on a web page (also called *what you see is what you get*, or WYSIWYG, editing). Though no specification covers this, a de facto standard has emerged from functionality originally introduced by Internet Explorer and now supported by Opera, Safari, Chrome, and Firefox. The basic technique is to embed an `iframe` containing a blank HTML file in the page. Through the `designMode` property, this blank document can be made editable, at which point you're editing the HTML of the page's `<body>` element. The `designMode` property has two possible values: "`off`" (the default) and "`on`". When set to "`on`", an entire document becomes editable (showing a caret), allowing you to edit text as if you were using a word processor complete with keystrokes for making text bold, italic, and so forth.

A very simple, blank HTML page is used as the source of the `iframe`. Here's an example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Blank Page for Rich Text Editing</title>
  </head>
  <body>
  </body>
</html>
```

This page is loaded inside an `iframe` as any other page would be. To allow it to be edited, you must set `designMode` to "`on`", but this can happen only after the document is fully loaded. In the

containing page, you'll need to use the `onload` event handler to indicate the appropriate time to set `designMode`, as shown in the following example:

```
<iframe name="richedit" style="height: 100px; width: 100px" src="blank.htm">
</iframe>

<script type="text/javascript">
EventUtil.addHandler(window, "load", function(){
    frames["richedit"].document.designMode = "on";
});
</script>
```

Once this code is loaded, you'll see what looks like a text box on the page. The box has the same default styling as any web page, though this can be adjusted by applying CSS to the blank page.

Using `contenteditable`

Another way to interact with rich text, also first implemented by Internet Explorer, is through the use of a special attribute called `contenteditable`. The `contenteditable` attribute can be applied to any element on a page and instantly makes that element editable by the user. This approach has gained favor because it doesn't require the overhead of an iframe, blank page, and JavaScript. Instead, you can just add the attribute to an element:

```
<div class="editable" id="richedit" contenteditable></div>
```

Any text already contained within the element is automatically made editable by the user, making it behave similarly to the `<textarea>` element. You can also toggle the editing mode on or off by setting the `contentEditable` property on an element:

```
var div = document.getElementById("richedit");
richedit.contentEditable = "true";
```

There are three possible values for `contentEditable`: `"true"` to turn on, `"false"` to turn off, or `"inherit"` to inherit the setting from a parent (required since elements can be created/destroyed inside of a `contenteditable` element). The `contentEditable` attribute is supported in Internet Explorer, Firefox, Chrome, Safari, and Opera. For mobile devices, `contenteditable` is supported on Safari for iOS 5+ and WebKit for Android 3+.

Interacting with Rich Text

The primary method of interacting with a rich text editor is through the use of `document.execCommand()`. This method executes named commands on the document and can be used to apply most formatting changes. There are three possible arguments for `document.execCommand()`: the name of the command to execute, a Boolean value indicating if the browser should provide a user interface for the command, and a value necessary for the command to work (or `null` if none is necessary). The second argument should always be `false` for cross-browser compatibility, because Firefox throws an error when `true` is passed in.

Each browser supports a different set of commands. The most commonly supported commands are listed in the following table.

COMMAND	VALUE (THIRD ARGUMENT)	DESCRIPTION
backcolor	A color string	Sets the background color of the document.
bold	null	Toggles bold text for the text selection.
copy	null	Executes a clipboard copy on the text selection.
createlink	A URL string	Turns the current text selection into a link that goes to the given URL.
cut	null	Executes a clipboard cut on the text selection.
delete	null	Deletes the currently selected text.
fontname	The font name	Changes the text selection to use the given font name.
fontsize	1 through 7	Changes the font size for the text selection.
forecolor	A color string	Changes the text color for the text selection.
formatblock	The HTML tag to surround the block with; for example, <h1>	Formats the entire text box around the selection with a particular HTML tag.
indent	null	Indents the text.
inserthorizontalrule	null	Inserts an <hr> element at the caret location.
insertimage	The image URL	Inserts an image at the caret location.
insertorderedlist	null	Inserts an element at the caret location.
insertparagraph	null	Inserts a <p> element at the caret location.
insertunorderedlist	null	Inserts a element at the caret location.
italic	null	Toggles italic text for the text selection.
justifycenter	null	Centers the block of text in which the caret is positioned.
justifyleft	null	Left-aligns the block of text in which the caret is positioned.
outdent	null	Outdents the text.

COMMAND	VALUE (THIRD ARGUMENT)	DESCRIPTION
paste	null	Executes a clipboard paste on the text selection.
removeformat	null	Removes block formatting from the block in which the caret is positioned. This is the opposite of <code>formatblock</code> .
selectall	null	Selects all of the text in the document.
underline	null	Toggles underlined text for the text selection.
unlink	null	Removes a text link. This is the opposite of <code>createlink</code> .

The clipboard commands are very browser-dependent. Opera doesn't implement any of the clipboard commands, and Firefox has them disabled by default. (You must change a user preference to enable them.) Safari and Chrome implement `cut` and `copy` but not `paste`. Note that even though these commands aren't available via `document.execCommand()`, they still work with the appropriate keyboard shortcuts.

These commands can be used at any time to modify the appearance of the iframe rich text area, as in this example:



```
//toggle bold text in an iframe
frames["richedit"].document.execCommand("bold", false, null);

//toggle italic text in an iframe
frames["richedit"].document.execCommand("italic", false, null);

//create link to www.wrox.com in an iframe
frames["richedit"].document.execCommand("createlink", false,
                                         "http://www.wrox.com");

//format as first-level heading in an iframe
frames["richedit"].document.execCommand("formatblock", false, "<h1>");
```

RichTextEditingExample01.htm

You can use the same methods to act on a `contenteditable` section of the page, just use the `document` object of the current window instead of referencing the iframe:

```
//toggle bold text
document.execCommand("bold", false, null);

//toggle italic text
```

```

document.execCommand("italic", false, null);

//create link to www.wrox.com
document.execCommand("createlink", false,
                      "http://www.wrox.com");

//format as first-level heading
document.execCommand("formatblock", false, "<h1>");

```

RichTextEditingExample01.htm

Note that even when commands are supported across all browsers, the HTML that the commands produce is often very different. For instance, applying the **bold** command surrounds text with **** in Internet Explorer and Opera, with **** in Safari and Chrome, and with a **** in Firefox. You cannot rely on consistency in the HTML produced from a rich text editor, because of both command implementation and the transformations done by `innerHTML`.

There are some other methods related to commands. The first is `queryCommandEnabled()`, which determines if a command can be executed given the current text selection or caret position. This method accepts a single argument, the command name to check, and returns `true` if the command is allowed given the state of the editable area or `false` if not. Consider this example:

```
var result = frames["richedit"].document.queryCommandEnabled("bold");
```

This code returns `true` if the "bold" command can be executed on the current selection. It's worth noting that `queryCommandEnabled()` indicates not if you are allowed to execute the command but only if the current selection is appropriate for use with the command. In Firefox, `queryCommandEnabled("cut")` returns `true` even though it isn't allowed by default.

The `queryCommandState()` method lets you determine if a given command has been applied to the current text selection. For example, to determine if the text in the current selection is bold, you can use the following:

```
var isBold = frames["richedit"].document.queryCommandState("bold");
```

RichTextEditingExample01.htm

If the "bold" command was previously applied to the text selection, then this code returns `true`. This is the method by which full-featured rich text editors are able to update buttons for bold, italic, and so on.

The last method is `queryCommandValue()`, which is intended to return the value with which a command was executed. (The third argument in `execCommand` is in the earlier example.) For instance, a range of text that has the "fontsize" command applied with a value of 7 returns "7" from the following:

```
var fontSize = frames["richedit"].document.queryCommandValue("fontsize");
```

RichTextEditingExample01.htm



This method can be used to determine how a command was applied to the text selection, allowing you to determine whether the next command is appropriate to be executed.

Rich Text Selections

You can determine the exact selection in a rich text editor by using the `getSelection()` method of the `iframe`. This method is available on both the `document` object and the `window` object and returns a `Selection` object representing the currently selected text. Each `Selection` object has the following properties:

- `anchorNode` — The node in which the selection begins.
- `anchorOffset` — The number of characters within the `anchorNode` that are skipped before the selection begins.
- `focusNode` — The node in which the selection ends.
- `focusOffset` — The number of characters within the `focusNode` that are included in the selection.
- `isCollapsed` — Boolean value indicating if the start and end of the selection are the same.
- `rangeCount` — The number of DOM ranges in the selection.

The properties for a `Selection` don't contain a lot of useful information. Fortunately, the following methods provide more information and allow manipulation of the selection:

- `addRange(range)` — Adds the given DOM range to the selection.
- `collapse(node, offset)` — Collapses the selection to the given text offset within the given node.
- `collapseToEnd()` — Collapses the selection to its end.
- `collapseToStart()` — Collapses the selection to its start.
- `containsNode(node)` — Determines if the given node is contained in the selection.
- `deleteFromDocument()` — Deletes the selection text from the document. This is the same as `execCommand("delete", false, null)`.
- `extend(node, offset)` — Extends the selection by moving the `focusNode` and `focusOffset` to the values specified.
- `getRangeAt(index)` — Returns the DOM range at the given index in the selection.
- `removeAllRanges()` — Removes all DOM ranges from the selection. This effectively removes the selection, because there must be at least one range in a selection.
- `removeRange(range)` — Removes the specified DOM range from the selection.
- `selectAllChildren(node)` — Clears the selection and then selects all child nodes of the given node.
- `toString()` — Returns the text content of the selection.

The methods of a Selection object are extremely powerful and make extensive use of DOM ranges (discussed in Chapter 12) to manage the selection. Access to DOM ranges allows you to modify the contents of the rich text editor in even finer-grain detail than is available using `execCommand()`, because you can directly manipulate the DOM of the selected text. Consider the following example:



```
var selection = frames["richedit"].getSelection();
//get selected text
var selectedText = selection.toString();

//get the range representing the selection
var range = selection.getRangeAt(0);

//highlight the selected text
var span = frames["richedit"].document.createElement("span");
span.style.backgroundColor = "yellow";
range.surroundContents(span);
```

RichTextEditingExample01.htm

This code places a yellow highlight around the selected text in a rich text editor. Using the DOM range in the default selection, the `surroundContents()` method surrounds the selection with a `` element whose background color is yellow.

The `getSelection()` method was standardized in HTML5 and is implemented in Internet Explorer 9, Firefox, Safari, Chrome, and Opera 8. Firefox 3.6+ incorrectly returns a string from `document.getSelection()` because of legacy support issues. You can retrieve a `Selection` object in Firefox 3.6+ by using the `window.getSelection()` method instead. Firefox 8 fixed `document.getSelection()` to return the same value as `window.getSelection()`.

Internet Explorer 8 and earlier versions don't support DOM ranges, but they do allow interaction with the selected text via the proprietary `selection` object. The `selection` object is a property of `document`, as discussed earlier in this chapter. To get the selected text in a rich text editor, you must first create a text range (discussed in Chapter 12) and then use the `text` property as follows:

```
var range = frames["richedit"].document.selection.createRange();
var selectedText = range.text;
```

Performing HTML manipulations using Internet Explorer text ranges is not as safe as using DOM ranges, but it is possible. To achieve the same highlighting effect as described using DOM ranges, you can use a combination of the `htmlText` property and the `pasteHTML()` method:

```
var range = frames["richedit"].document.selection.createRange();
range.pasteHTML("<span style=\"background-color:yellow\>" + range.htmlText +
"</span>");
```

This code retrieves the HTML of the current selection using `htmlText` and then surrounds it with a `` and inserts it back into the selection using `pasteHTML()`.

Rich Text in Forms

Since rich text editing is implemented using an `iframe` or a `contenteditable` element instead of a form control, a rich text editor is technically not part of a form. That means the HTML will not be submitted to the server unless you extract the HTML manually and submit it yourself. This is typically done by having a hidden form field that is updated with the HTML from the `iframe` or the `contenteditable` element. Just before the form is submitted, the HTML is extracted from the `iframe` or element and inserted into the hidden field. For example, the following may be done in the form's `onsubmit` event handler when using an `iframe`:



```
EventUtil.addHandler(form, "submit", function(event) {
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    target.elements["comments"].value = frames["richedit"].document.body.innerHTML;
});
```

RichTextEditingExample01.htm

Here, the HTML is retrieved from the `iframe` using the `innerHTML` property of the document's body and inserted into a form field named "`comments`". Doing so ensures that the "`comments`" field is filled in just before the form is submitted. If you are submitting the form manually using the `submit()` method, take care to perform this operation beforehand. You can perform a similar operation with a `contenteditable` element:

```
EventUtil.addHandler(form, "submit", function(event) {
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    target.elements["comments"].value =
        document.getElementById("richedit").innerHTML;
});
```

SUMMARY

Even though HTML and web applications have changed dramatically since their inception, web forms have remained mostly unchanged. JavaScript can be used to augment existing form fields to provide new functionality and usability enhancements. To aid in this, forms and form fields have properties, methods, and events for JavaScript usage. Here are some of the concepts introduced in this chapter:

- It's possible to select all of the text in a text box or just part of the text using a variety of standard and nonstandard methods.
- All browsers have adopted Firefox's way of interacting with text selection, making it a true standard.
- Text boxes can be changed to allow or disallow certain characters by listening for keyboard events and inspecting the characters being inserted.

All browsers except Opera support events for the clipboard, including `copy`, `cut`, and `paste`. Clipboard event implementations across the other browsers vary in the following ways:

- Internet Explorer, Firefox, Chrome, and Safari allow access to clipboard data from JavaScript, whereas Opera doesn't allow such access.
- Even amongst Internet Explorer, Chrome, and Safari, there are differences in implementation.
- Firefox, Safari, and Chrome allow reading of clipboard data only during the `paste` event, whereas Internet Explorer has no such restrictions.
- Firefox, Safari, and Chrome limit the availability of clipboard information to clipboard-related events, whereas Internet Explorer allows access to the data at any time.

Hooking into clipboard events is useful for blocking paste events when the contents of a text box must be limited to certain characters.

Select boxes are also frequently controlled using JavaScript. Thanks to the DOM, manipulating select boxes is much easier than it was previously. Options can be added, removed, moved from one select box to another, or reordered using standard DOM techniques.

Rich text editing is handled by using an `iframe` containing a blank HTML document. By setting the document's `designMode` property to "on", you make the page editable and it acts like a word processor. You can also use an element set as `contenteditable`. By default, you can toggle font styles such as bold and italic and use clipboard actions. JavaScript can access some of this functionality by using the `execCommand()` method and can get information about the text selection by using the `queryCommandEnabled()`, `queryCommandState()`, and `queryCommandValue()` methods. Since building a rich text editor in this manner does not create a form field, it's necessary to copy the HTML from the `iframe` or `contenteditable` element into a form field if it is to be submitted to the server.

15

Graphics with Canvas

WHAT'S IN THIS CHAPTER?

- Understanding the `<canvas>` element
- Drawing simple 2D graphics
- 3D drawing with WebGL

Arguably, HTML5's most popular addition is the `<canvas>` element. This element designates an area of the page where graphics can be created, on the fly, using JavaScript. Originally proposed by Apple for use with its Dashboard widgets, `<canvas>` quickly was added into HTML5 and found a very fast adoption rate amongst browsers. Internet Explorer 9+, Firefox 1.5+, Safari 2+, Opera 9+, Chrome, Safari for iOS, and WebKit for Android all support `<canvas>` to some degree.

Similar to the other parts of the browser environment, `<canvas>` is made up of a few API sets and not all browsers support all API sets. There is a 2D context with basic drawing capabilities and a proposed 3D context called WebGL. The latest versions of the supporting browsers now support the 2D context and the text API; support for WebGL is slowly evolving, but since WebGL is still experimental, full support will likely take longer. Firefox 4+ and Chrome support early versions of the WebGL specification, though older operating systems such as Windows XP lack the necessary graphics drivers for enabling WebGL even when these browsers are present.

BASIC USAGE

The `<canvas>` element requires at least its `width` and `height` attributes to be set in order to indicate the size of the drawing to be created. Any content appearing between the opening and closing tags is fallback data that is displayed only if the `<canvas>` element isn't supported. For example:

```
<canvas id="drawing" width="200" height="200">A drawing of something.</canvas>
```

As with other elements, the `width` and `height` attributes are also available as properties on the DOM element object and may be changed at any time. The entire element may be styled using CSS as well, and the element is invisible until it is styled or drawn upon.

To begin drawing on a canvas, you need to retrieve a drawing context. A reference to a drawing context is retrieved using the `getContext()` method and passing in the name of the context. For example, passing "2d" retrieves a 2D context object:

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

//make sure <canvas> is completely supported
if (drawing.getContext){

    var context = drawing.getContext("2d");

    //more code here
}
```

When using the `<canvas>` element, it's important to test for the presence of the `getContext()` method. Some browsers create default HTML element objects for elements that aren't officially part of HTML. In that case, the `getContext()` method would not be available even though `drawing` would contain a valid element reference.

Images created on a `<canvas>` element can be exported using the `toDataURL()` method. This method accepts a single argument, the MIME type format of the image to produce, and is applicable regardless of the context used to create the image. For example, to return a PNG-formatted image from a canvas, use the following:



Available for
download on
Wrox.com

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

//make sure <canvas> is completely supported
if (drawing.getContext){

    //get data URI of the image
    var imgURI = drawing.toDataURL("image/png");

    //display the image
    var image = document.createElement("img");
    image.src = imgURI;
    document.body.appendChild(image);
}
```

[2DDataUrlExample01.htm](#)

By default, browsers encode the image as PNG unless otherwise specified. Firefox and Opera also support JPEG encoding via "image/jpeg". Because this method was added later in the process, it was adopted in later versions of supporting browsers, including Internet Explorer 9, Firefox 3.5, and Opera 10.



The toDataURL() method throws an error if an image from a different domain is drawn onto a canvas. More details are available later in this chapter.

THE 2D CONTEXT

The 2D drawing context provides methods for drawing simple 2D shapes such as rectangles, arcs, and paths. The coordinates in a 2D context begin at the upper-left of the <canvas> element, which is considered point (0,0). All coordinate values are calculated in relation to that point, with x increasing to the right and y increasing toward the bottom. By default, the width and height indicate how many pixels are available in each direction.

Fills and Strokes

There are two basic drawing operations on the 2D context: fill and stroke. Fill automatically fills in the shape with a specific style (color, gradient, or image) while stroke colors only the edges. Most of the 2D context operations have both fill and stroke variants, and how they are displayed is based on a couple of properties: `fillStyle` and `strokeStyle`.

Both properties can be set to a string, a gradient object, or a pattern object, and both default to a value of “#000000”. A string value indicates a color defined using one of the various CSS color formats: name, hex code, `rgb`, `rgba`, `hsl`, or `hsla`. For example:

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

//make sure <canvas> is completely supported
if (drawing.getContext){

    var context = drawing.getContext("2d");
    context.strokeStyle = "red";
    context.fillStyle = "#0000ff";
}
```

This code sets the `strokeStyle` to “red” (a named CSS color) and `fillStyle` to “#0000ff” (also known as blue). All drawing operations involving stroke and fill will use these styles until the properties are changed again. These properties can also be set to a gradient or a pattern, both of which are discussed later in this chapter.

Drawing Rectangles

The only shape that can be drawn directly on the 2D drawing context is the rectangle. There are three methods for working with rectangles: `fillRect()`, `strokeRect()`, and `clearRect()`. Each of these methods accepts four arguments: the x-coordinate of the rectangle, the y-coordinate of the rectangle, the width of the rectangle, and the height of the rectangle. Each of these arguments is considered to be in pixels.

The `fillRect()` method is used to draw a rectangle that is filled with a specific color onto the canvas. The fill color is specified using the `fillStyle` property, for example:



```

var drawing = document.getElementById("drawing");

//make sure <canvas> is completely supported
if (drawing.getContext){

    var context = drawing.getContext("2d");

    /*
     * Based on Mozilla's documentation:
     * http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_usage
     */

    //draw a red rectangle
    context.fillStyle = "#ff0000";
    context.fillRect(10, 10, 50, 50);

    //draw a blue rectangle that's semi-transparent
    context.fillStyle = "rgba(0,0,255,0.5)";
    context.fillRect(30, 30, 50, 50);
}

}

```

[2DFillRectExample01.htm](#)

This code first sets the `fillStyle` to red and draws a rectangle located at (10,10) that's 50 pixels tall and wide. Next, it sets the `fillStyle` to a semitransparent blue color using `rgba()` format and draws another rectangle that overlaps the first. The result is that you can see the red rectangle through the blue rectangle (see Figure 15-1).

The `strokeRect()` method draws a rectangle outline using the color specified with the `strokeStyle` property. Here is an example:

```

var drawing = document.getElementById("drawing");

//make sure <canvas> is completely supported
if (drawing.getContext){

    var context = drawing.getContext("2d");

    /*
     * Based on Mozilla's documentation:
     * http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_usage
     */

    //draw a red outlined rectangle
    context.strokeStyle = "#ff0000";
    context.strokeRect(10, 10, 50, 50);

    //draw a blue outlined rectangle that's semi-transparent

```

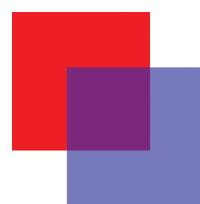


FIGURE 15-1

```

        context.strokeStyle = "rgba(0,0,255,0.5)";
        context.strokeRect(30, 30, 50, 50);
    }
}

```

[2DStrokeRectExample01.htm](#)

This code also draws two rectangles that overlap; however, they are just outlines rather than filled rectangles (see Figure 15-2).

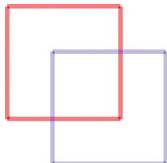


FIGURE 15-2



The size of the stroke is controlled by the lineWidth property, which can be set to any whole number. Likewise, a lineCap property describes the shape that should be used at the end of lines ("butt", "round", or "square") and lineJoin indicates how lines should be joined ("round", "bevel", or "miter").

You can erase an area of the canvas by using the `clearRect()` method. This method is used to make an area of the drawing context transparent. By drawing shapes and then clearing specific areas, you are able to create interesting effects, such as cutting out a section of another shape. Here is an example:



Available for
download on
[Wrox.com](#)

```

var drawing = document.getElementById("drawing");

//make sure <canvas> is completely supported
if (drawing.getContext){

    var context = drawing.getContext("2d");

    /*
     * Based on Mozilla's documentation:
     * http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_usage
     */

    //draw a red rectangle
    context.fillStyle = "#ff0000";
    context.fillRect(10, 10, 50, 50);

    //draw a blue rectangle that's semi-transparent
    context.fillStyle = "rgba(0,0,255,0.5)";
    context.fillRect(30, 30, 50, 50);

    //clear a rectangle that overlaps both of the previous rectangles
    context.clearRect(40, 40, 10, 10);
}

```

[2DClearRectExample01.htm](#)

Here, two filled rectangles overlap one another and then a small rectangle is cleared inside of that overlapping area. Figure 15-3 shows the result.

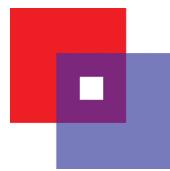


FIGURE 15-3

Drawing Paths

The 2D drawing context supports a number of methods for drawing paths on a canvas. Paths allow you to create complex shapes and lines. To start creating a path, you must first call `beginPath()` to indicate that a new path has begun. After that, the following methods can be called to create the path:

- `arc(x, y, radius, startAngle, endAngle, counterclockwise)`—Draws an arc centered at point `(x,y)` with a given radius and between `startAngle` and `endAngle` (expressed in radians). The last argument is a Boolean indicating if the `startAngle` and `endAngle` should be calculated counterclockwise instead of clockwise.
- `arcTo(x1, y1, x2, y2, radius)`—Draws an arc from the last point to `(x2,y2)`, passing through `(x1,y1)` with the given `radius`.
- `bezierCurveTo(c1x, c1y, c2x, c2y, x, y)`—Draws a curve from the last point to the point `(x,y)` using the control points `(c1x,c1y)` and `(c2x,c2y)`.
- `lineTo(x, y)`—Draws a line from the last point to the point `(x,y)`.
- `moveTo(x, y)`—Moves the drawing cursor to the point `(x,y)` without drawing a line.
- `quadraticCurveTo(cx, cy, x, y)`—Draws a quadratic curve from the last point to the point `(x,y)` using a control point of `(cx,cy)`.
- `rect(x, y, width, height)`—Draws a rectangle at point `(x,y)` with the given width and height. This is different from `strokeRect()` and `fillRect()` in that it creates a path rather than a separate shape.

Once the path has been created, you have several options. To draw a line back to the origin of the path, you can call `closePath()`. If the path is already completed and you want to fill it with `fillStyle`, call the `fill()` method. Another option is to stroke the path by calling the `stroke()` method, which uses `strokeStyle`. The last option is to call `clip()`, which creates a new clipping region based on the path.

As an example, consider the following code for drawing the face of a clock without the numbers:



Available for download on
Wrox.com

```
var drawing = document.getElementById("drawing");
//make sure <canvas> is completely supported
if (drawing.getContext){

    var context = drawing.getContext("2d");

    //start the path
    context.beginPath();

    //draw outer circle
    context.arc(250, 250, 240, 0, 2 * Math.PI);
    context.fill();
}
```

```

        context.arc(100, 100, 99, 0, 2 * Math.PI, false);

        //draw inner circle
        context.moveTo(194, 100);
        context.arc(100, 100, 94, 0, 2 * Math.PI, false);

        //draw minute hand
        context.moveTo(100, 100);
        context.lineTo(100, 15);

        //draw hour hand
        context.moveTo(100, 100);
        context.lineTo(35, 100);

        //stroke the path
        context.stroke();
    }
}

```

[2DPathExample01.htm](#)

This example draws two circles using `arc()`: an outer one and an inner one to create a border around the clock. The outer circle has a radius of 99 pixels and is centered at (100,100), which is the center of the canvas. To draw a complete circle, you must start at an angle of 0 radians and draw all the way around to 2π radians (calculated using `Math.PI`). Before drawing the inner circle, you must move the path to a point that will be on the circle to avoid an additional line being drawn. The second call to `arc()` uses a slightly smaller radius for the border effect. After that, combinations of `moveTo()` and `lineTo()` are used to draw the hour and minute hands. The last step is to call `stroke()`, which makes the image appear as shown in Figure 15-4.

Paths are the primary drawing mechanism for the 2D drawing context because they provide more control over what is drawn. Since paths are used so often, there is also a method called `isPointInPath()`, which accepts an x-coordinate and a y-coordinate as arguments. This method can be called anytime before the path is closed to determine if a point exists on the path, as shown here:

```

if (context.isPointInPath(100, 100)){
    alert("Point (100, 100) is in the path.");
}

```

The path API for the 2D drawing context is robust enough to create very complex images using multiple fill styles, stroke styles, and more.

Drawing Text

Since it's often necessary to mix text and graphics, the 2D drawing context provides methods to draw text. There are two methods for drawing text, `fillText()` and `strokeText()`, and each takes

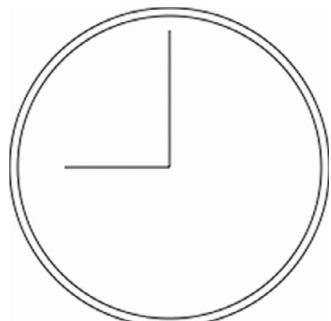


FIGURE 15-4

four arguments: the string to draw, the x-coordinate, the y-coordinate, and an optional maximum pixel width to draw. Both methods base their drawing on the following three properties:

- **font** — Indicates the font style, size, and family in the same manner specified in CSS, such as "10px Arial".
- **textAlign** — Indicates how the text should be aligned. Possible values are "start", "end", "left", "right", and "center". It's recommended to use "start" and "end" instead of "left" and "right" as these are more indicative of rendering in both left-to-right languages and right-to-left languages.
- **textBaseline** — Indicates the baseline of the text. Possible values are "top", "hanging", "middle", "alphabetic", "ideographic", and "bottom".

These properties have a default value, so there's no need to set them each time you want to draw text. The `fillText()` method uses the `fillStyle` property to draw the text, whereas the `strokeText()` method uses the `strokeStyle` property. You will probably use `fillText()` most of the time, since this mimics normal text rendering on web pages. For example, the following renders a 12 at the top of the clock created in the previous section:



```
context.font = "bold 14px Arial";
context.textAlign = "center";
context.textBaseline = "middle";
context.fillText("12", 100, 20);
```

2DTextExample01.htm

The resulting image is displayed in Figure 15-5.

Since `textAlign` is set to "center" and `textBaseline` is set to "middle", the coordinates (100,80) indicate the horizontal and vertical center and top coordinates for the text. If `textAlign` were "start", then the x-coordinate would represent the left coordinate of the text in a left-to-right language while "end" would make the x-coordinate represent the right coordinate in a left-to-right language. For example:

```
//normal
context.font = "bold 14px Arial";
context.textAlign = "center";
context.textBaseline = "middle";
context.fillText("12", 100, 20);

//start-aligned
context.textAlign = "start";
context.fillText("12", 100, 40);

//end-aligned
context.textAlign = "end";
context.fillText("12", 100, 60);
```

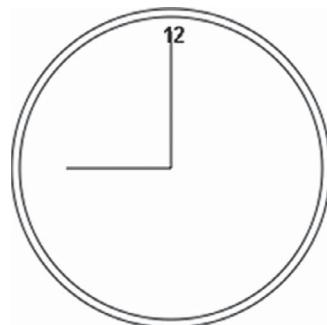


FIGURE 15-5

2DTextExample02.htm

The string "12" is drawn three times, each using the same x-coordinate but with three different `textAlign` values. The y-coordinate values are also incremented so that the strings don't render on top of one another. The resulting image is shown in Figure 15-6.

The vertical line of the clock is directly at the center so the alignment of the text becomes obvious. You can similarly adjust how the text is aligned vertically by altering `textBaseline`. Setting to "top" means that the y-coordinate is the top of the text, "bottom" means it's the bottom, and "hanging", "alphabetic", and "ideographic" refer to specific baseline coordinates of a font.

Since drawing text is quite complicated, especially when you want text to render within a specific area, the 2D context provides a little extra help to determine the dimensions of text via the `measureText()` method. This method accepts a single argument, the text to draw, and returns a `TextMetrics` object. The returned object currently has only one property, `width`, but the intent is to provide more metrics in the future.

The `measureText()` method uses the current values for `font`, `textAlign`, and `textBaseline` to calculate the size of the specified text. For example, suppose you want to fit the text "Hello world!" within a rectangle that is 140 pixels wide. The following code starts with a font size of 100 pixels and decrements until the text fits:



Available for download on Wrox.com

```
var fontSize = 100;
context.font = fontSize + "px Arial";

while(context.measureText("Hello world!").width > 140) {
    fontSize--;
    context.font = fontSize + "px Arial";
}

context.fillText("Hello world!", 10, 10);
context.fillText("Font size is " + fontSize, 10, 50);
```

[2DTextExample03.htm](#)

There is also a fourth argument for both `fillText()` and `strokeText()`, which is the maximum width of the text. This argument is optional and hasn't been implemented in all browsers yet (Firefox 4 was the first to implement it). When provided, calling `fillText()` or `strokeText()` with a string that will not fit within the maximum width results in the text being drawn with the correct character height, but the characters are scaled horizontally to fit. Figure 15-7 shows this effect.



FIGURE 15-7

Text drawing is one of the more complex drawing operations and, as such, not all portions of the API have been implemented in all browsers that support the `<canvas>` element.

Transformations

Context transformations allow the manipulation of images drawn onto the canvas. The 2D drawing context supports all of the basic drawing transformations. When the drawing context is created, the

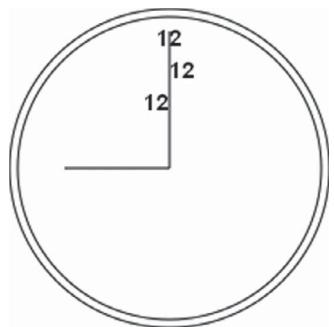


FIGURE 15-6

transformation matrix is initialized with default values that cause all drawing operations to be applied directly as they are described. Applying transformations to the drawing context causes operations to be applied using a different transformation matrix and thus produces a different result.

The transformation matrix can be augmented by using any of the following methods:

- `rotate(angle)`— Rotates the image around the origin by `angle` radians.
- `scale(scaleX, scaleY)`— Scales the image by a multiple of `scaleX` in the x dimension and by `scaleY` in the y dimension. The default value for both `scaleX` and `scaleY` is 1.0.
- `translate(x, y)`— Moves the origin to the point `(x, y)`. After performing this operation, the coordinates (0,0) are located at the point previously described as `(x, y)`.
- `transform(m1_1, m1_2, m2_1, m2_2, dx, dy)`— Changes the transformation matrix directly by multiplying by the matrix described as this:

```
m1_1 m1_2 dx
m2_1 m2_2 dy
0      0      1
```

- `setTransform(m1_1, m1_2, m2_1, m2_2, dx, dy)`— Resets the transformation matrix to its default state and then calls `transform()`.

Transformations can be as simple or as complex as necessary. For example, it may be easier to draw the hands on the clock in the previous example by translating the origin to the center of the clock and then drawing the hands from there. Consider the following:



Available for
download on
Wrox.com

```
var drawing = document.getElementById("drawing");
//make sure <canvas> is completely supported
if (drawing.getContext){

    var context = drawing.getContext("2d");

    //start the path
    context.beginPath();

    //draw outer circle
    context.arc(100, 100, 99, 0, 2 * Math.PI, false);

    //draw inner circle
    context.moveTo(194, 100);
    context.arc(100, 100, 94, 0, 2 * Math.PI, false);

    //translate to center
    context.translate(100, 100);

    //draw minute hand
    context.moveTo(0,0);
    context.lineTo(0, -85);

    //draw hour hand
```

```

    context.moveTo(0, 0);
    context.lineTo(-65, 0);

    //stroke the path
    context.stroke();
}

```

[2DTransformExample01.htm](#)

After translating the origin to (100,100), the center of the clock face, it's just a matter of simple math to draw the lines in the same direction. All math is now based on (0,0) instead of (100,100). You can go further, moving the hands of the clock by using the `rotate()` method as shown here:



Available for
download on
Wrox.com

```

var drawing = document.getElementById("drawing");

//make sure <canvas> is completely supported
if (drawing.getContext){

    var context = drawing.getContext("2d");

    //start the path
    context.beginPath();

    //draw outer circle
    context.arc(100, 100, 99, 0, 2 * Math.PI, false);

    //draw inner circle
    context.moveTo(194, 100);
    context.arc(100, 100, 94, 0, 2 * Math.PI, false);

    //translate to center
    context.translate(100, 100);

    //rotate the hands
    context.rotate(1);

    //draw minute hand
    context.moveTo(0,0);
    context.lineTo(0, -85);

    //draw hour hand
    context.moveTo(0, 0);
    context.lineTo(-65, 0);

    //stroke the path
    context.stroke();
}

```

[2DTransformExample01.htm](#)

Since the origin has already been translated to the center of clock, the rotation is applied from that point. This means that the hands are anchored at the center and then rotated around to the right. The result is displayed in Figure 15-8.

All of these transformations, as well as properties like `fillStyle` and `strokeStyle`, remain set on the context until explicitly changed. Although there's no way to explicitly reset everything to their default values, there are two methods that can help keep track of changes. Whenever you want to be able to return to a specific set of properties and transformations, call the `save()` method. Once called, this method pushes all of the settings at the moment onto a stack for safekeeping. You can then go on to make other changes to the context. When you want to go back to the previous settings, call the `restore()` method, which pops the settings stack and restores all of the settings. You can keep calling `save()` to store more settings on the stack and then systematically go back through them using `restore()`. Here is an example:



Available for
download on
Wrox.com

```
context.fillStyle = "#ff0000";
context.save();

context.fillStyle = "#00ff00";
context.translate(100, 100);
context.save();

context.fillStyle = "#0000ff";
context.fillRect(0, 0, 100, 200);    //draws blue rectangle at (100, 100)

context.restore();
context.fillRect(10, 10, 100, 200);    //draws green rectangle at (110, 110)

context.restore();
context.fillRect(0, 0, 100, 200);    //draws red rectangle at (0,0)
```

[2DSaveRestoreExample01.htm](#)

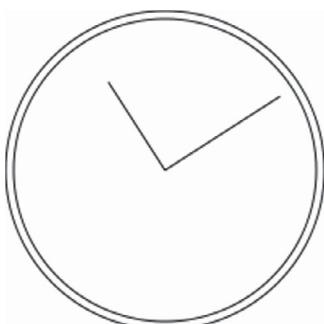


FIGURE 15-8

In this code, the `fillStyle` is set to red and then `save()` is called. Next, the `fillStyle` is changed to green, and the coordinates are translated to (100,100). Once again, `save()` is called to save these settings. The `fillStyle` property is then set to blue and a rectangle is drawn. Because the coordinates are translated, the rectangle actually ends up being drawn at (100,100). When `restore()` is called, `fillStyle` is set back to green, so the next rectangle that's drawn is green. This rectangle is drawn at (110,110) because the translation is still in effect. When `restore()` is called one more time, the translation is removed and `fillStyle` is set back to red. The last rectangle is drawn at (0,0).

Note that `save()` saves only the settings and transformations applied to the drawing context but not the contents of the drawing context.

Drawing Images

The 2D drawing context has built-in support for working with images. If you have an existing image that should be drawn on the canvas, you can do so using the `drawImage()` method. This method can be called with three different sets of arguments based on the desired result. The simplest call is to pass in an HTML `` element, as well as the destination x- and y-coordinates, which simply draws the image at the specified location. Here is an example:



Available for
download on
Wrox.com

```
var image = document.images[0];
context.drawImage(image, 10, 10);
```

[2DDrawImageExample01.htm](#)

This code gets the first image in the document and draws it on the context at position (10,10). The image is drawn in the same scale as the original. You can change how the image is drawn by adding two more arguments: the destination width and destination height. This scales the drawing without affecting the transformation matrix of the context. Here's an example:

```
context.drawImage(image, 50, 10, 20, 30);
```

[2DDrawImageExample01.htm](#)

When this code is executed, the image is scaled to be 20 pixels wide by 30 pixels tall.

You can also select just a region of the image to be drawn onto the context. This is done by providing nine arguments to `drawImage()`: the image to draw, the source x-coordinate, the source y-coordinate, the source width, the source height, the destination x-coordinate, the destination y-coordinate, the destination width, and the destination height. Using this overload of `drawImage()` gives you the most control. Consider this example:

```
context.drawImage(image, 0, 10, 50, 50, 0, 100, 40, 60);
```

[2DDrawImageExample01.htm](#)

Here, only part of the image is drawn on the canvas. That part of the image begins at point (0,10) and is 50 pixels wide and 50 pixels tall. The image is drawn to point (0,100) on the context and scaled to fit in a 40×60 area.

These drawing operations allow you to create interesting effects such as those shown in Figure 15-9.

In addition to passing in an HTML `` element as the first argument, you can also pass in another `<canvas>` element to draw the contents of one canvas onto another.

The `drawImage()` method, in combination with other methods, can easily be used to perform basic image manipulation, the result of which can be retrieved using `toDataURL()`. There is, however, one instance where this won't work: if an image from a different origin than the page is drawn onto



FIGURE 15-9

the context. In that case, calling `toDataURL()` throws an error. For example, if a page hosted on `www.example.com` draws an image hosted on `www.wrox.com`, the context is considered “dirty” and an error is thrown.

Shadows

The 2D context will automatically draw a shadow along with a shape or path based on the value of several properties:

- `shadowColor` — The CSS color in which the shadow should be drawn. The default is black.
- `shadowOffsetX` — The x-coordinate offset from the x-coordinate of the shape or path. The default is 0.
- `shadowOffsetY` — The y-coordinate offset from the y-coordinate of the shape or path. The default is 0.
- `shadowBlur` — The number of pixels to blur. If set to 0, the shadow has no blur. The default is 0.

Each of these properties can be read and written on the `context` object. You just need to set the values appropriately before drawing and the shadows are drawn automatically. For example:



Available for
download on
Wrox.com

```
var context = drawing.getContext("2d");

//setup shadow
context.shadowOffsetX = 5;
context.shadowOffsetY = 5;
context.shadowBlur    = 4;
context.shadowColor   = "rgba(0, 0, 0, 0.5)";

//draw a red rectangle
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

//draw a blue rectangle
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);
```

[2DFillRectShadowExample01.htm](#)

A shadow is drawn using the same styles for both rectangles, resulting in the image displayed in Figure 15-10.

There are some quirks with shadow support across browsers. Internet Explorer 9, Firefox 4, and Opera 11 have the correct behavior in all situations while the others have strange effects or none at all. Chrome (through version 10) will incorrectly apply a filled shadow to a stroked shape. Both Chrome and Safari (through version 5) have a problem drawing shadows for images with transparent pixels. While the shadow should be under the nontransparent parts of the image, it actually just disappears. Safari will also not apply a shadow to a gradient, while the other browsers will.

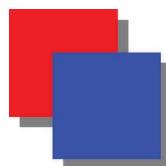


FIGURE 15-10

Gradients

Gradients are represented by an instance of `CanvasGradient` and are very simple to create and modify using the 2D context. To create a new linear gradient, call the `createLinearGradient()` method. This method accepts four arguments: the starting x-coordinate, the starting y-coordinate, the ending x-coordinate, and the ending y-coordinate. Once called, the method creates a new `CanvasGradient` object of the size you specified and returns the instance.

Once you have the `gradient` object, the next step is to assign color stops using the `addColorStop()` method. This method accepts two arguments: the location of the color stop and a CSS color. The color stop location is a number between 0 (the first color) and 1 (the last color). For example:



Available for
download on
[Wrox.com](#)

```
var gradient = context.createLinearGradient(30, 30, 70, 70);
gradient.addColorStop(0, "white");
gradient.addColorStop(1, "black");
```

[2DFillRectGradientExample01.htm](#)

The `gradient` object now represents a gradient that is drawn from point (30,30) to point (70,70) on the canvas. The starting color is white and the stopping color is black. You can now set the `fillStyle` or `strokeStyle` properties to this value to draw a shape using the gradient:

```
//draw a red rectangle
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

//draw a gradient rectangle
context.fillStyle = gradient;
context.fillRect(30, 30, 50, 50);
```

[2DFillRectGradientExample01.htm](#)

In order for the gradient to be drawn over the entire rectangle and not just part of it, the coordinates need to match up. This code produces the drawing in Figure 15-11.

If the rectangle isn't drawn in exactly this spot, then only part of the gradient is displayed. For example:

```
context.fillStyle = gradient;
context.fillRect(50, 50, 50, 50);
```

[2DFillRectGradientExample02.htm](#)



FIGURE 15-11

This code creates a rectangle with only a small amount of white in the upper-left corner. That's because the rectangle is drawn at the midpoint of the gradient, where the color transition is almost complete. The rectangle is therefore mostly black since gradients do not repeat. Keeping the gradient



```
function createRectLinearGradient(context, x, y, width, height){
    return context.createLinearGradient(x, y, x+width, y+height);
}
```

[2DFillRectGradientExample03.htm](#)

This function creates a gradient based on the starting x- and y-coordinates, along with a width and height, so that the same numbers can be used as `fillRect()`:

```
var gradient = createRectLinearGradient(context, 30, 30, 50, 50);

gradient.addColorStop(0, "white");
gradient.addColorStop(1, "black");

//draw a gradient rectangle
context.fillStyle = gradient;
context.fillRect(30, 30, 50, 50);
```

[2DFillRectGradientExample03.htm](#)

Keeping track of coordinates is an important and tricky aspect of using canvas. Helper functions such as `createRectLinearGradient()` can take some of the pain out of managing coordinates.

Radial gradients are created using the `createRadialGradient()` method. This method accepts six arguments corresponding to the center of a circle and its radius. The first three arguments define the starting circle's center (x and y) and radius, while the last three define the same for the ending circle. When thinking about radial gradients, you will find it helps to think of a long cylinder where you're defining the size of the circle on each end. By making one circle smaller and the other larger, you've effectively made a cone, and you rotate that cone around by moving the center of each circle.

To create a radial gradient that starts in the center of a shape and continues out, you need to set the center of both circles to the same origin. For example, to create a radial gradient in the center of the rectangle in the previous example, both circles must be centered at (55,55). That's because the rectangle is drawn from point (30,30) to point (80,80). Here's the code:

```
var gradient = context.createRadialGradient(55, 55, 10, 55, 55, 30);

gradient.addColorStop(0, "white");
gradient.addColorStop(1, "black");

//draw a red rectangle
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

//draw a gradient rectangle
context.fillStyle = gradient;
context.fillRect(30, 30, 50, 50);
```

[2DFillRectGradientExample04.htm](#)

Running this code results in the drawing displayed in Figure 15-12.

Radial gradients are a little bit more difficult to work with because of the complexities of their creation, but generally you'll end up using the same center for both starting circle and ending circle and just altering the radii of the circles for most basic effects.

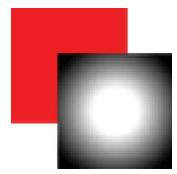


FIGURE 15-12

Patterns

Patterns are simply repeating images that may be used to fill or stroke a shape. To create a new pattern, call the `createPattern()` method and pass in two arguments: an HTML `` element and a string indicating how the image should be repeated. The second argument is the same as the values for the CSS `background-repeat` property: "repeat", "repeat-x", "repeat-y", and "no-repeat". For example:



```
var image = document.images[0],
    pattern = context.createPattern(image, "repeat");

//draw a rectangle
context.fillStyle = pattern;
context.fillRect(10, 10, 150, 150);
```

[2DFillRectPatternExample01.htm](#)

Keep in mind that, like gradients, a pattern actually starts at point (0,0) on the canvas. Setting the fill style to a pattern means revealing the pattern in the specified location rather than starting to draw at that position. This code results in a page that looks like Figure 15-13.

The first argument for `createPattern()` can also be a `<video>` element or another `<canvas>` element.

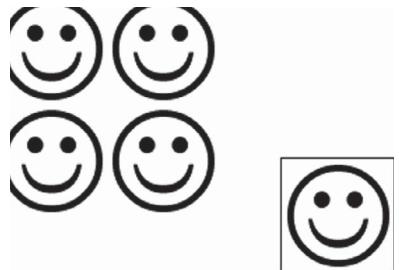


FIGURE 15-13

Working with Image Data

One of the more powerful aspects of the 2D context is the ability to retrieve raw image data using the `getImageData()` method. This method accepts four arguments: the left and top position of the first pixel whose data should be retrieved, and the pixel width and the pixel height to retrieve. For instance, to get image data for a 50 by 50 area starting at (10,5), use the following:

```
var imageData = context.getImageData(10, 5, 50, 50);
```

The returned object is an instance of `ImageData`. Each `ImageData` object contains just three properties: `width`, `height`, and `data`. The `data` property is an array that contains the raw pixel information for the image. Each pixel is actually represented as four items in the `data` array, one each for red, green, blue, and alpha. So the data for the first pixel is contained in items 0 through 3, such as:

```
var data = imageData.data,
    red = data[0],
```

```
green = data[1],
blue = data[2],
alpha = data[3];
```

Each value in the array is a number between 0 and 255, inclusive. Having access to the raw image data allows you to manipulate the image in a variety of ways. For example, a simple grayscale filter can be created by changing the image data:



Available for
download on
Wrox.com

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

//make sure <canvas> is completely supported
if (drawing.getContext){

    var context = drawing.getContext("2d"),
        image = document.images[0],
        imageData, data,
        i, len, average,
        red, green, blue, alpha;

    //draw regular size
    context.drawImage(image, 0, 0);

    //get the image data
    imageData = context.getImageData(0, 0, image.width, image.height);
    data = imageData.data;

    for (i=0, len=data.length; i < len; i+=4){

        red = data[i];
        green = data[i+1];
        blue = data[i+2];
        alpha = data[i+3];

        //get the average of rgb
        average = Math.floor((red + green + blue) / 3);

        //set the colors, leave alpha alone
        data[i] = average;
        data[i+1] = average;
        data[i+2] = average;

    }

    //assign back to image data and display
    imageData.data = data;
    context.putImageData(imageData, 0, 0);
}
```

[2DImageDataExample01.htm](#)

This example first draws an image onto the canvas and then retrieves its image data. A `for` loop iterates over each pixel in the image data. Note that each trip through the loop adds 4 to the value of `i`. Once the red, green, and blue values are retrieved, they are averaged together to get a new value. Then each

of the values is set back to that average, effectively washing out the color and leaving only a gray of similar brightness in its place. The `data` array is then assigned back onto the `imageData` object. After that, the `putImageData()` method is called to draw the image data back to the canvas. The result is a grayscale version of the image.

Of course, grayscale isn't the only type of filter that can be implemented by manipulating raw pixel values. For more information on creating filters with raw image data, see "Making Image Filters with Canvas" by Ilmari Heikkinen (www.html5rocks.com/en/tutorials/canvas/imagefilters/).



Image data is available only if the canvas isn't dirty from loading a cross-domain resource. Attempting to access image data when the canvas is dirty causes a JavaScript error.

Compositing

There are two properties that apply to all drawing done on the 2D context: `globalAlpha` and `globalCompositionOperation`. The `globalAlpha` property is a number between 0 and 1, inclusive, that specifies the alpha value for all drawings. The default value is 0. If all of the upcoming drawings should be done with the same alpha, set `globalAlpha` to the appropriate value, perform the drawings, and then set `globalAlpha` back to 0. For example:



Available for download on
Wrox.com

```
//draw a red rectangle
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

//change the global alpha
context.globalAlpha = 0.5;

//draw a blue rectangle
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);

//reset
context.globalAlpha = 0;
```

[2DGlobalAlphaExample01.htm](#)

In this example, a blue rectangle is drawn on top of a red rectangle. Since `globalAlpha` is set to 0.5 before drawing the blue rectangle, it becomes partially transparent, allowing the red rectangle to be seen through the blue.

The `globalCompositionOperation` property indicates how newly drawn shapes should merge with the already-existing image on the context. This property is a string value of one of the following:

- `source-over` (default) — New drawing is drawn on top of the existing image.
- `source-in` — New drawing is drawn only where it overlaps the existing image. Everything else becomes transparent.

- `source-out` — New drawing is drawn only where it does not overlap the existing image. Everything else becomes transparent.
- `source-atop` — New drawing is drawn only where it overlaps the existing image. The existing image is otherwise unaffected.
- `destination-over` — New drawing is drawn underneath the existing image, visible only through previously transparent pixels.
- `destination-in` — New drawing is drawn underneath the existing image, and all places where the two images do not overlap become transparent.
- `destination-out` — New drawing erases the parts of the existing image where they overlap.
- `destination-atop` — New drawing is drawn behind the existing image. The existing image becomes transparent where there is no overlap with new drawing.
- `lighter` — New drawing is drawn by combining its values with the existing image values to create a lighter image.
- `copy` — New drawing erases the existing image and replaces it completely.
- `xor` — New drawing is drawn by XORing the image data with the existing image.

The descriptions of these composite operations are difficult to represent in words or black-and-white images. For a better demonstration of each operation, see https://developer.mozilla.org/samples/canvas-tutorial/6_1_canvas_composite.html. It's recommended to visit this site in Internet Explorer 9+ or Firefox 4+, as they have the most complete implementations of canvas. Here's a simple example:



```
//draw a red rectangle
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

//set composite operation
context.globalCompositeOperation = "destination-over";

//draw a blue rectangle
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);
```

[2DGlobalCompositeOperationExample01.htm](#)

Even though the blue rectangle would normally be drawn over the red, changing `globalCompositeOperation` to "destination-over" means that the red rectangle actually ends up on top of the blue.

When using `globalCompositeOperation`, be sure to test across a wide variety of browsers. There are still significant differences between how these operations are implemented cross-browser. Safari and Chrome still have several issues with these operations, which can be seen by going to the previously mentioned URL and comparing the rendering to that of Internet Explorer or Firefox.

WEBGL

WebGL is a 3D context for canvas. Unlike other web technologies, WebGL is not specified by the W3C. Instead, the Khronos Group is developing the specification. According to its website, “The Khronos Group is a not for profit, member-funded consortium focused on the creation of royalty-free open standards for parallel computing, graphics and dynamic media on a wide variety of platforms and devices.” The Khronos Group has also worked on other graphics APIs, such as OpenGL ES 2.0, which is the basis for WebGL in the browser.

3D graphics languages such as OpenGL are complex topics, and it is beyond the scope of this book to cover all concepts. Familiarity with OpenGL ES 2.0 is recommended for using WebGL as a lot of concepts map directly.

This section assumes a working knowledge of OpenGL ES 2.0 concepts and simply attempts to describe how certain parts of OpenGL ES 2.0 have been implemented in WebGL. For more information on OpenGL, please visit www.opengl.org and for an excellent series of WebGL tutorials, please visit www.learningwebgl.com.

Typed Arrays

Since WebGL deals with complex calculations requiring predictable precision, standard JavaScript numbers do not work. Instead, WebGL introduces the concept of *typed arrays*, which are arrays whose items are set to be values of a particular type.

At the core of typed arrays is a type called `ArrayBuffer`. An `ArrayBuffer` object represents a specified number of bytes in memory but does not specify the type to treat the bytes. All you can do with an `ArrayBuffer` is allocate a certain number of bytes for use. For example, the following allocates 20 bytes:

```
var buffer = new ArrayBuffer(20);
```

Once the `ArrayBuffer` is created, all you can do with the object itself is retrieve the number of bytes contained within by accessing the `byteLength` property:

```
var bytes = buffer.byteLength;
```

Although the `ArrayBuffer` object itself isn’t very interesting, its use is extremely important to WebGL and is made more interesting when you use views.

Views

An array buffer *view* is a particular way of using the bytes within an array buffer. The most generic view is `DataView`, which allows you to select a subset of bytes in an `ArrayBuffer`. To do so, create a new instance of `DataView` and pass in the `ArrayBuffer`, an optional byte offset from which to select, and an optional number of bytes to select. For example:

```
//create a new view over the entire buffer
var view = new DataView(buffer);

//create a new view starting with byte 9
```

```

var view = new DataView(buffer, 9);

//create a new view going from byte 9 to byte 18
var view = new DataView(buffer, 9, 10);

```

Once instantiated, a `DataView` keeps the byte offset and length information in the `byteOffset` and `byteLength` properties, respectively:

```

alert(view.byteOffset);
alert(view.byteLength);

```

These properties let you easily inspect the view later on. You can also retrieve the array buffer through the `buffer` property.

Reading and writing to the `DataView` are done through a series of getter and setter methods based on the type of data you're working with. The following table lists the supported data types and their associated methods:

DATA TYPE	GETTER	SETTER
Signed 8-bit integer	<code>getInt8(byteOffset)</code>	<code>setInt8(byteOffset, value)</code>
Unsigned 8-bit integer	<code>getUint8(byteOffset)</code>	<code>setUint8(byteOffset, value)</code>
Signed 16-bit integer	<code>getInt16(byteOffset, littleEndian)</code>	<code>setInt16(byteOffset, value, littleEndian)</code>
Unsigned 16-bit integer	<code>getUint16(byteOffset, littleEndian)</code>	<code>setUint16(byteOffset, value, littleEndian)</code>
Signed 32-bit integer	<code>getInt32(byteOffset, littleEndian)</code>	<code>setInt32(byteOffset, value, littleEndian)</code>
Unsigned 32-bit integer	<code>getUint32(byteOffset, littleEndian)</code>	<code>setUint32(byteOffset, value, littleEndian)</code>
32-bit float	<code>getFloat32(byteOffset, littleEndian)</code>	<code>setFloat32(byteOffset, value, littleEndian)</code>
64-bit float	<code>getFloat64(byteOffset, littleEndian)</code>	<code>setFloat64(byteOffset, value, littleEndian)</code>

Each method expects the first argument to be the byte offset to retrieve from or write to. Keep in mind that, depending on the data type, the data may take more than one byte to store. An unsigned 8-bit integer takes one byte to store while a 32-bit float takes four bytes. Using a `DataView`, you'll need to manage this yourself by ensuring you know exactly how many bytes your data needs and using the correct method. For example:

```

var buffer = new ArrayBuffer(20),
    view = new DataView(buffer),
    value;

view.setUint16(0, 25);

```



```
view.setUint16(2, 50); //don't start at 1, 16-bit integers take two bytes
value = view.getUint16(0);
```

[DataViewExample01.htm](#)

This code saves two unsigned 16-bit integers into the array buffer. Since each 16-bit integer takes two bytes, the first number is stored at byte offset 0 while the second is stored at byte offset 2.

Each method dealing with 16-bit or larger numbers has an optional argument called `littleEndian`. This is a Boolean value indicating if the value should be read or written as little-endian (least significant byte is the first byte) instead of big-endian (least significant byte is the last byte). If you're not sure which to use, then leave off this option to use the default big-endian storage pattern.

Because you're dealing with byte offsets rather than item numbers, it's possible to access the same bytes in different ways. For example:



```
var buffer = new ArrayBuffer(20),
    view = new DataView(buffer),
    value;

view.setUint16(0, 25);
value = view.getInt8(0);

alert(value); //0
```

[DataViewExample02.htm](#)

In this example, the number 25 is written into a 16-bit unsigned integer beginning at byte offset 0. An attempt to read the value as an 8-bit signed integer results in a return value of 0. That's because the binary form of 25 has all zeros in the first byte (see Figure 15-14).

So while the `DataView` gives you access to byte-level data in an array buffer, you'll need to keep track of where data is being stored and how many bytes it needs to do so. This can be a lot of work, and so typed views are also available.

Typed Views

The typed views are typically referred to as *typed arrays* because they act like regular arrays with the exception that their elements must be of a particular data type. There are several typed views, and all of them inherit from `DataView`:

- `Int8Array` — Represents numbers as 8-bit two's complement integers.
- `Uint8Array` — Represents numbers as 8-bit unsigned integers.
- `Int16Array` — Represents numbers as 16-bit two's complement integers.

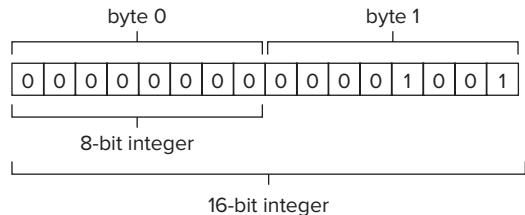


FIGURE 15-14

- `Uint16Array` — Represents numbers as 16-bit unsigned integers.
- `Int32Array` — Represents numbers as 32-bit two's complement integers.
- `Uint32Array` — Represents numbers as 32-bit unsigned integers.
- `Float32Array` — Represents numbers as 32-bit IEEE floating-point values.
- `Float64Array` — Represents numbers as 64-bit IEEE floating-point values.

Each view represents data in a different way, and each piece of data may take one or more bytes to represent. For example, an `ArrayBuffer` of 20 bytes will hold 20 values in `Int8Array` or `Uint8Array`; 10 values in `Int16Array` or `Uint16Array`; 5 values in `Int32Array`, `Uint32Array`, or `Float32Array`; or two values in `Float64Array`.

Since each of these types inherits from `DataView`, you can instantiate them using the same constructor arguments: the `ArrayBuffer` object to use, the starting byte offset (0 by default), and the number of bytes to include. Only the first argument is required. Some examples:

```
//create a new array that uses the whole buffer
var int8s = new Int8Array(buffer);

//only use bytes starting at byte 9
var int16s = new Int16Array(buffer, 9);

//only use bytes starting at 9 going through 18
var uint16s = new Uint16Array(buffer, 9, 10);
```

Being able to specify a subset of a buffer means that you can store different numeric types within the same buffer. For example, the following allows storing of 8-bit integers at the start of the buffer and 16-bit integers in the rest:

```
//use part of the buffer for 8-bit integers, part for 16-bit
var int8s = new Int8Array(buffer, 0, 10);
var uint16s = new Uint16Array(buffer, 11, 10);
```

Each view constructor has a property called `BYTES_PER_ELEMENT` that indicates how many bytes each element of a typed array requires. So `Uint8Array.BYTES_PER_ELEMENT` is 1 while `Float32Array.BYTES_PER_ELEMENT` is 4. You can use this to help initialize a view:

```
//need space for 10 items
var int8s = new Int8Array(buffer, 0, 10 * Int8Array.BYTES_PER_ELEMENT);

//need space for 5 items
var uint16s = new Uint16Array(buffer, int8s.byteOffset + int8s.byteLength,
    5 * Uint16Array.BYTES_PER_ELEMENT);
```

This example creates two views onto an array buffer. The first 10 bytes are used for storing 8-bit integers while the rest are used to store unsigned 16-bit integers. The `Uint16Array` uses the `byteOffset` and `byteLength` properties of the `Int8Array` to ensure the view starts after the 8-bit data.

Since the point of typed views are to make working with binary data easier, you can also create a new typed view without first creating an `ArrayBuffer` object. Just pass in the number of items you'd like the array to hold, and an `ArrayBuffer` will be automatically created with the correct number of bytes. For example:

```
//create an array for 10 8-bit integers (10 bytes)
var int8s = new Int8Array(10);

//create an array for 10 16-bit integers (20 bytes)
var int16s = new Int16Array(10);
```

Regular arrays can also be converted into typed views by passing them into a typed view constructor:

```
//create an array for 5 8-bit integers (10 bytes)
var int8s = new Int8Array([10, 20, 30, 40, 50]);
```

This is the best way to initialize typed views with default values and is used quite frequently with WebGL projects.

Using typed views in this way makes them more like regular `Array` objects and ensures that the proper data types are used when reading or writing information.

When using a typed view, you can access data members using bracket notation and use the `length` property to determine how many items are present. This makes iterating over typed views exactly the same as iterating over `Array` objects:

```
for (var i=0, len=int8s.length; i < len; i++){
    console.log("Value at position " + i + " is " + int8s[i]);
}
```

Values can also be assigned to spots in a typed view using bracket notation. If the value doesn't fit within the specified number of bytes for an item, the number is stored as the modulo of the largest possible number. For example, the largest number that can be represented as an unsigned 16-bit integer is 65535. If you attempt to store 65536, it becomes 0; attempting to store 65537 yields 1, and so on:

```
var uint16s = new Uint16Array(10);
uint16s[0] = 65537;
alert(uint16s[0]); //1
```

No errors are thrown when data types don't match, so you must be certain that numbers fit within their byte limits.

Typed views have one additional method called `subarray()`, which allows you to create a new view on a subset of the underlying array buffer. This method accepts two arguments, the item index to start with and an optional item index to end with. The returned type is the same as the type on which the method was called. For example:

```
var uint16s = new Uint16Array(10),
    sub = uint16s.subarray(2, 5);
```

In this code, `sub` is also an instance of `Uint16Array` and is a view into the same `ArrayBuffer` object as `uint16s`. The advantage of subarrays is in allowing access to a smaller number of items in a larger array without fear of unintentionally modifying other items.

Typed arrays are an important part of performing operations in WebGL.

The WebGL Context

The WebGL context name in supporting browsers is currently "experimental-webgl", as the WebGL specification is still under development. Once development is complete, the context name will simply be "webgl". If the browser doesn't support WebGL, then attempting to retrieve a WebGL context returns `null`. You should always check the returned value before attempting to use the context:



Available for
download on
Wrox.com

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

//make sure <canvas> is completely supported
if (drawing.getContext){

    var gl = drawing.getContext("experimental-webgl");
    if (gl){
        //proceed with WebGL
    }
}
```

WebGLExample01.htm

The WebGL context object is typically called `gl`. Most WebGL applications and examples use this convention because OpenGL ES 2.0 methods and values typically begin with "`gl`". Doing so means the JavaScript code reads more closely like an OpenGL program.

Once the WebGL context is established, you're ready to start 3D drawing. As mentioned previously, since WebGL is a web version of OpenGL ES 2.0, the concepts discussed in this section are really OpenGL concepts as implemented in JavaScript.

You can specify options for the WebGL context by passing in a second argument to `getContext()`. The argument is an object containing one or more of the following properties:

- `alpha` — When set to `true`, creates an alpha channel buffer for the context. Default is `true`.
- `depth` — When set to `true`, a 16-bit depth buffer is available. Default is `true`.
- `stencil` — When set to `true`, an 8-bit stencil buffer is available. Default is `false`.
- `antialias` — When set to `true`, antialiasing will be performed using the default mechanism. Default is `true`.
- `premultipliedAlpha` — When set to `true`, the drawing buffer is assumed to have premultiplied alpha values. Default is `true`.
- `preserveDrawingBuffer` — When set to `true`, the drawing buffer is preserved after drawing is completed. Default is `false`. Recommended to change only if you know exactly what this does, as there may be performance implications.

The options object is passed in like this:



Available for download on Wrox.com

```
var drawing = document.getElementById("drawing");
//make sure <canvas> is completely supported
if (drawing.getContext){

    var gl = drawing.getContext("experimental-webgl", { alpha: false});
    if (gl){
        //proceed with WebGL
    }
}
```

[WebGLExample01.htm](#)

Most of the context options are for advanced use. In many cases, the default values will serve your purpose.

Some browsers may throw an error if the WebGL context can't be created via `getContext()`. For that reason, it's best to wrap the call in a `try-catch` block:

```
Insert IconMargin [download]var drawing = document.getElementById("drawing"),
gl;

//make sure <canvas> is completely supported
if (drawing.getContext){
    try {
        gl = drawing.getContext("experimental-webgl");
    } catch (ex) {
        //noop
    }

    if (gl){
        //proceed with WebGL
    } else {
        alert("WebGL context could not be created.");
    }
}
```

[WebGLExample01.htm](#)

Constants

If you're familiar with OpenGL, then you're familiar with the large number of constants used for operations. These constants are named in OpenGL with a prefix of `GL_`. In WebGL, each constant is available on the `WebGL context` object without the `GL_` prefix. For example, the `GL_COLOR_BUFFER_BIT` constant is available as `gl.COLOR_BUFFER_BIT`. WebGL supports most OpenGL constants in this manner (some constants are not available).

Method Naming

Many method names in OpenGL, and so also in WebGL, tend to include information about the type of data to be used with the method. If a method can accept different types and numbers of arguments then it is suffixed to indicate the expected input. The method will indicate the number of arguments (1 through 4) followed by the data type (“f” for float and “i” for int). For example, `gl.uniform4f()` expects four floats to be passed in and `gl.uniform3i()` expects three integers to be passed in.

Many methods also allow an array to be passed in instead of individual arguments. This is indicated by the letter “v,” which is short for vector. So `gl.uniform3iv()` accepts an array of integers with three values. Keep this convention in mind throughout the discussion of WebGL.

Getting Ready to Draw

One of the first steps when working on a WebGL context is to clear the `<canvas>` with a solid color to prepare for drawing. To do this, you first must assign the color to use via the `clearColor()` method. This method accepts four arguments: red, green, blue, and alpha. Each argument must be a number between 0 and 1 defining the strength of value as part of a final color. Consider the following example:



```
gl.clearColor(0,0,0,1);      //black
gl.clear(gl.COLOR_BUFFER_BIT);
```

WebGLExample01.htm

This code sets the clear color buffer value to black and then calls the `clear()` method, which is the equivalent of `glClear()` in OpenGL. Providing the argument `gl.COLOR_BUFFER_BIT` tells WebGL to use the previously defined color to fill the area. Generally speaking, all drawing operations begin with a call to clear the area for drawing.

Viewports and Coordinates

To get started, it’s a good idea to define the WebGL viewport. By default, the viewport is set to use the entire `<canvas>` area. To change the viewport, call the `viewport()` method and pass in the x, y, width, and height of the viewport relative to the `<canvas>` element. For example, this call uses the entire `<canvas>` element:

```
gl.viewport(0, 0, drawing.width, drawing.height);
```

The viewport is defined using a different coordinate system than is typically used in a web page. The x- and y-coordinates start with (0,0) at the bottom-left of the `<canvas>` element and increase toward the top and right, which can be defined as point `(width-1, height-1)` (see Figure 15-15).

Knowing how the viewport is defined allows you to use just a part of the `<canvas>` element for drawing. Consider the following examples:

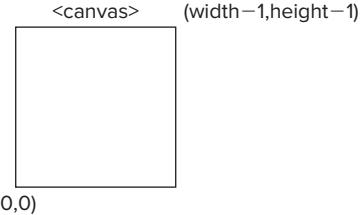


FIGURE 15-15

```
//viewport is a quarter of the <canvas> in the lower-left corner
gl.viewport(0, 0, drawing.width/2, drawing.height/2);

//viewport is a quarter of the <canvas> in the upper-left corner
gl.viewport(0, drawing.height/2, drawing.width/2, drawing.height/2);

//viewport is a quarter of the <canvas> in the lower-right corner
gl.viewport(drawing.width/2, 0, drawing.width/2, drawing.height/2);
```

The coordinate system within a viewport is different than the coordinate system for defining a viewport. Inside of a viewport, the coordinates start with point (0,0) in the center of the viewport. The lower-left corner is (-1,-1) while the upper-right is (1,1) (see Figure 15-16).

If a coordinate outside of the viewport is used for a drawing operation then the drawing is clipped along the viewport. For instance, attempting to draw a shape with a vertex at (1,2) will result in a shape that is cut off on the right side of the viewport.

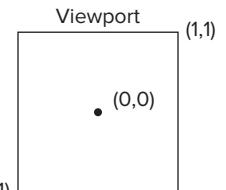


FIGURE 15-16

Buffers

Vertex information is stored in typed arrays in JavaScript and must be converted into WebGL buffers for use. Buffers are created by calling `gl.createBuffer()` and then bound to the WebGL context using `gl.bindBuffer()`. Once that happens, you can fill the buffer with data. For example:

```
var buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([0, 0.5, 1]), gl.STATIC_DRAW);
```

The call to `gl.bindBuffer()` sets `buffer` as the current buffer for the context. After that point, all buffer operations are performed on `buffer` directly. So the call to `gl.bufferData()` doesn't contain a direct reference to `buffer` but works on it nonetheless. That last line initializes `buffer` with information from a `Float32Array` (you generally will be using `Float32Array` for all vertex information). You can also use `gl.ELEMENT_ARRAY_BUFFER` if you intend to use `drawElements()` for outputting the buffer content.

The last argument of `gl.bufferData()` indicates how the buffer will be used. This is one of the following constants:

- `gl.STATIC_DRAW` — The data will be loaded once and used for drawing multiple times.
- `gl.STREAM_DRAW` — The data will be loaded once and used for drawing just a few times.
- `gl.DYNAMIC_DRAW` — The data will be modified repeatedly and used for drawing multiple times.

You'll likely use `gl.STATIC_DRAW` for most buffers unless you're an experienced OpenGL programmer.

Buffers stay in memory until the containing page is unloaded. If you no longer need a buffer, then it's best to free its memory by calling `gl.deleteBuffer()`:

```
gl.deleteBuffer(buffer);
```

Errors

One of the differences between most JavaScript and WebGL is that errors are generally not thrown from WebGL operations. Instead, you must call the `gl.getError()` method after invoking a method that may have failed. This method returns a constant value indicating the type of error that has occurred. The constants are as follows:

- `gl.NO_ERROR` — There wasn't an error during the last operation (value of 0).
- `gl.INVALID_ENUM` — An incorrect argument was passed to a method that was expecting one of the WebGL constants.
- `gl.INVALID_VALUE` — A negative number was passed where only an unsigned number is accepted.
- `gl.INVALID_OPERATION` — The operation cannot be completed in the current state.
- `gl.OUT_OF_MEMORY` — There is not enough memory to complete the operation.
- `gl.CONTEXT_LOST_WEBGL` — The WebGL context was lost because of an external event (such as loss of power on a device).

Each call to `gl.getError()` returns a single error value. After the initial call, the next call to `gl.getError()` may return another error value. If there are multiple errors, then this process continues until `gl.getError()` returns `gl.NO_ERROR`. If you have performed a number of operations, then you'll likely want to call `getError()` in a loop, such as:

```
var errorCode = gl.getError();
while(errorCode) {
    console.log("Error occurred: " + errorCode);
    errorCode = gl.getError();
}
```

If your WebGL script is not resulting in the correct output, then putting a few calls to `gl.getError()` into your script may help debug the issue.

Shaders

Shaders are another concept from OpenGL. There are two types of shaders in WebGL: *vertex shaders* and *fragment shaders*. Vertex shaders are used to convert a 3D vertex into a 2D point to be rendered. Fragment shaders are used to compute the correct color for drawing a single pixel. The unique and challenging aspect of WebGL shaders is that they are not written in JavaScript. Shaders are written using *OpenGL Shading Language (GLSL)*, a completely separate language from C or JavaScript.

Writing Shaders

GLSL is a C-like language that is used specifically for defining OpenGL shaders. Since WebGL is an implementation of OpenGL ES 2, the shaders used in OpenGL can be used directly in WebGL, allowing for easy porting of desktop graphics to the Web.

Each shader has a method called `main()` that is executed repeatedly during drawing. There are two ways to pass data into a shader: *attributes* and *uniforms*. Attributes are used to pass vertices into a vertex shader while uniforms are used to pass constant values to either type of shader. Attributes and uniforms are defined outside of `main()` by using the keywords `attribute` or `uniform`, respectively. After the value type keyword, the data type is specified followed by a name. Here's a simple example vertex shader:



Available for
download on
Wrox.com

```
//OpenGL Shading Language
//Shader by Bartek Drozdz in his article at
//http://www.netmagazine.com/tutorials/get-started-webgl-draw-square
attribute vec2 aVertexPosition;

void main() {
    gl_Position = vec4(aVertexPosition, 0.0, 1.0);
}
```

[WebGLExample02.htm](#)

This vertex shader defines a single attribute called `aVertexPosition`. This attribute is an array of two items (`vec2` data type) representing an x- and y-coordinate. A vertex shader must always result in a four-part vertex being assigned to the special variable `gl_Position` even though only two coordinates were passed. This shader creates a new four-item array (`vec4`) and fills in the missing coordinates, effectively turning a 2D coordinate into a 3D one.

Fragment shaders are similar to vertex shaders except you can pass data only in via uniforms. Here's an example fragment shader:

```
//OpenGL Shading Language
//Shader by Bartek Drozdz in his article at
//http://www.netmagazine.com/tutorials/get-started-webgl-draw-square
uniform vec4 uColor;

void main() {
    gl_FragColor = uColor;
}
```

[WebGLExample02.htm](#)

Fragment shaders must result in a value being assigned to `gl_FragColor`, which indicates the color to use while drawing. This shader defined a uniform four-part (`vec4`) color named `uColor` to be set. Literally, this shader does nothing but assign the passed-in value to `gl_FragColor`. The value of `uColor` cannot be changed within the shader.



OpenGL Shading Language is a more complex language than represented here. There are entire books devoted to explaining the intricacies of the languages, and so this section is just a quick introduction to the language as a way of facilitating WebGL usage. For more information, please read OpenGL Shading Language by Randi J. Rost (Addison-Wesley, 2006).

Creating Shader Programs

GLSL cannot be natively understood by a browser, so you must have a string of GLSL ready for compilation and linking into a shader program. For ease of use, shaders are typically included in a page using `<script>` elements with a custom `type` attribute. Using an invalid `type` attribute prevents the browser from attempting to interpret the `<script>` contents while allowing you easy access. For example:



Available for download on Wrox.com

```
<script type="x-webgl/x-vertex-shader" id="vertexShader">
attribute vec2 aVertexPosition;

void main() {
    gl_Position = vec4(aVertexPosition, 0.0, 1.0);
}
</script>
<script type="x-webgl/x-fragment-shader" id="fragmentShader">
uniform vec4 uColor;

void main() {
    gl_FragColor = uColor;
}
</script>
```

[WebGLExample02.htm](#)

You can then extract the contents of the `<script>` element using the `text` property:

```
var vertexGls1 = document.getElementById("vertexShader").text,
fragmentGls1 = document.getElementById("fragmentShader").text;
```

More complex WebGL applications may choose to download shaders dynamically using Ajax (discussed in Chapter 21). The important aspect is that you need a GLSL string in order to use a shader.

Once you have a GLSL string, the next step is to create a `shader` object. This is done by calling the `gl.createShader()` method and passing in the type of shader to create (`gl.VERTEX_SHADER` or `gl.FRAGMENT_SHADER`). After that, the source code of the shader is applied using `gl.shaderSource()` and the shader is compiled using `gl.compileShader()`. Here's an example:

```
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexGls1);
gl.compileShader(vertexShader);

var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
```

```
gl.shaderSource(fragmentShader, fragmentGlsl);
gl.compileShader(fragmentShader);
```

[WebGLExample02.htm](#)

This code creates two shaders and stores them in `vertexShader` and `fragmentShader`. These two objects can then be linked into a shader program by using the following code:

```
var program = gl.createProgram();
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);
```

[WebGLExample02.htm](#)

The first line creates a program and then `attachShader()` is used to include the two shaders. The call to `gl.linkProgram()` encapsulates both shaders together into the variable `program`. With the program linked, you can instruct the WebGL context to use the program via the `gl.useProgram()` method:

```
gl.useProgram(program);
```

After `gl.useProgram()` has been called, all further drawing operations will use the specified program.

Passing Values to Shaders

Each of the previously defined shaders has a value that must be passed in to complete the shader's job. To pass values into a shader, you must first locate the variable whose value must be filled. For uniform variables, this is done through `gl.getUniformLocation()`, which returns an object representing the location of the uniform variable in memory. You can then use this location to assign data. For example:



Available for download on
Wrox.com

```
var uColor = gl.getUniformLocation(program, "uColor");
gl.uniform4fv(uColor, [0, 0, 0, 1]);
```

[WebGLExample02.htm](#)

This example locates the uniform variable `uColor` in `program` and returns its memory location. The second line assigns a value into `uColor` using `gl.uniform4fv()`.

A similar process is followed for attribute variables in vertex shaders. To get the location of an attribute variable, use `gl.getAttributeLocation()`. Once the location is retrieved, it can be used as in this example:

```
var aVertexPosition = gl.getAttributeLocation(program, "aVertexPosition");
gl.enableVertexAttribArray(aVertexPosition);
gl.vertexAttribPointer(aVertexPosition, itemSize, gl.FLOAT, false, 0, 0);
```

[WebGLExample02.htm](#)

Here, the location of `aVertexPosition` is retrieved so that it may be enabled for use via `gl.enableVertexAttribArray()`. The last line creates a pointer into the last buffer specified using `gl.bindBuffer()` and stores it in `aVertexPosition` so that it may be used by the vertex shader.

Debugging Shaders and Programs

As with other operations in WebGL, shader operations may fail and will do so silently. You need to manually ask the WebGL context for information about the shader or program if you think there has been an error.

For shaders, call `gl.getShaderParameter()` to get the compiled status of the shader after attempting compilation:



```
if (!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)) {
    alert(gl.getShaderInfoLog(vertexShader));
}
```

[WebGLExample02.htm](#)

This example checks the compilation status of `vertexShader`. If the shader compiled successfully, then the call to `gl.getShaderParameter()` returns `true`. If the call returns `false`, then there was an error during compilation and you can retrieve the error by using `gl.getShaderInfoLog()` and passing in the shader. This method returns a string message indicating the issue. Both `gl.getShaderParameter()` and `gl.getShaderInfoLog()` may be used on vertex shaders and fragment shaders.

Programs may also fail and have a similar method, `gl.getProgramParameter()`, to check status. The most common program failure is during the linking process, for which you would check using the following code:

```
if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
    alert(gl.getProgramInfoLog(program));
}
```

[WebGLExample02.htm](#)

As with `gl.getShaderParameter()`, the `gl.getProgramParameter()` returns either `true` to indicate that the link succeeded or `false` to indicate it failed. There is also `gl.getProgramInfoLog()`, which is used to get information about the program during failures.

These methods are primarily used during development to aid in debugging. As long as there are no external dependencies, it's safe to remove them in production.

Drawing

WebGL can draw only three types of shapes: points, lines, and triangles. All other shapes must be composed using a combination of these three basic shapes drawn in three-dimensional space. Drawing is executed by using the `drawArrays()` or `drawElements()` methods; the former works on array buffers while the latter acts on element array buffers.

The first argument for both `gl.drawArrays()` and `drawElements()` is a constant indicating the type of shape to draw. The constants are:

- `gl.POINTS` — Treats each vertex as a single point to be drawn.
- `gl.LINES` — Treats the array as a series of vertices between which to draw lines. Each set of vertices is a start point and an end point, so you must have an even number of vertices in the array for all drawing to take place.
- `gl.LINE_LOOP` — Treats the array as a series of vertices between which to draw lines. The line is drawn from the first vertex to the second, from the second to the third, etc., until the last vertex is reached. A line is then drawn from the last vertex to the first vertex. This effectively creates an outline of a shape.
- `gl.LINE_STRIP` — Same as `gl.LINE_LOOP` except a line is not drawn from the last vertex back to the first.
- `gl.TRIANGLES` — Treats the array as a series of vertices within which triangles should be drawn. Each triangle is drawn separately from the previous without sharing vertex unless explicitly specified.
- `gl.TRIANGLES_STRIP` — Same as `gl.TRIANGLES` except vertices after the first three are treated as the third vertex for a new triangle made with the previous two vertices. For example, if an array contains vertices A, B, C, D, the first triangle is drawn as ABC while the second is drawn as BCD.
- `gl.TRIANGLES_FAN` — Same as `gl.TRIANGLES` except vertices after the first three are treated as the third vertex for a triangle made with the previous vertex and the first coordinate. For example, if an array contains vertices A, B, C, D, the first triangle is drawn as ABC while the second is drawn as ACD.

The `gl.drawArrays()` method accepts one of these values as its first argument, the starting index within the array buffer as the second argument, and the number of sets contained in the array buffer as the third argument. The following code uses `gl.drawArrays()` to draw a single triangle across the canvas:



Available for
download on
[Wrox.com](#)

```
//assume viewport is cleared using the shaders from earlier in the section

//define three vertices, x and y for each
var vertices = new Float32Array([ 0, 1, 1, -1, -1, -1 ]),
    buffer = gl.createBuffer(),
    vertexSetSize = 2,
    vertexSetCount = vertices.length/vertexSetSize,
    uColor, aVertexPosition;

//put data into the buffer
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

//pass color to fragment shader
uColor = gl.getUniformLocation(program, "uColor");
gl.uniform4fv(uColor, [ 0, 0, 0, 1 ]);

//pass vertex information to shader
```

```
aVertexPosition = gl.getAttribLocation(program, "aVertexPosition");
gl.enableVertexAttribArray(aVertexPosition);
gl.vertexAttribPointer(aVertexPosition, vertexSetSize, gl.FLOAT, false, 0, 0);

//draw the triangle
gl.drawArrays(gl.TRIANGLES, 0, vertexSetCount);
```

[WebGLExample02.htm](#)

This example defines a `Float32Array` containing three sets of two-point vertices. It's important to keep track of the size and number of vertex sets for use in later calculations. The `vertexSetSize` is set to 2 while the `vertexSetCount` is calculated. The vertex information is stored in a buffer. Color information is then passed to the fragment shader.

The vertex shader is passed the size of the vertex set and indicates that the vertex coordinates are floats (`gl.FLOAT`). The fourth argument is a Boolean indicating that the coordinates are not normalized. The fifth argument is the *stride value*, which indicates how many array items need to be skipped to get the next value. This is 0 unless you really know what you're doing. The last argument is the starting offset, which is 0 to start at the first item.

The last step is to draw the triangle by using `gl.drawArrays()`. By specifying the first argument as `gl.TRIANGLES`, a triangle will be drawn from (0,1) to (1,-1) to (-1,-1) and filled in with the color passed to the fragment shader. The second argument is the starting offset in the buffer, and the last argument is the total number of vertex sets to read. The result of this drawing operation is displayed in Figure 15-17.

By changing the first argument to `gl.drawArrays()`, you can change how the triangle is drawn. Figure 15-18 shows some other possible outputs based on changing the first argument.

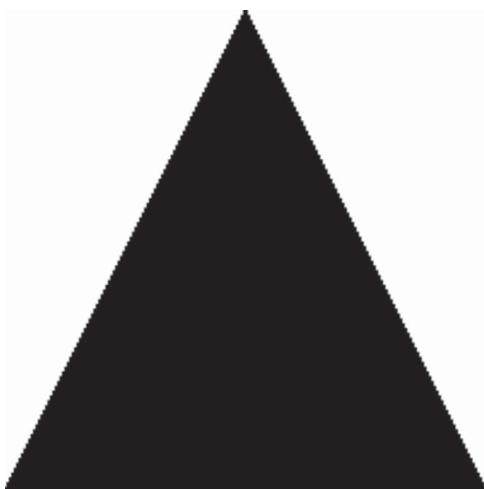


FIGURE 15-17

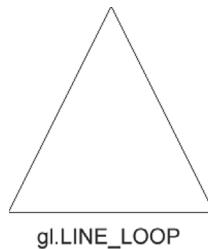
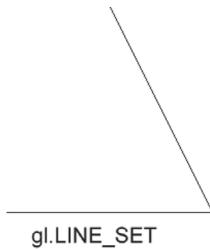


FIGURE 15-18



gl.LINE_SET

Textures

WebGL textures work together with images from the DOM. You create a new texture using `gl.createTexture()` and then bind an image to that texture. If the image isn't already loaded, then you may create a new instance of `Image` to dynamically load it. A texture isn't initialized until the image is completely loaded, so texture setup steps must be done after the `load` event has fired. For example:

```
var image = new Image(),
    texture;
image.src = "smile.gif";
image.onload = function(){
    texture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);

    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

    //clear current texture
    gl.bindTexture(gl.TEXTURE_2D, null);
}
```

Aside from using a DOM image, these steps are the same for creating texture in OpenGL. The biggest difference is in setting the pixel storage format with `gl.pixelStorei()`. The constant `gl.UNPACK_FLIP_Y_WEBGL` is unique to WebGL and must be used in most circumstances when loading Web-based images. This is because of the different coordinate systems used by GIF, JPEG, and PNG images as compared to the internal coordinate system of WebGL. Without this flag, the image is interpreted upside down.

Images used for textures must be of the same origin as the containing page or else exist on servers that have Cross-Origin Resource Sharing (CORS) enabled for the images. CORS is discussed in Chapter 21.



Texture sources may be images, videos loaded into the <video> element, and even another <canvas> element. The same restrictions regarding cross-origin resources apply to videos.

Reading Pixels

As with the 2D context, it's possible to read pixels from the WebGL context. The `readPixels()` method has the same arguments as in OpenGL with the exception that the last argument must be a typed array. Pixel information is read from the frame buffer and placed into the typed array. The arguments for `readPixels()` are `x`, `y`, `width`, `height`, `image format`, `type`, and `typed array`. The first four arguments specify the location of the pixels to read. The `image format` argument will almost

always be `gl.RGBA`. The type argument is the type of data that will be stored in the typed array and has the following restrictions:

- If the type is `gl.UNSIGNED_BYTE`, then the typed array must be `Uint8Array`.
- If the type is `gl.UNSIGNED_SHORT_5_6_5`, `gl.UNSIGNED_SHORT_4_4_4`, or `gl.UNSIGNED_SHORT_5_5_5_1`, then the typed array must be `Uint16Array`.

Here's a simple example:

```
var pixels = new Uint8Array(25*25);
gl.readPixels(0, 0, 25, 25, gl.RGBA, gl.UNSIGNED_BYTE, pixels);
```

This code reads a 25×25 area of the frame buffer and stores the pixel information in the `pixels` array. Each pixel color is represented as four array items, one each for red, green, blue, and alpha. The values are numbers 0 through 255, inclusive. Don't forget to initialize the typed array for the amount of data you're expecting back.

Calling `readPixels()` before the browser has drawn the updated WebGL image works as expected. After the paint has occurred, the frame buffer is reverted to its original cleared state and calling `readPixels()` will result in pixel data matching the cleared state. If you want to read pixels after the paint has occurred, then you must initialize the WebGL context with the `preserveDrawingBuffer` option discussed previously:

```
var gl = drawing.getContext("experimental-webgl", { preserveDrawingBuffer: true; });
```

Setting this flag forces the frame buffer to stay in its last state until the next draw occurs. This option does have some performance overhead, so it's best to avoid using if possible.

Support

The WebGL API has been implemented in Firefox 4+ and Chrome. Safari 5.1 has implemented WebGL but has it disabled by default. WebGL is unique in that having a particular browser version doesn't automatically ensure support. Two things must happen for a browser to support WebGL. First, the browser itself must have implemented the API. Second, the computer must have updated graphics card drivers. Older computers, such as those running Windows XP, typically have out-of-date drivers, and so WebGL will be disabled in browsers running on those computers. For this reason, it's important to specifically check for support of WebGL rather than particular browser versions.

Keep in mind that the WebGL specification is still undergoing development and changes. Everything from function names to function signatures to data types is in a state of flux, so while WebGL may be fun to experiment with, it is definitely not suitable for production use quite yet.

SUMMARY

The HTML5 `<canvas>` element provides a JavaScript API for creating graphics on the fly. Graphics are created in a specific context, of which there are currently two. The first is a 2D context that allows primitive drawing operations:

- Setting fill and stroke colors and patterns
- Drawing rectangles
- Drawing paths
- Drawing text
- Creating gradients and patterns

The second context is a 3D context called WebGL. WebGL is a browser port of OpenGL ES 2.0, a language frequently used by game developers for computer graphics. WebGL allows far more powerful graphics processing than the 2D context, providing:

- Vertex and fragment shaders written in OpenGL Shading Language (GLSL)
- Typed array support, limiting the type of data contained in an array to specific numeric types
- Texture creation and manipulation

The `<canvas>` tag itself is now widely supported and is available in the most recent version of all major browsers. Support for the 2D context is also available in the same browsers while WebGL support is limited to Firefox 4+ and Chrome.

16

HTML5 Scripting

WHAT'S IN THIS CHAPTER?

- Using cross-document messaging
- Drag-and-drop APIs
- Working with audio and video

As discussed earlier in the book, the HTML5 specification defines much more than HTML markup. A significant portion of the HTML5 defines JavaScript APIs that are intended to work in concert with the markup changes. The goal of these APIs is to make previously difficult tasks easier with the ultimate goal of allowing the creation of dynamic web interfaces.

CROSS-DOCUMENT MESSAGING

Cross-document messaging, sometimes abbreviated as XDM, is the ability to pass information between pages from different origins. For example, a page on www.wrox.com wants to communicate with a page from p2p.wrox.com that is contained in an `<iframe>`. Prior to XDM, achieving this communication in a secure manner took a lot of work. XDM formalizes this functionality in a way that is both secure and easy to use.

At the heart of XDM is the `postMessage()` method. This method name is used in many parts of HTML5 in addition to XDM and is always used for the same purpose: to pass data into another location. In the case of XDM, that other location is an `<iframe>` element or pop-up window owned by the page.

The `postMessage()` method accepts two arguments: a message and a string indicating the intended recipient origin. The second argument is very important for security reasons and restricts where the browser will deliver the message. Consider this example:

```
//note: all browsers that support XDM also support iframe contentWindow
var iframeWindow = document.getElementById("myframe").contentWindow;
iframeWindow.postMessage("A secret", "http://www.wrox.com");
```

The last line attempts to send a message into the iframe and specifies that the origin must be "http://www.wrox.com". If the origin matches, then the message will be delivered into the iframe; otherwise `postMessage()` silently does nothing. This restriction protects your information should the location of the window change without your knowledge. It is possible to allow posting to any origin by passing in "*" as the second argument to `postMessage()`, but this is not recommended.

A message event is fired on a window when an XDM message is received. This message is fired asynchronously so there may be a delay between the time at which the message was sent and the time at which the message event is fired in the receiving window. The event object that is passed to an `onmessage` event handler has three important pieces of information:

- `data` — The string data that was passed as the first argument to `postMessage()`.
- `origin` — The origin of the document that sent the message, for example, "http://www.wrox.com".
- `source` — A proxy for the `window` object of the document that sent the message. This proxy object is used primarily to execute the `postMessage()` method on the window that sent the last message. If the sending window has the same origin, this may be the actual `window` object.

It's very important when receiving a message to verify the origin of the sending window. Just like specifying the second argument to `postMessage()` ensures that data doesn't get passed unintentionally to an unknown page, checking the origin during `onmessage` ensures that the data being passed is coming from the right place. The basic pattern is as follows:

```
EventUtil.addHandler(window, "message", function(event) {
    //ensure the sender is expected
    if (event.origin == "http://www.wrox.com") {
        //do something with the data
        processMessage(event.data);

        //optional: send a message back to the original window
        event.source.postMessage("Received!", "http://p2p.wrox.com");
    }
});
```

Keep in mind that `event.source` is a proxy for a `window` in most cases, not the actual `window` object, so you can't access all of the window information. It's best to just use `postMessage()`, which is always present and always callable.

There are a few quirks with XDM. First, the first argument of `postMessage()` was initially implemented as always being a string. The definition of that first argument changed to allow any structured data to be passed in; however, not all browsers have implemented this change. For this reason, it's best to always pass a string using `postMessage()`. If you need to pass structured data, then the best approach is to call `JSON.stringify()` on the data, passing the string to `postMessage()`, and then call `JSON.parse()` in the `onmessage` event handler.

XDM is extremely useful when trying to sandbox content using an iframe to a different domain. This approach is frequently used in mashups and social networking applications. The containing page is able to keep itself secure against malicious content by only communicating into an embedded iframe via XDM. XDM can also be used with pages from the same domain.

XDM is supported in Internet Explorer 8+, Firefox 3.5+, Safari 4+, Opera, Chrome, Safari on iOS, and WebKit on Android. XDM was separated out into its own specification, which is now called Web Messaging and is found at <http://dev.w3.org/html5/postmsg/>.

NATIVE DRAG AND DROP

Internet Explorer 4 first introduced JavaScript support for drag-and-drop functionality for web pages. At the time, only two items on a web page could initiate a system drag: an image or some text. When dragging an image, you simply held the mouse button down and then moved it; with text, you first highlighted some text and then you could drag it the same way as you would drag an image. In Internet Explorer 4, the only valid drop target was a text box. In version 5, Internet Explorer extended its drag-and-drop capabilities by adding new events and allowing nearly anything on a web page to become a drop target. Version 5.5 went a little bit further by allowing nearly anything to become draggable. (Internet Explorer 6 supports this functionality as well.) HTML5 uses the Internet Explorer drag-and-drop implementation as the basis for its drag-and-drop specification. Firefox 3.5, Safari 3+, and Chrome have also implemented native drag and drop according to the HTML5 spec.

Perhaps the most interesting thing about drag-and-drop support is that elements can be dragged across frames, browser windows, and sometimes, other applications. Drag-and-drop support in the browser allows you to tap into that functionality.

Drag-and-Drop Events

The events provided for drag and drop enable you to control nearly every aspect of a drag-and-drop operation. The tricky part is determining where each event is fired: some fire on the dragged item; others fire on the drop target. When an item is dragged, the following events fire (in this order):

1. dragstart
2. drag
3. dragend

At the moment you hold a mouse button down and begin to move the mouse, the `dragstart` event fires on the item that is being dragged. The cursor changes to the no-drop symbol (a circle with a line through it), indicating that the item cannot be dropped on itself. You can use the `ondragstart` event handler to run JavaScript code as the dragging begins.

After the `dragstart` event fires, the `drag` event fires and continues firing as long as the object is being dragged. This is similar to `mousemove`, which also fires repeatedly as the mouse is moved. When the dragging stops (because you drop the item onto either a valid or an invalid drop target), the `dragend` event fires.

The target of all three events is the element that is being dragged. By default, the browser does not change the appearance of the dragged element while a drag is happening, so it's up to you to change the appearance. Most browsers do, however, create a semitransparent clone of the element being dragged that always stays immediately under the cursor.

When an item is dragged over a valid drop target, the following sequence of events occurs:

1. dragenter
2. dragover
3. dragleave or drop

The `dragenter` event (similar to the `mouseover` event) fires as soon as the item is dragged over the drop target. Immediately after the `dragenter` event fires, the `dragover` event fires and continues to fire as the item is being dragged within the boundaries of the drop target. When the item is dragged outside of the drop target, `dragover` stops firing and the `dragleave` event is fired (similar to `mouseout`). If the dragged item is actually dropped on the target, the `drop` event fires instead of `dragleave`. The target of these events is the drop target element.

Custom Drop Targets

When you try to drag something over an invalid drop target, you see a special cursor (a circle with a line through it) indicating that you cannot drop. Even though all elements support the drop target events, the default is to not allow dropping. If you drag an element over something that doesn't allow a drop, the `drop` event will never fire regardless of the user action. However, you can turn any element into a valid drop target by overriding the default behavior of both the `dragenter` and the `dragover` events. For example, if you have a `<div>` element with an ID of "droptarget", you can use the following code to turn it into a drop target:

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

EventUtil.addHandler(droptarget, "dragover", function(event){
    EventUtil.preventDefault(event);
});

EventUtil.addHandler(droptarget, "dragenter", function(event){
    EventUtil.preventDefault(event);
});
```

After making these changes, you'll note that the cursor now indicates that a drop is allowed over the drop target when dragging an element. Also, the `drop` event will fire.

In Firefox 3.5+, the default behavior for a `drop` event is to navigate to the URL that was dropped on the drop target. That means dropping an image onto the drop target will result in the page navigating to the image file, and text that is dropped on the drop target results in an invalid URL error. For Firefox support, you must also cancel the default behavior of the `drop` event to prevent this navigation from happening:

```
EventUtil.addHandler(droptarget, "drop", function(event){
    EventUtil.preventDefault(event);
});
```

The dataTransfer Object

Simply dragging and dropping isn't of any use unless data is actually being affected. To aid in the transmission of data via a drag-and-drop operation, Internet Explorer 5 introduced the `dataTransfer` object, which exists as a property of `event` and is used to transfer string data from the dragged item to the drop target. Because it is a property of `event`, the `dataTransfer` object doesn't exist except within the scope of an event handler for a drag-and-drop event. Within an event handler, you can use the object's properties and methods to work with your drag-and-drop functionality. The `dataTransfer` object is now part of the working draft of HTML5.

The `dataTransfer` object has two primary methods: `getData()` and `setData()`. As you might expect, `getData()` is capable of retrieving a value stored by `setData()`. The first argument for `setData()`, and the only argument of `getData()`, is a string indicating the type of data being set: either "text" or "URL", as shown here:

```
//working with text
event.dataTransfer.setData("text", "some text");
var text = event.dataTransfer.getData("text");

//working with a URL
event.dataTransfer.setData("URL", "http://www.wrox.com/");
var url = event.dataTransfer.getData("URL");
```

Even though Internet Explorer started out by introducing only "text" and "URL" as valid data types, HTML5 extends this to allow any MIME type to be specified. The values "text" and "URL" will be supported by HTML5 for backwards compatibility, but they are mapped to "text/plain" and "text/uri-list".

The `dataTransfer` object can contain exactly one value of each MIME type, meaning that you can store both text and a URL at the same time without overwriting either. The data stored in the `dataTransfer` object is available only until the drop event. If you do not retrieve the data in the `ondrop` event handler, the `dataTransfer` object is destroyed and the data is lost.

When you drag text from a text box, the browser calls `setData()` and stores the dragged text in the "text" format. Likewise, when a link or image is dragged, `setData()` is called and the URL is stored. It is possible to retrieve these values when the data is dropped on a target by using `getData()`. You can also call `setData()` manually during the `dragstart` event to store custom data that you may want to retrieve later.

There is a difference between data treated as text and data treated as a URL. When you specify data to be stored as text, it gets no special treatment whatsoever. When you specify data to be stored as a URL, however, it is treated just like a link on a web page, meaning that if you drop it onto another browser window, the browser will navigate to that URL.

Firefox through version 5 doesn't properly alias "url" to "text/uri-list" or "text" to "text/plain". It does, however, alias "Text" (uppercase T) to "text/plain". For best cross-browser

compatibility of retrieving data from `dataTransfer`, you'll need to check for two values for URLs and use "Text" for plain text:



```
var dataTransfer = event.dataTransfer;
//read a URL
var url = dataTransfer.getData("url") || dataTransfer.getData("text/uri-list");

//read text
var text = dataTransfer.getData("Text");
```

DataTransferExample01.htm

It's important that the shortened data name be tried first, because Internet Explorer through version 10 doesn't support the extended names and also throws an error when the data name isn't recognized.

dropEffect and effectAllowed

The `dataTransfer` object can be used to do more than simply transport data to and fro; it can also be used to determine what type of actions can be done with the dragged item and the drop target. You accomplish this by using two properties: `dropEffect` and `effectAllowed`.

The `dropEffect` property is used to tell the browser which type of drop behaviors are allowed. This property has the following four possible values:

- "none" — A dragged item cannot be dropped here. This is the default value for everything except text boxes.
- "move" — The dragged item should be moved to the drop target.
- "copy" — The dragged item should be copied to the drop target.
- "link" — Indicates that the drop target will navigate to the dragged item (but only if it is a URL).

Each of these values causes a different cursor to be displayed when an item is dragged over the drop target. It is up to you, however, to actually cause the actions indicated by the cursor. In other words, nothing is automatically moved, copied, or linked without your direct intervention. The only thing you get for free is the cursor change. In order to use the `dropEffect` property, you must set it in the `ondragenter` event handler for the drop target.

The `dropEffect` property is useless, unless you also set the `effectAllowed`. This property indicates which `dropEffect` is allowed for the dragged item. The possible values are as follows:

- "uninitialized" — No action has been set for the dragged item.
- "none" — No action is allowed on the dragged item.
- "copy" — Only `dropEffect "copy"` is allowed.
- "link" — Only `dropEffect "link"` is allowed.
- "move" — Only `dropEffect "move"` is allowed.

- "copyLink" — dropEffect "copy" and "link" are allowed.
- "copyMove" — dropEffect "copy" and "move" are allowed.
- "linkMove" — dropEffect "link" and "move" are allowed.
- "all" — All dropEffect values are allowed.

This property must be set inside the `ondragstart` event handler.

Suppose that you want to allow a user to move text from a text box into a `<div>`. To accomplish this, you must set both `dropEffect` and `effectAllowed` to "move". The text won't automatically move itself, because the default behavior for the drop event on a `<div>` is to do nothing. If you override the default behavior, the text is automatically removed from the text box. It is then up to you to insert it into the `<div>` to finish the action. If you were to change `dropEffect` and `effectAllowed` to "copy", the text in the text box would not automatically be removed.



Firefox through version 5 has an issue with `effectAllowed` where the drop event may not fire when this value is set in code.

Draggability

By default, images, links, and text are draggable, meaning that no additional code is necessary to allow a user to start dragging them. Text is draggable only after a section has been highlighted, while images and links may be dragged at any point in time.

It is possible to make other elements draggable. HTML5 specifies a `draggable` property on all HTML elements indicating if the element can be dragged. Images and links have `draggable` automatically set to `true`, whereas everything else has a default value of `false`. This property can be set in order to allow other elements to be draggable or to ensure that an image or link won't be draggable. For example:

```
<!-- turn off dragging for this image -->
![Smiley face](smile.gif)<...>
```

The `draggable` attribute is supported in Internet Explorer 10+, Firefox 4+, Safari 5+, and Chrome. Opera, as of version 11.5, does not support HTML5 drag and drop. In order for Firefox to initiate the drag, you must also add an `ondragstart` event handler that sets some information on the `dataTransfer` object.



Internet Explorer 9 and earlier allow you to make any element draggable by calling the `dragDrop()` method on it during the `mousedown` event. Safari 4 and earlier required the addition of a CSS style `-khtml-user-drag: element` to make an element draggable.

Additional Members

The HTML5 specification indicates the following additional methods on the `dataTransfer` object:

- `addElement(element)` — Adds an element to the drag operation. This is purely for data purposes and doesn't affect the appearance of the drag operation. As of the time of this writing, no browsers have implemented this method.
- `clearData(format)` — Clears the data being stored with the particular format. This has been implemented in Internet Explorer, Firefox 3.5+, Chrome, and Safari 4+.
- `setDragImage(element, x, y)` — Allows you to specify an image to be displayed under the cursor as the drag takes place. This method accepts three arguments: an HTML element to display and the x- and y-coordinates on the image where the cursor should be positioned. The HTML element may be an image, in which case the image is displayed, or any other element, in which case a rendering of the element is displayed. Firefox 3.5+, Safari 4+, and Chrome all support this method.
- `types` — A list of data types currently being stored. This collection acts like an array and stores the data types as strings such as "text". Internet Explorer 10+, Firefox 3.5+, and Chrome implemented this property.

MEDIA ELEMENTS

With the explosive popularity of embedded audio and video on the Web, most content producers have been forced to use Flash for optimal cross-browser compatibility. HTML5 introduces two media-related elements to enable cross-browser audio and video embedding into a browser baseline without any plug-ins: `<audio>` and `<video>`.

Both of these elements allow web developers to easily embed media files into a page, as well as provide JavaScript hooks into common functionality, allowing custom controls to be created for the media. The elements are used as follows:

```
<!-- embed a video -->
<video src="conference.mpg" id="myVideo">Video player not available.</video>

<!-- embed an audio file -->
<audio src="song.mp3" id="myAudio">Audio player not available.</audio>
```

Each of these elements requires, at a minimum, the `src` attribute indicating the media file to load. You can also specify `width` and `height` attributes to indicate the intended dimensions of the video player and a `poster` attribute that is an image URI to display while the video content is being loaded. The `controls` attribute, if present, indicates that the browser should display a UI enabling the user to interact directly with the media. Any content between the opening and the closing tags is considered alternate content to display if the media player is unavailable.

You may optionally specify multiple different media sources, because not all browsers support all media formats. To do so, omit the `src` attribute from the element and instead include one or more `<source>` elements, as in this example:

```

<!-- embed a video -->
<video id="myVideo">
  <source src="conference.webm" type="video/webm; codecs='vp8, vorbis'">
  <source src="conference.ogv" type="video/ogg; codecs='theora, vorbis'">
  <source src="conference.mpg">
  Video player not available.
</video>

<!-- embed an audio file -->
<audio id="myAudio">
  <source src="song.ogg" type="audio/ogg">
  <source src="song.mp3" type="audio/mpeg">
  Audio player not available.
</audio>

```

It's beyond the scope of this book to discuss the various codecs used with video and audio, but suffice to say that browsers support a varying range of codecs, so multiple source files are typically required. The media elements are supported by Internet Explorer 9+, Firefox 3.5+, Safari 4+, Opera 10.5+, Chrome, Safari for iOS, and WebKit for Android.

Properties

The `<video>` and `<audio>` elements provide robust JavaScript interfaces. There are numerous properties shared by both elements that can be evaluated to determine the current state of the media, as described in the following table.

PROPERTY NAME	DATA TYPE	DESCRIPTION
autoplay	Boolean	Gets or sets the <code>autoplay</code> flag.
buffered	TimeRanges	An object indicating the buffered time ranges that have already been downloaded.
bufferedBytes	ByteRanges	An object indicating the buffered byte ranges that have already been downloaded.
bufferingRate	Integer	The average number of bits per second received from the download.
bufferingThrottled	Boolean	Indicates if the buffering has been throttled by the browser.
controls	Boolean	Gets or sets the <code>controls</code> attribute, which displays or hides the browser's built-in controls.
currentLoop	Integer	The number of loops that the media has played.
currentSrc	String	The URL for the currently playing media.
currentTime	Float	The number of seconds that have been played.

continues

(continued)

PROPERTY NAME	DATA TYPE	DESCRIPTION
defaultPlaybackRate	Float	Gets or sets the default playback rate. By default, this is 1.0 seconds.
duration	Float	The total number of seconds for the media.
ended	Boolean	Indicates if the media has completely played.
loop	Boolean	Gets or sets whether the media should loop back to the start when finished.
muted	Boolean	Gets or sets if the media is muted.
networkState	Integer	Indicates the current state of the network connection for the media: 0 for empty, 1 for loading, 2 for loading meta data, 3 for loaded first frame, and 4 for loaded.
paused	Boolean	Indicates if the player is paused.
playbackRate	Float	Gets or sets the current playback rate. This may be affected by the user causing the media to play faster or slower, unlike defaultPlaybackRate, which remains unchanged unless the developer changes it.
played	TimeRanges	The range of times that have been played thus far.
readyState	Integer	Indicates if the media is ready to be played. Values are 0 if the data is unavailable, 1 if the current frame can be displayed, 2 if the media can begin playing, and 3 if the media can play from beginning to end.
seekable	TimeRanges	The ranges of times that are available for seeking.
seeking	Boolean	Indicates that the player is moving to a new position in the media file.
src	String	The media file source. This can be rewritten at any time.
start	Float	Gets or sets the location in the media file, in seconds, where playing should begin.
totalBytes	Integer	The total number of bytes needed for the resource (if known).
videoHeight	Integer	Returns the height of the video (not necessarily of the element). Only for <video>.
videoWidth	Integer	Returns the width of the video (not necessarily of the element). Only for <video>.
volume	Float	Gets or sets the current volume as a value between 0.0 and 1.0.

Many of these properties can also be specified as attributes on either the `<audio>` or the `<video>` elements.

Events

In addition to the numerous properties, there are also numerous events that fire on these media elements. The events monitor all of the different properties that change because of media playback and user interaction with the player. These events are listed in the following table.

EVENT NAME	FIRE WHEN
abort	Downloading has been aborted.
canplay	Playback can begin; <code>readyState</code> is 2.
canplaythrough	Playback can proceed and should be uninterrupted; <code>readyState</code> is 3.
canshowcurrentframe	The current frame has been downloaded; <code>readyState</code> is 1.
dataunavailable	Playback can't happen because there's no data; <code>readyState</code> is 0.
durationchange	The <code>duration</code> property value has changed.
emptied	The network connection has been closed.
empty	An error occurs that prevents the media download.
ended	The media has played completely through and is stopped.
error	A network error occurred during download.
load	All of the media has been loaded. This event is considered deprecated; use <code>canplaythrough</code> instead.
loadeddata	The first frame for the media has been loaded.
loadedmetadata	The meta data for the media has been loaded.
loadstart	Downloading has begun.
pause	Playback has been paused.
play	The media has been requested to start playing.
playing	The media has actually started playing.
progress	Downloading is in progress.
ratechange	The speed at which the media is playing has changed.
seeked	Seeking has ended.
seeking	Playback is being moved to a new position.

continues

(continued)

EVENT NAME	FIRE WHEN
stalled	The browser is trying to download, but no data is being received.
timeupdate	The currentTime is updated in an irregular or unexpected way.
volumechange	The volume property value or muted property value has changed.
waiting	Playback is paused to download more data.

These events are designed to be as specific as possible to enable web developers to create custom audio/video players using little more than HTML and JavaScript (as opposed to creating a new Flash movie).

Custom Media Players

You can manually control the playback of a media file, using the `play()` and `pause()` methods that are available on both `<audio>` and `<video>`. Combining the properties, events, and these methods makes it easy to create a custom media player, as shown in this example:



```
<div class="mediaplayer">
    <div class="video">
        <video id="player" src="movie.mov" poster="mymovie.jpg"
               width="300" height="200">
            Video player not available.
        </video>
    </div>
    <div class="controls">
        <input type="button" value="Play" id="video-btn">
        <span id="curtime">0</span>/<span id="duration">0</span>
    </div>
</div>
```

[VideoPlayerExample01.htm](#)

This basic HTML can then be brought to life by using JavaScript to create a simple video player, as shown here:

```
//get references to the elements
var player = document.getElementById("player"),
    btn = document.getElementById("video-btn"),
    curtime = document.getElementById("curtime"),
    duration = document.getElementById("duration");

//update the duration
duration.innerHTML = player.duration;

//attach event handler to button
```

```

EventUtil.addHandler(btn, "click", function(event) {
    if (player.paused){
        player.play();
        btn.value = "Pause";
    } else {
        player.pause();
        btn.value = "Play";
    }
});

//update the current time periodically
setInterval(function(){
    curtime.innerHTML = player.currentTime;
}, 250);

```

[VideoPlayerExample01.htm](#)

The JavaScript code here simply attaches an event handler to the button that either pauses or plays the video, depending on its current state. Then, an event handler is set for the `<video>` element's `load` event so that the duration can be displayed. Last, a repeating timer is set to update the current time display. You can extend the behavior of this custom video player by listening for more events and making use of more properties. The exact same code can also be used with the `<audio>` element to create a custom audio player.

Codec Support Detection

As mentioned previously, not all browsers support all codecs for `<video>` and `<audio>`, which frequently means you must provide more than one media source. There is also a JavaScript API for determining if a given format and codec is supported by the browser. Both media elements have a method called `canPlayType()`, which accepts a format/codec string and returns a string value of "probably", "maybe", or "" (empty string). The empty string is a falsy value, which means you can still use `canPlayType()` in an `if` statement like this:

```

if (audio.canPlayType("audio/mpeg")){
    //do something
}

```

Both "probably" and "maybe" are truthy values and so get coerced to `true` within the context of an `if` statement.

When just a MIME type is provided to `canPlayType()`, the most likely return values are "maybe" and the empty string. This is because a file is really just a container for audio or video data; it is the encoding that really determines if the file can be played. When both a MIME type and a codec are specified, you increase the likelihood of getting "probably" as the return value. Some examples:

```

var audio = document.getElementById("audio-player");

//most likely "maybe"

```

```

if (audio.canPlayType("audio/mpeg")) {
    //do something
}

//could be "probably"
if (audio.canPlayType("audio/ogg; codecs='vorbis'")){
    //do something
}

```

Note that the codecs list must always be enclosed in quotes to work properly. The following is a list of known supported audio formats and codecs:

NAME	STRING	SUPPORTING BROWSERS
AAC	audio/mp4; codecs="mp4a.40.2"	Internet Explorer 9+, Safari 4+, Safari for iOS
MP3	audio/mpeg	Internet Explorer 9+, Chrome
Vorbis	audio/ogg; codecs="vorbis"	Firefox 3.5+, Chrome, Opera 10.5+
WAV	audio/wav; codecs="1"	Firefox 3.5+, Opera 10.5+, Chrome

You can also detect video formats using `canPlayType()` on any video element. The following is a list of known supported video formats and codecs:

NAME	STRING	SUPPORTING BROWSERS
H.264	video/mp4; codecs="avc1.42E01E, mp4a.40.2"	Internet Explorer 9+, Safari 4+, Safari for iOS, WebKit for Android
Theora	video/ogg; codecs="theora"	Firefox 3.5+, Opera 10.5, Chrome
WebM	video/webm; codecs="vp8, vorbis"	Firefox 4+, Opera 10.6, Chrome

The Audio Type

The `<audio>` element also has a native JavaScript constructor called `Audio` to allow the playing of audio at any point in time. The `Audio` type is similar to `Image` in that it is the equivalent of a DOM element but doesn't require insertion into the document to work. Just create a new instance and pass in the audio source file:

```

var audio = new Audio("sound.mp3");
EventUtil.addHandler(audio, "canplaythrough", function(event) {
    audio.play();
});

```

Creating a new instance of `Audio` begins the process of downloading the specified file. Once it's ready, you can call `play()` to start playing the audio.

Calling the `play()` method on iOS causes a dialog to pop up asking for the user's permission to play the sound. In order to play one sound after another, you must call `play()` immediately within the `onfinish` event handler.

HISTORY STATE MANAGEMENT

One of the most difficult aspects of modern web application programming is history management. Gone are the days where every action takes a user to a completely new page, which also means that the Back and Forward buttons have been taken away from users as a familiar way to say “get me to a different state.” The first step to solving that problem was the `hashchange` event (discussed in Chapter 13). HTML5 updates the `history` object to provide easy state management.

Where the `hashchange` event simply let you know when the URL hash changed and expected you to act accordingly, the state management API actually lets you change the browser URL without loading a new page. To do so, use the `history.pushState()` method. This method accepts three arguments: a `data` object, the title of the new state, and an optional relative URL. For example:

```
history.pushState({name:"Nicholas"}, "Nicholas' page", "nicholas.html");
```

As soon as `pushState()` executes, the state information is pushed onto the history stack and the browser’s address bar changes to reflect the new relative URL. Despite this change, the browser does not make a request to the server, even though querying `location.href` will return exactly what’s in the address bar. The second argument isn’t currently used by any implementations and so it is safe to either leave it as an empty string or provide a short title. The first argument should contain all of the information necessary to correctly initialize this page state when necessary.

Since `pushState()` creates a new history entry, you’ll notice that the Back button is enabled. When the Back button is pressed, the `popstate` event fires on the `window` object. The event object for `popstate` has a property called `state`, which contains the object that was passed into `pushState()` as the first argument:

```
EventUtil.addHandler(window, "popstate", function(event) {
    var state = event.state;
    if (state){ //state is null when at first page load
        processState(state);
    }
});
```

Using this state, you must then reset the page into the state represented by the data in the `state` object (as the browser doesn’t do this automatically for you). Keep in mind that when a page is first loaded, there is no state, so hitting the Back button until you get to the original page state will result in `event.state` being `null`.

You can update the current state information by using `replaceState()` and passing in the same first two arguments as `pushState()`. Doing so does not create a new entry in history, it just overwrites the current state:

```
history.replaceState({name:"Greg"}, "Greg's page");
```

HTML5 history state management is supported in Firefox 4+, Safari 5+, Opera 11.5+, and Chrome. In Safari and Chrome, the `state` object passed into `pushState()` or `replaceState()` must not contain any DOM elements. Firefox properly supports putting DOM elements in the `state` object. Opera also supports the `history.state` property, which returns the `state` object for the current state.



When using HTML5 history state management, make sure that any “fake” URL you create using `pushState()` is backed up by a real, physical URL on the web server. Otherwise, hitting the Refresh button will result in a 404.

SUMMARY

HTML5, in addition to defining new markup rules, also defines several JavaScript APIs. These APIs are designed to enable better web interfaces that can rival the capabilities of desktop applications. The APIs covered in this chapter are as follows:

- Cross-document messaging provides the ability to send messages across documents from different origins while keeping the security of the same-origin policy intact.
- Native drag and drop allows you to easily indicate that an element is draggable and respond as the operating system does to drops. You can create custom draggable elements and drop targets.
- The new media elements `<audio>` and `<video>` have their own APIs for interacting with the audio and video. Not all media formats are supported by all browsers, so make use of the `canPlayType()` method to properly detect browser support.
- History state management allows you to change the browser history stack without unloading the current page. This allows the user of the Back and Forward buttons to move between page states that are handled purely by JavaScript.

17

Error Handling and Debugging

WHAT'S IN THIS CHAPTER?

- ▶ Understanding browser error reporting
- ▶ Handling errors
- ▶ Debugging JavaScript code

JavaScript has traditionally been known as one of the most difficult programming languages to debug because of its dynamic nature and years without proper development tools. Errors typically resulted in confusing browser messages such as "object expected" that provided little or no contextual information. The third edition of ECMAScript aimed to improve this situation, introducing the `try-catch` and `throw` statements, along with various error types to help developers deal with errors when they occur. A few years later, JavaScript debuggers and debugging tools began appearing for web browsers. By 2008, most web browsers supported some JavaScript debugging capabilities.

Armed with the proper language support and development tools, web developers are now empowered to implement proper error-handling processes and figure out the cause of problems.

BROWSER ERROR REPORTING

All of the major web browsers — Internet Explorer, Firefox, Safari, Chrome, and Opera — have some way to report JavaScript errors to the user. By default, all browsers hide this information, because it's of little use to anyone but the developer. When developing browser-based JavaScript solutions, be sure to enable JavaScript error reporting to be notified when there is an error.

Internet Explorer

Internet Explorer is the only browser that displays a JavaScript error indicator in the browser's chrome by default. When a JavaScript error occurs, a small yellow icon appears in the lower-left corner of the browser next to the text "Error on page". The icon is easy to miss if you're not expecting it. When you double-click the icon, a dialog box is displayed containing the error message and allowing you to see other related information, such as the line number, character number, error code, and filename (which is always the URL you are viewing) (see Figure 17-1).



FIGURE 17-1

This default behavior is fine for general users but is insufficient for web development. The settings can be changed such that an error dialog is displayed every time there is an error. To make this change, click on the Tools menu, then Internet Options. When the dialog box appears, click the Advanced tab and check the box next to "Display a notification about every script error" (see Figure 17-2). Click OK to save this setting.

After updating this setting, the dialog box that is typically displayed when double-clicking the yellow icon is shown by default whenever an error occurs.

If script debugging is enabled (it is disabled by default), and you have the browser set to always display a notification about errors, then you may see an alternate dialog box that asks if you'd like to debug the error (see Figure 17-3).

To enable script debugging, you must first have a script debugger installed that is compatible with Internet Explorer. (Versions 8 and 9 come bundled with a debugger.) Debuggers are discussed later in this chapter.

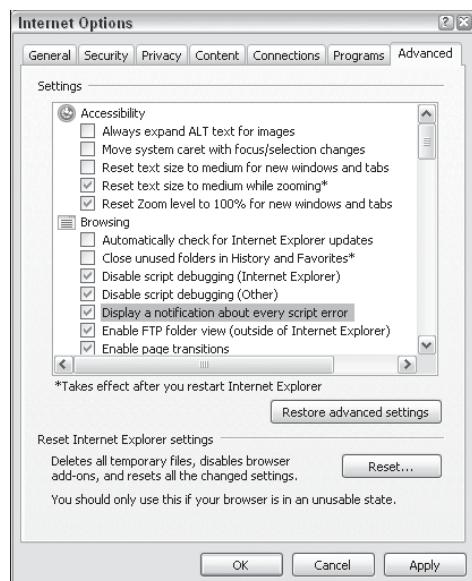


FIGURE 17-2



FIGURE 17-3



In Internet Explorer 7 and earlier, the line number in the error message is typically off by one when the error was caused by a script in an external file. Errors caused by inline scripts have an accurate line number. Internet Explorer 8 and later fixed this issue so all line numbers are accurate.

Firefox

By default, Firefox makes no changes to the user interface when a JavaScript error occurs. Instead, it silently logs the error to the error console. Click on the Tools menu then Error Console to display the error console (see Figure 17-4). Be aware that the error console also contains warnings and information about JavaScript, CSS, and HTML, so it may be useful to filter the results.

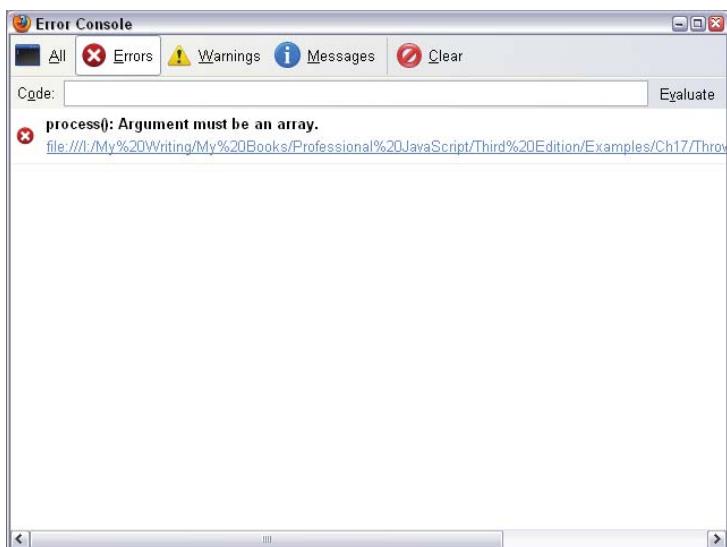
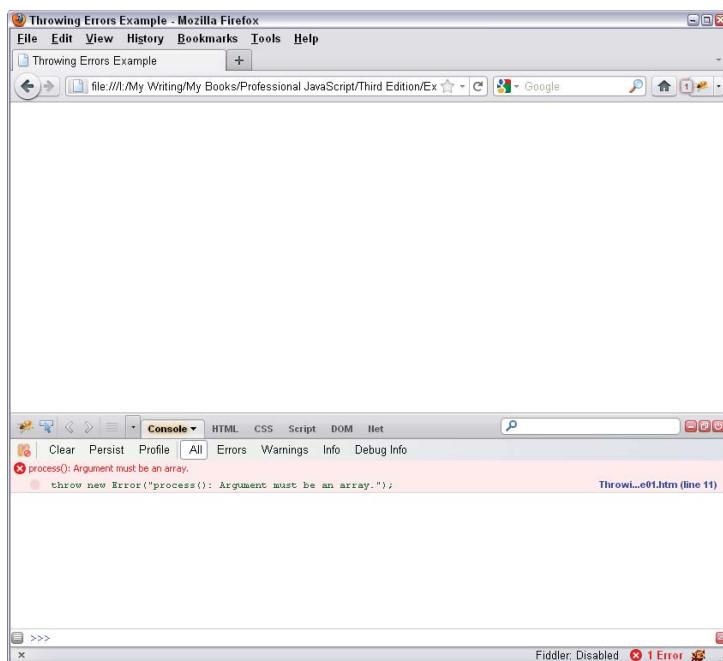


FIGURE 17-4

When a JavaScript error occurs, it gets logged as an error with an error message, the URL on which the error occurred, and the line number. Clicking the filename opens a read-only view of the script that caused the error with the offending line highlighted.

Firebug, arguably the most popular browser add-on for web developers, augments the default Firefox JavaScript error behavior. Firebug, available at www.getfirebug.com, adds an area in the bottom-right Firefox status bar for JavaScript information. By default, a green check mark icon is displayed in this location. The icon changes to a red X when a JavaScript error occurs and displays the number of errors. Clicking the red X opens the Firebug console, which displays the error message, the line of code that caused the error (out of context), the URL where the error occurred, and the line number (see Figure 17-5).

**FIGURE 17-5**

When the line that caused the error is clicked in Firebug, it opens a new Firebug view with the line highlighted in the context of the entire script file.



Firebug has many more uses beyond displaying error messages. It is a full-featured debugging environment for Firefox, providing ways to debug JavaScript, CSS, the DOM, and network information.

Safari

Safari on both Windows and Mac OS hide all JavaScript error information by default. In order to get access to this information, you must enable the Develop menu. To do so, choose Edit → Preferences and then click the Advanced tab. There is a check box titled “Show develop menu in menu bar” that should be checked. Once the setting is enabled, a menu named “Develop” appears in the Safari menu bar (see Figure 17-6).

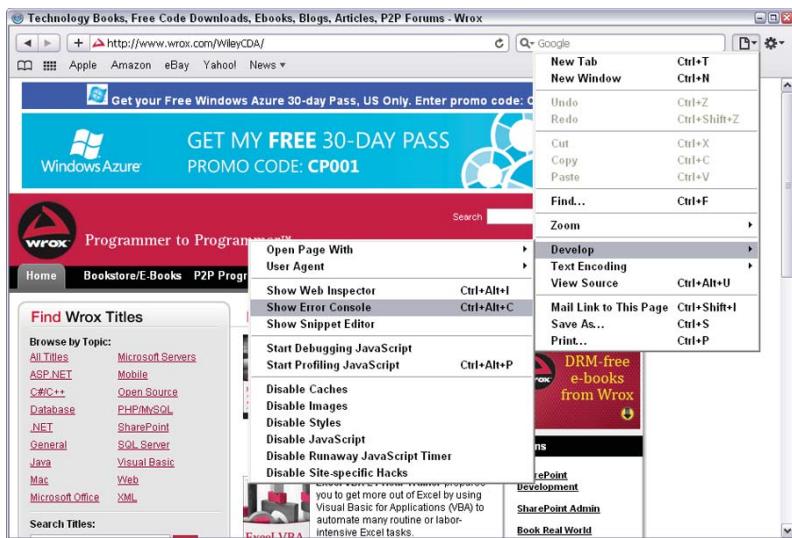


FIGURE 17-6

The Develop menu provides several options for debugging and otherwise working with the page that is currently loaded. You can click Show Error Console to display a list of JavaScript and other errors. The console displays the error message, the URL of the error, and the line number for the error (see Figure 17-7).

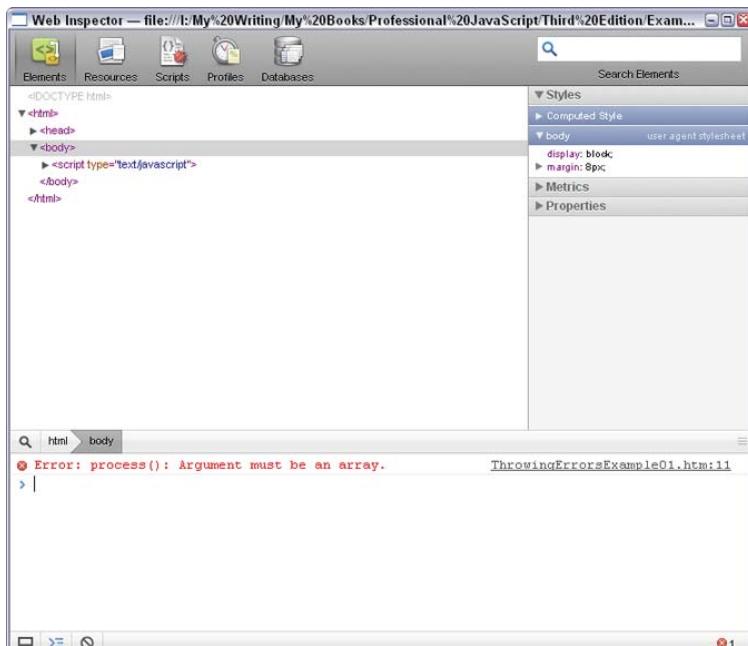


FIGURE 17-7

When you click on the error message, you are taken to the source code that caused the error. Other than outputting to the console, JavaScript errors cause no change in the Safari window.

Opera

Opera also hides JavaScript errors by default. All errors are logged to the error console, which can be displayed by selecting the Page menu, then Page \Rightarrow Developer Tools \Rightarrow Error Console. As with Firefox, the Opera error console contains information about not only JavaScript errors but also errors or warnings for HTML, CSS, XML, XSLT, and a number of other sources. You can filter on the type of messages you want to see by using the drop-down boxes in the lower-left corner (see Figure 17-8).

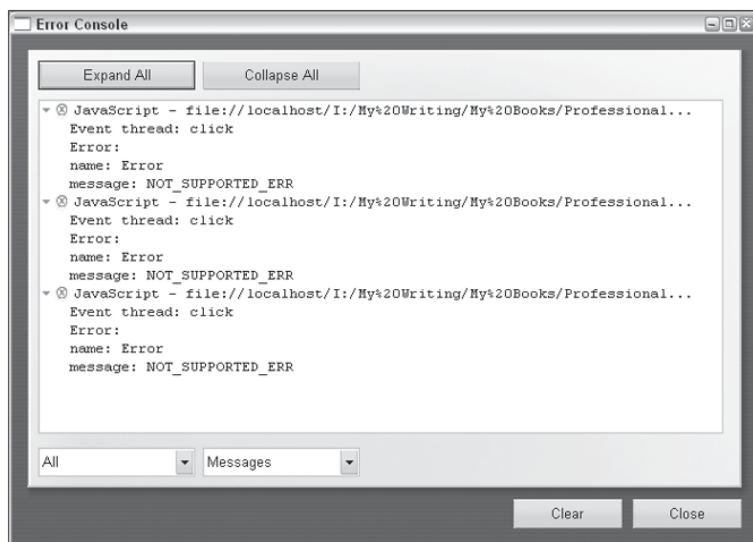


FIGURE 17-8

The error messages appear with information about the URL that caused the error and the thread in which the error occurred. In some cases, a stack trace is also provided. There is no way to get additional data about the error other than the details displayed in the error console.

It's possible to have the error console pop up whenever a JavaScript error occurs. To do so, go to the Settings menu and click Preferences. Click the Advanced tab, and then select Content from the left menu. Click the JavaScript Options button to bring up the JavaScript Options dialog box (shown in Figure 17-9).

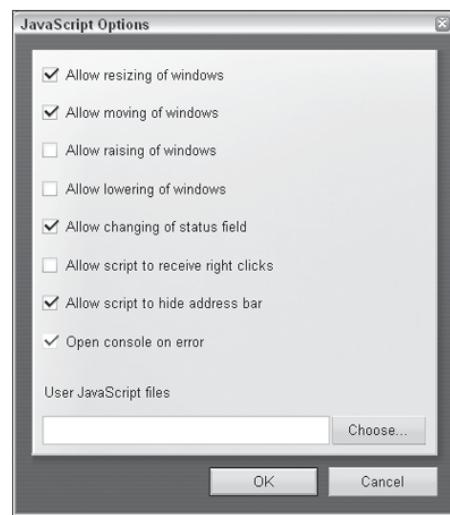


FIGURE 17-9

Ensure that the check box next to “Open console on error” is checked, and then click OK. The error console will now pop up any time there is a JavaScript error. This can also be done on a per-site basis by choosing Tools ⇔ Quick Preferences ⇔ Edit Site Preferences, selecting the Scripting tab, and checking the “Open console on error” check box.

Chrome

As with Safari and Opera, Chrome hides JavaScript errors. All errors are logged to the Web Inspector console. In order to access this information, you must manually open the Web Inspector. To do so, click the “Control this page” button to the right of the address bar, and select Developer ⇔ JavaScript console (see Figure 17-10).

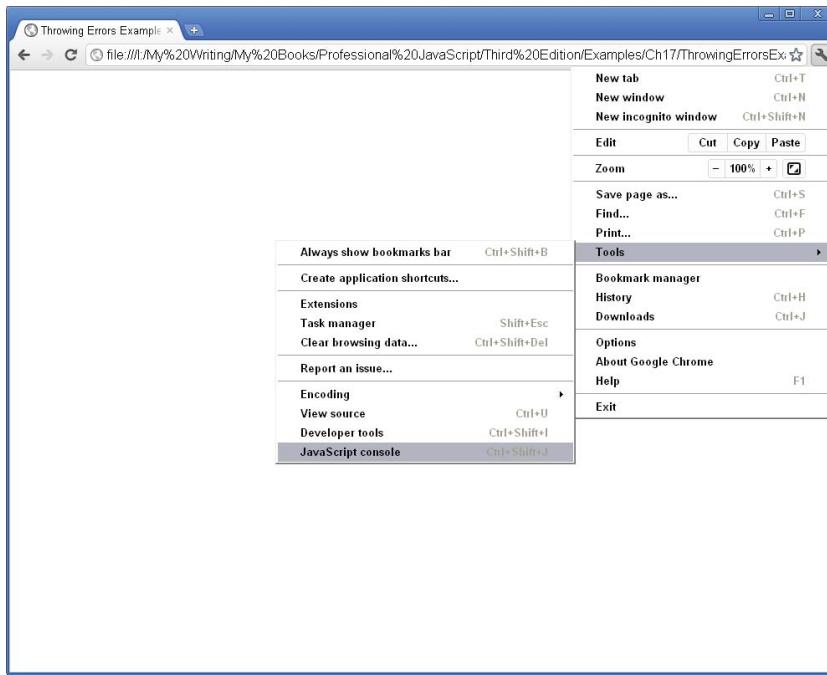


FIGURE 17-10

The Web Inspector contains information about the page and the JavaScript console. Errors are displayed in the console with the error message, the URL of the error, and the line number for the error (see Figure 17-11).

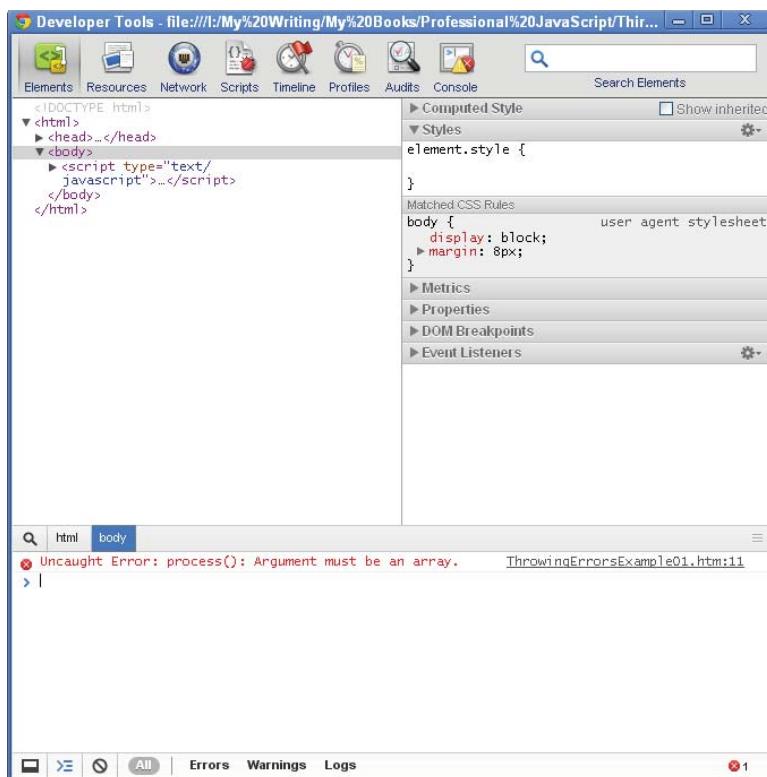


FIGURE 17-11

Clicking on the error in the JavaScript console takes you to the source code in the file that caused the error.

ERROR HANDLING

No one doubts the importance of error handling in programming. Every major web application needs a good error-handling protocol and most good ones do, though it is typically on the server side of the application. In fact, great care is usually taken by the server-side team to define an error-logging mechanism that categorizes errors by type, frequency, and any other metric that may be important. The result is the ability to understand how the application is working in the public with a simple database query or report-generating script.

Error handling has slowly been adopted on the browser side of web applications even though it is just as important. An important fact to understand is that most people who use the Web are not technically savvy — most don't even fully comprehend what a web browser is, let alone which one they're using. As described earlier in this chapter, each browser behaves a little bit differently when a JavaScript error occurs. From a small icon appearing in the corner of a browser to absolutely nothing happening, the default browser experience for JavaScript errors is horrible for the end

user. In the best case, the user has no idea what happened and will try again; in the worst case, the user gets incredibly annoyed and never comes back. Having a good error-handling strategy keeps your users informed about what is going on without scaring them. To accomplish this, you must understand the various ways that you can trap and deal with JavaScript errors as they occur.

The try-catch Statement

ECMA-262, third edition, introduced the `try-catch` statement as a way to handle exceptions in JavaScript. The basic syntax is as follows, which is the same as the `try-catch` statement in Java:

```
try {
    //code that may cause an error
} catch (error) {
    //what to do when an error occurs
}
```

Any code that might possibly throw an error should be placed in the `try` portion of the statement, and the code to handle the error is placed in the `catch` portion, as shown in the following example:

```
try {
    window.someNonexistentFunction();
} catch (error){
    alert("An error happened!");
}
```

If an error occurs at any point in the `try` portion of the statement, code execution immediately exits and resumes in the `catch` portion. The `catch` portion of the statement receives an object containing information about the error that occurred. Unlike other languages, you must define a name for the error object even if you don't intend to use it. The exact information available on this object varies from browser to browser but contains, at a minimum, a `message` property that holds the error message. ECMA-262 also specifies a `name` property that defines the type of error; this property is available in all current browsers. You can, therefore, display the actual browser message if necessary, as shown in the following example:



Available for
download on
Wrox.com

```
try {
    window.someNonexistentFunction();
} catch (error){
    alert(error.message);
}
```

[TryCatchExample01.htm](#)

This example uses the `message` property when displaying an error message to the user. The `message` property is the only one that is guaranteed to be there across Internet Explorer, Firefox, Safari, Chrome, and Opera, even though each browser adds other information. Internet Explorer adds a `description` property that is always equal to the `message`, as well as a `number` property that gives an internal error number. Firefox adds `fileName`, `lineNumber`, and `stack` (which contains a stack trace). Safari adds `line` (for the line number), `sourceID` (an internal error code), and `sourceURL`. Once again, it is best to rely only on the `message` property for cross-browser compatibility.

The finally Clause

The optional `finally` clause of the `try`-`catch` statement always runs its code no matter what. If the code in the `try` portion runs completely, the `finally` clause executes; if there is an error and the `catch` portion executes, the `finally` portion still executes. There is literally nothing that can be done in the `try` or `catch` portion of the statement to prevent the code in `finally` from executing, which includes using a `return` statement. Consider the following function:



Available for download on Wrox.com

```
function testFinally() {
    try {
        return 2;
    } catch (error) {
        return 1;
    } finally {
        return 0;
    }
}
```

TryCatchExample02.htm

This function simply places a `return` statement in each portion of the `try`-`catch` statement. It looks like the function should return 2, since that is in the `try` portion and wouldn't cause an error. However, the presence of the `finally` clause causes that `return` to be ignored; the function returns 0 when called no matter what. If the `finally` clause were removed, the function would return 2.

If `finally` is provided, then `catch` becomes optional (only one or the other is required). Internet Explorer 7 and earlier had a bug where the code in `finally` would never be executed unless there was also a `catch`. If you are still dealing with earlier versions of Internet Explorer, then the workaround is to also provide a `catch` clause even if it is empty. This was fixed in version 8.



It's very important to understand that any `return` statements in either the `try` or the `catch` portion will be ignored if a `finally` clause is also included in your code. Be sure to double-check the intended behavior of your code when using `finally`.

Error Types

There are several different types of errors that can occur during the course of code execution. Each error type has a corresponding object type that is thrown when an error occurs. ECMA-262 defines the following seven error types:

- `Error`
- `Evaluator`
- `RangeError`
- `ReferenceError`

- `SyntaxError`
- `TypeError`
- `URIError`

The `Error` type is the base type from which all other error types inherit. As a result of this, all error types share the same properties (the only methods on error objects are the default object methods). An error of type `Error` is rarely, if ever, thrown by a browser; it is provided mainly for developers to throw custom errors.

The `EvalError` type is thrown when an exception occurs while using the `eval()` function. ECMA-262 states that this error is thrown “if value of the `eval` property is used in any way other than a direct call (that is, other than by the explicit use of its name as an `Identifier`, which is the `MemberExpression` in a `CallExpression`), or if the `eval` property is assigned to.” This basically means using `eval()` as anything other than a function call, such as:

```
new eval();      //throws EvalError
eval = foo;     //throws EvalError
```

In practice, browsers don’t always throw `EvalError` when they’re supposed to. For example, Firefox 4+ and Internet Explorer 8 throw an `TypeError` in the first case but the second succeeds without error. Because of this and the unlikelihood of these patterns being used, it is highly unlikely that you will run into this error type.

A `RangeError` occurs when a number is outside the bounds of its range. For example, this error may occur when an attempt is made to define an array with an unsupported number of items, such as `-20` or `Number.MAX_VALUE`, as shown here:

```
var items1 = new Array(-20);           //throws RangeError
var items2 = new Array(Number.MAX_VALUE); //throws RangeError
```

Range errors occur infrequently in JavaScript.

The `ReferenceError` type is used when an object is expected. (This is literally the cause of the famous “object expected” browser error.) This type of error typically occurs when attempting to access a variable that doesn’t exist, as in this example:

```
var obj = x;      //throws ReferenceError when x isn't declared
```

A `SyntaxError` object is thrown most often when there is a syntax error in a JavaScript string that is passed to `eval()`, as in this example:

```
eval("a ++ b");    //throws SyntaxError
```

Outside of using `eval()`, the `SyntaxError` type is rarely used, because syntax errors occurring in JavaScript code stop execution immediately.

The `TypeError` type is the most used in JavaScript and occurs when a variable is of an unexpected type or an attempt is made to access a nonexistent method. This can occur for any number of

reasons, most often when a type-specific operation is used with a variable of the wrong type. Here are some examples:

```
var o = new 10;           //throws TypeError
alert("name" in true);   //throws TypeError
Function.prototype.toString.call("name"); //throws TypeError
```

Type errors occur most frequently with function arguments that are used without their type being verified first.

The last type of error is `URIError`, which occurs only when using the `encodeURI()` or `decodeURI()` with a malformed URI. This error is perhaps the most infrequently observed in JavaScript, because these functions are incredibly robust.

The different error types can be used to provide more information about an exception, allowing appropriate error handling. You can determine the type of error thrown in the `catch` portion of a `try-catch` statement by using the `instanceof` operator, as shown here:

```
try {
    someFunction();
} catch (error){
    if (error instanceof TypeError){
        //handle type error
    } else if (error instanceof ReferenceError){
        //handle reference error
    } else {
        //handle all other error types
    }
}
```

Checking the error type is the easiest way to determine the appropriate course of action in a cross-browser way, since the error message contained in the `message` property differs from browser to browser.

Usage of try-catch

When an error occurs within a `try-catch` statement, the browser considers the error to have been handled, and so it won't report it using the mechanisms discussed earlier in this chapter. This is ideal for web applications with users who aren't technically inclined and wouldn't otherwise understand when an error occurs. The `try-catch` statement allows you to implement your own error-handling mechanism for specific error types.

The `try-catch` statement is best used where an error might occur that is out of your control. For example, if you are using a function that is part of a larger JavaScript library, that function may throw errors either purposefully or by mistake. Since you can't modify the library's code, it would be appropriate to surround the call in a `try-catch` statement in case an error does occur and then handle the error appropriately.

It's not appropriate to use a `try-catch` statement if you know an error will occur with your code specifically. For example, if a function will fail when a string is passed in instead of a number, you

should check the data type of the argument and act accordingly; there is no need in this case to use a try-catch statement.

Throwing Errors

A companion to the try-catch statement is the throw operator, which can be used to throw custom errors at any point in time. The throw operator must be used with a value but places no limitation on the type of value. All of the following lines are legal:

```
throw 12345;
throw "Hello world!";
throw true;
throw { name: "JavaScript" };
```

When the throw operator is used, code execution stops immediately and continues only if a try-catch statement catches the value that was thrown.

Browser errors can be more accurately simulated by using one of the built-in error types. Each error type's constructor accepts a single argument, which is the exact error message. Here is an example:

```
throw new Error("Something bad happened.');
```

This code throws a generic error with a custom error message. The error is handled by the browser as if it were generated by the browser itself, meaning that it is reported by the browser in the usual way and your custom error message is displayed. You can achieve the same result using the other error types, as shown in these examples:

```
throw new SyntaxError("I don't like your syntax.");
throw new TypeError("What type of variable do you take me for?");
throw new RangeError("Sorry, you just don't have the range.");
throw new EvalError("That doesn't evaluate.");
throw new URIError("Uri, is that you?");
throw new ReferenceError("You didn't cite your references properly.");
```

The most often used error types for custom error messages are `Error`, `RangeError`, `ReferenceError`, and `TypeError`.

You can also create custom error types by inheriting from `Error` using prototype chaining (discussed in Chapter 6). You should provide both a `name` property and a `message` property on your error type. Here is an example:



```
function CustomError(message) {
    this.name = "CustomError";
    this.message = message;
}

CustomError.prototype = new Error();

throw new CustomError("My message");
```

[ThrowingErrorsExample01.htm](#)

Custom error types that are inherited from `Error` are treated just like any other error by the browser. Creating custom error types is helpful when you will be catching the errors that you throw and need to decipher them from browser-generated errors.



Internet Explorer displays custom error messages only when throwing `Error` objects. For all other types, it simply displays "exception thrown and not caught".

When to Throw Errors

Throwing custom errors is a great way to provide more information about why a function has failed. Errors should be thrown when a particular known error condition exists that won't allow the function to execute properly. That is, the browser will throw an error while executing this function given a certain condition. For example, the following function will fail if the argument is not an array:



Available for
download on
Wrox.com

```
function process(values) {
    values.sort();

    for (var i=0, len=values.length; i < len; i++) {
        if (values[i] > 100){
            return values[i];
        }
    }

    return -1;
}
```

[ThrowingErrorsExample02.htm](#)

If this function is run with a string as the argument, the call to `sort()` fails. Each browser gives a different, though somewhat obtuse, error message, as listed here:

- **Internet Explorer** — Property or method doesn't exist.
- **Firefox** — `values.sort()` is not a function.
- **Safari** — Value undefined (result of expression `values.sort`) is not an object.
- **Chrome** — Object name has no method 'sort'.
- **Opera** — Type mismatch (usually a non-object value used where an object is required).

Although Firefox, Chrome, and Safari at least indicate the part of the code that caused the error, none of the error messages are particularly clear as to what happened or how it could be fixed.

When dealing with one function, as in the preceding example, debugging is easy enough to handle with these error messages. However, when you're working on a complex web application with thousands of lines of JavaScript code, finding the source of the error becomes much more difficult.

This is where a custom error with appropriate information will significantly contribute to the maintainability of the code. Consider the following example:



Available for
download on
Wrox.com

```
function process(values) {
    if (!(values instanceof Array)){
        throw new Error("process(): Argument must be an array.");
    }

    values.sort();

    for (var i=0, len=values.length; i < len; i++){
        if (values[i] > 100){
            return values[i];
        }
    }

    return -1;
}
```

[ThrowingErrorsExample02.htm](#)

In this rewritten version of the function, an error is thrown if the `values` argument isn't an array. The error message provides the name of the function and a clear description as to why the error occurred. If this error occurred in a complex web application, you would have a much clearer idea of where the real problem is.

When you're developing JavaScript code, take a critical eye toward each function and the circumstances under which it may fail. A good error-handling protocol ensures that the only errors that occur are the ones that you throw.



Using `instanceof` to identify arrays has some issues when used in a cross-frame environment. See Safe Type Detection in Chapter 22 for more details.

Throwing Errors versus try-catch

A common question that arises is when to throw errors versus when to use try-catch to capture them. Generally speaking, errors are thrown in the low levels of an application architecture, at a level where not much is known about the ongoing process, and so the error can't really be handled. If you are writing a JavaScript library that may be used in a number of different applications, or even a utility function that will be used in a number of different places in a single application, you should strongly consider throwing errors with detailed information. It is then up to the application to catch the errors and handle them appropriately.

The best way to think about the difference between throwing errors and catching errors is this: you should catch errors only if you know exactly what to do next. The purpose of catching an error is

to prevent the browser from responding in its default manner; the purpose of throwing an error is to provide information about why an error occurred.

The error Event

Any error that is not handled by a `try-catch` causes the `error` event to fire on the `window` object. This event was one of the first supported by web browsers, and its format has remained intact for backwards compatibility in Internet Explorer, Firefox, and Chrome. (The `error` event is not supported in Opera or Safari.) An `onerror` event handler doesn't create an `event` object in any browser, instead, it receives three arguments: the error message, the URL on which the error occurred, and the line number. In most cases, only the error message is relevant since the URL is the same as the location of the document, and the line number could be for inline JavaScript or code in external files. The `onerror` event handler needs to be assigned using the DOM Level 0 technique shown here, since it doesn't follow the DOM Level 2 Events standard format:

```
window.onerror = function(message, url, line){
    alert(message);
};
```

When any error occurs, whether browser-generated or not, the `error` event fires, and this event handler executes. Then, the default browser behavior takes over, displaying the error message as it would normally. You can prevent the default browser error reporting by returning `false`, as shown here:



```
window.onerror = function(message, url, line){
    alert(message);
    return false;
};
```

OnErrorExample01.htm

By returning `false`, this function effectively becomes a `try-catch` statement for the entire document, capturing all unhandled runtime errors. This event handler is the last line of defense against errors being reported by the browser and, ideally, should never have to be used. Proper usage of the `try-catch` statement means that no errors reach the browser level and, therefore, should never fire the `error` event.



There is a significant difference between the way browsers handle errors using this event. When the `error` event occurs in Internet Explorer, normal code execution continues; all variables and data are retained and remain accessible from within the `onerror` event handler. In Firefox, however, normal code execution ends, and all variables and data that existed prior to the error occurring are destroyed, making it difficult to truly evaluate the error.

Images also support an `error` event. Any time the URL in an image's `src` attribute doesn't return a recognized image format, the `error` event fires. This event follows the DOM format by returning an `event` object with the image as the target. Here is an example:



```
var image = new Image();
EventUtil.addHandler(image, "load", function(event) {
    alert("Image loaded!");
});
EventUtil.addHandler(image, "error", function(event) {
    alert("Image not loaded!");
});
image.src = "smilex.gif"; //doesn't exist
```

[OnErrorExample02.htm](#)

In this example, an `alert` is displayed when the image fails to load. It's important to understand that once the `error` event fires, the image download process is already over and will not be resumed.

Error-Handling Strategies

Error-handling strategies have traditionally been confined to the server for web applications. There's often a lot of thought that goes into errors and error handling, including logging and monitoring systems. The point of such tools is to analyze error patterns in the hopes of tracking down the root cause and understanding how many users the error affects.

It is equally important to have an error-handling strategy for the JavaScript layer of a web application. Since any JavaScript error can cause a web page to become unusable, understanding when and why errors occur is vital. Most web-application users are not technical and can easily get confused when something doesn't work as expected. They may reload the page in an attempt to fix the problem, or they may just stop trying. As the developer, you should have a good understanding of when and how the code could fail and have a system to track such issues.

Identify Where Errors Might Occur

The most important part of error handling is to first identify where errors might occur in the code. Since JavaScript is loosely typed and function arguments aren't verified, there are often errors that become apparent only when the code is executed. In general, there are three error categories to watch for:

- Type coercion errors
- Data type errors
- Communication errors

Each of these errors occurs when using specific patterns or not applying sufficient value checking.

Type Coercion Errors

Type coercion errors occur as the result of using an operator or other language construct that automatically changes the data type of a value. The two most common type coercion errors occur as a result of using the equal (==) or not equal (!=) operator and using a non-Boolean value in a flow control statement, such as `if`, `for`, and `while`.

The equal and not equal operators, discussed in Chapter 3, automatically convert values of different types before performing a comparison. Since the same symbols typically perform straight comparisons in nondynamic languages, developers often mistakenly use them in JavaScript in the same way. In most cases, it's best to use the identically equal (==) and not identically equal (!==) operators to avoid type coercion. Here is an example:

```
alert(5 == "5");           //true
alert(5 === "5");         //false
alert(1 == true);          //true
alert(1 === true);         //false
```

In this code, the number 5 and the string "5" are compared using the equal operator and the identically equal operator. The equal operator first converts the string "5" into the number 5 and then compares it with the other number 5, resulting in `true`. The identically equal operator notes that the two data types are different and simply returns `false`. The same occurs with the values 1 and `true`: they are considered equal by the equal operator but not equal using the identically equal operator. Using the identically equal and not identically equal operators can prevent type coercion errors that occur during comparisons and are highly recommended over using the equal and not equal operators.

The second place that type coercion errors occur is in flow control statements. Statements such as `if` automatically convert any value into a Boolean before determining the next step. The `if` statement, specifically, is often used in error-prone ways. Consider the following example:

```
function concat(str1, str2, str3){
  var result = str1 + str2;
  if (str3){ //avoid!!!
    result += str3;
  }
  return result;
}
```

This function's intended purpose is to concatenate two or three strings and return the result. The third string is an optional argument and so must be checked. As mentioned in Chapter 3, named variables that aren't used are automatically assigned the value of `undefined`. The value `undefined` converts into the Boolean value `false`, so the intent of the `if` statement in this function is to concatenate the third argument only if it is provided. The problem is that `undefined` is not the only value that gets converted to `false`, and a string is not the only value that gets converted to `true`. If the third argument is the number 0, for example, the `if` condition fails, while a value of 1 causes the condition to pass.

Using non-Boolean values as conditions in a flow control statement is a very common cause of errors. To avoid such errors, always make sure that a Boolean value is passed as the condition. This is most often accomplished by doing a comparison of some sort. For example, the previous function can be rewritten as shown here:

```
function concat(str1, str2, str3){
    var result = str1 + str2;
    if (typeof str3 == "string"){ //proper comparison
        result += str3;
    }
    return result;
}
```

In this updated version of the function, the `if` statement condition returns a Boolean value based on a comparison. This function is much safer and is less affected by incorrect values.

Data Type Errors

Since JavaScript is loosely typed, variables and function arguments aren't compared to ensure that the correct type of data is being used. It is up to you, as the developer, to do an appropriate amount of data type checking to ensure that an error will not occur. Data type errors most often occur as a result of unexpected values being passed into a function.

In the previous example, the data type of the third argument is checked to ensure that it's a string, but the other two arguments aren't checked at all. If the function must return a string, then passing in two numbers and omitting the third argument easily breaks it. A similar situation is present in the following function:

```
//unsafe function, any non-string value causes an error
function getQueryString(url){
    var pos = url.indexOf("?");
    if (pos > -1){
        return url.substring(pos +1);
    }
    return "";
}
```

The purpose of this function is to return the query string of a given URL. To do so, it first looks for a question mark in the string using `indexOf()` and, if found, returns everything after the question mark using the `substring()` method. The two methods used in this example are specific to strings, so any other data type that is passed in will cause an error. The following simple type check makes this function less error prone:

```
function getQueryString(url){
    if (typeof url == "string"){ //safer with type check
        var pos = url.indexOf("?");
        if (pos > -1){
            return url.substring(pos +1);
        }
    }
    return "";
}
```

In this rewritten version of the function, the first step is to check that the value passed in is actually a string. This ensures that the function will never cause an error because of a nonstring value.

As discussed in the previous section, using non-Boolean values as conditions for flow control statements is a bad idea because of type coercion. This is also a bad practice that can cause data type errors. Consider the following function:

```
//unsafe function, non-array values cause an error
function reverseSort(values){
    if (values){ //avoid!!!
        values.sort();
        values.reverse();
    }
}
```

The `reverseSort()` function sorts an array in reverse order, using both the `sort()` and the `reverse()` methods. Because of the control condition in the `if` statement, any nonarray value that converts to `true` will cause an error. Another common mistake is to compare the argument against `null`, as in this example:

```
//still unsafe, non-array values cause an error
function reverseSort(values){
    if (values != null){ //avoid!!!
        values.sort();
        values.reverse();
    }
}
```

Comparing a value against `null` only protects the code from two values: `null` and `undefined` (which are equivalent to using the equal and not equal operators). A `null` comparison doesn't do enough to ensure that the value is appropriate; therefore, this technique should be avoided. It's also recommended that you don't compare a value against `undefined`, for the same reason.

Another poor choice is to use feature detection for only one of the features being used. Here is an example:

```
//still unsafe, non-array values cause an error
function reverseSort(values){
    if (typeof values.sort == "function"){ //avoid!!!
        values.sort();
        values.reverse();
    }
}
```

In this example, the code checks for the existence of a `sort()` method on the argument. This leaves open the possibility that an object may be passed in with a `sort()` function that is not an array, in which case the call to `reverse()` causes an error. When you know the exact type of object that is expected, it's best to use `instanceof`, as shown in the following example, to determine that the value is of the right type:

```
//safe, non-array values are ignored
function reverseSort(values){
```

```

        if (values instanceof Array){ //fixed
            values.sort();
            values.reverse();
        }
    }
}

```

This last version of `reverseSort()` is safe — it tests the `values` argument to see if it's an instance of `Array`. In this way, the function is assured that any nonarray values are ignored.

Generally speaking, values that should be primitive types should be checked using `typeof`, and values that should be objects should be checked using `instanceof`. Depending on how a function is being used, it may not be necessary to check the data type of every argument, but any public-facing APIs should definitely perform type checking to ensure proper execution.

Communication Errors

Since the advent of Ajax programming (discussed in Chapter 21), it has become quite common for web applications to dynamically load information or functionality throughout the application's life cycle. Any communication between JavaScript and the server is an opportunity for an error to occur.

The first type of communication error involves malformed URLs or post data. This typically occurs when data isn't encoded using `encodeURIComponent()` before being sent to the server. The following URL, for example, isn't formed correctly:

```
http://www.yourdomain.com/?redir=http://www.someotherdomain.com?a=b&c=d
```

This URL can be fixed by using `encodeURIComponent()` on everything after "redir=", which produces the following result:

```
http://www.example.com/?redir=http%3A%2F%2Fwww.someotherdomain.com%3Fa%3Db%26c%3Dd
```

The `encodeURIComponent()` method should always be used for query string arguments. To ensure that this happens, you will find it's sometimes helpful to define a function that handles query string building, such as the following:

```

function addQueryStringArg(url, name, value) {
    if (url.indexOf("?") == -1){
        url += "?";
    } else {
        url += "&";
    }

    url += encodeURIComponent(name) + "=" + encodeURIComponent(value);
    return url;
}

```

This function accepts three arguments: the URL to append the query string argument to, the name of the argument, and the argument value. If the URL that's passed in doesn't contain a question

mark, then one is added; otherwise, an ampersand is added because this means there are other query string arguments. The query string name and value are then encoded and added to the URL. The function can be used as in the following example:

```
var url = "http://www.somedomain.com";
var newUrl = addQueryStringArg(url, "redir",
                               "http://www.someotherdomain.com?a=b&c=d");
alert(newUrl);
```

Using this function instead of manually building URLs can ensure proper encoding and avoid errors related to it.

Communication errors also occur when the server response is not as expected. When using dynamic script loading or dynamic style loading, as discussed in Chapter 10, there is the possibility that the requested resource is not available. Firefox, Chrome, and Safari fail silently when a resource isn't returned, whereas Internet Explorer and Opera both error out. Unfortunately, there is little you can do when using these techniques to determine that an error has occurred. In some cases, using Ajax communication can provide additional information about error conditions.



Communication errors can also occur when using Ajax communication. Issues and errors surrounding Ajax are discussed in Chapter 21.

Distinguishing between Fatal and Nonfatal Errors

One of the most important parts of any error-handling strategy is to determine whether or not an error is fatal. One or more of the following identifies a nonfatal error:

- It won't interfere with the user's main tasks.
- It affects only a portion of the page.
- Recovery is possible.
- Repeating the action may result in success.

In essence, nonfatal errors aren't a cause for concern. For example, Yahoo! Mail (<http://mail.yahoo.com>) has a feature that allows users to send SMS messages from the interface. If for some reason SMS messages don't work, it's a nonfatal error, because that is not the application's primary function. The primary use case for Yahoo! Mail is to read and write e-mail messages, and as long as the user can do that, there is no reason to interrupt the user experience. Nonfatal errors don't require you to send an explicit message to the user — you may be able to replace the area of the page that is affected with a message indicating that the functionality isn't available, but it's not necessary to interrupt the user.

Fatal errors, on the other hand, are identified by one or more of the following:

- The application absolutely cannot continue.
- The error significantly interferes with the user's primary objective.
- Other errors will occur as a result.

It's vitally important to understand when a fatal error occurs in JavaScript so appropriate action can be taken. When a fatal error occurs, you should send a message to the users immediately to let them know that they will not be able to continue what they were doing. If the page must be reloaded for the application to work, then you should tell the user this and provide a button that automatically reloads the page.

You must also make sure that your code doesn't dictate what is and is not a fatal error. Nonfatal and fatal errors are primarily indicated by their affect on the user. Good code design means that an error in one part of the application shouldn't unnecessarily affect another part that, in reality, isn't related at all. For example, consider a personalized home page, such as My Yahoo! (<http://my.yahoo.com>), that has multiple independent modules on the page. If each module has to be initialized using a JavaScript call, you may see code that looks something like this:

```
for (var i=0, len=mods.length; i < len; i++) {
    mods[i].init();      //possible fatal error
}
```

On its surface, this code appears fine: the `init()` method is called on each module. The problem is that an error in any module's `init()` method will cause all modules that come after it in the array to never be initialized. If the error occurs on the first module, then none of the modules on the page will be initialized. Logically, this doesn't make sense, because each module is an independent entity that isn't reliant on any other module for its functionality. It's the structure of the code that makes this type of error fatal. Fortunately, the code can be rewritten as follows to make an error in any one module nonfatal:

```
for (var i=0, len=mods.length; i < len; i++) {
    try {
        mods[i].init();
    } catch (ex) {
        //handle error here
    }
}
```

By adding a `try-catch` statement into the `for` loop, any error when a module initializes will not prevent other modules from initializing. When an error occurs in this code, it can be handled independently and in a way that doesn't interfere with the user experience.

Log Errors to the Server

A common practice in web applications is to have a centralized error log where important errors are written for tracking purposes. Database and server errors are regularly written to the log and categorized through some common API. With complex web applications, it's recommended that you also log JavaScript errors back to the server. The idea is to log the errors into the same system used for server-side errors and categorize them as having come from the front end. Using the same system allows for the same analytics to be performed on the data regardless of the error's source.

To set up a JavaScript error-logging system, you'll first need a page or entry point on the server that can handle the error data. The page need not do anything more than take data from the query string and save it to an error log. This page can then be used with code such as the following:

```
function logError(sev, msg){
    var img = new Image();
    img.src = "log.php?sev=" + encodeURIComponent(sev) + "&msg=" +
              encodeURIComponent(msg);
}
```

The `logError()` function accepts two arguments: a severity and the error message. The severity may be numbers or strings, depending on the system you're using. An `Image` object is used to send the request because of its flexibility, as described here:

- The `Image` object is available in all browsers, even those that don't support the `XMLHttpRequest` object.
- Cross-domain restrictions don't apply. Often there is one server responsible for handling error logging from multiple servers, and `XMLHttpRequest` would not work in that situation.
- There's less of a chance that an error will occur in the process of logging the error. Most Ajax communication is handled through functionality wrappers provided by JavaScript libraries. If that library's code fails, and you're trying to use it to log the error, the message may never get logged.

Whenever a `try-catch` statement is used, it's likely that the error should be logged. Here is an example:

```
for (var i=0, len=mods.length; i < len; i++) {
    try {
        mods[i].init();
    } catch (ex) {
        logError("nonfatal", "Module init failed: " + ex.message);
    }
}
```

In this code, `logError()` is called when a module fails to initialize. The first argument is "nonfatal", indicating the severity of the error, and the message provides contextual information plus the true JavaScript error message. Error messages that are logged to the server should provide as much contextual information as possible to help identify the exact cause of the error.

DEBUGGING TECHNIQUES

Before JavaScript debuggers were readily available, developers had to use creative methods to debug their code. This led to the placement of code specifically designed to output debugging information in one or more ways. The most common debugging technique was to insert alerts throughout the code in question, which was both tedious, because it required cleanup after the code was debugged, and annoying if an alert was mistakenly left in code that was used in a production environment. Alerts are no longer recommended for debugging purposes, because several other, more elegant solutions are available.

Logging Messages to a Console

Internet Explorer 8+, Firefox, Opera, Chrome, and Safari all have JavaScript consoles that can be used to view JavaScript errors. All three also allow you to write directly to the console from code. For this to work in Firefox 3.6 or earlier, you need to have Firebug installed (www.getfirebug.com), since it's the Firebug console that is used in Firefox. Internet Explorer 8+, Firefox, Chrome, Safari, and Opera 10.5 allow you to write to the JavaScript console via the `console` object, which has the following methods:

- `error(message)` — Logs an error message to the console
- `info(message)` — Logs an informational message to the console
- `log(message)` — Logs a general message to the console
- `warn(message)` — Logs a warning message to the console

In Internet Explorer 8, Firebug, Chrome, Safari, and Opera, the message display on the error console differs according to the method that was used to log the message. Error messages contain a red icon, whereas warnings contain a yellow icon. Console messages may be used, as in the following function:

```
function sum(num1, num2){
    console.log("Entering sum(), arguments are " + num1 + "," + num2);

    console.log("Before calculation");
    var result = num1 + num2;
    console.log("After calculation");

    console.log("Exiting sum()");
    return result;
}
```

As the `sum()` function is called, several messages are output to the JavaScript console to aid in debugging. The Safari JavaScript console can be opened via the Develop menu (discussed earlier); the Chrome JavaScript console is opened by clicking the “Control this page” button and selecting Developer JavaScript console (also discussed earlier); and the Firebug console is accessed by clicking the icon in the lower-right corner of the Firefox status bar. The Internet Explorer 8 console is part of the Developer Tools extension, which is available under the Tools menu; the console is on the Script tab.

Prior to version 10.5, Opera’s JavaScript console was accessible only using the `opera.postError()` method. This method accepts a single argument, the message to write to the console, and is used as follows:

```
function sum(num1, num2){
    opera.postError("Entering sum(), arguments are " + num1 + "," + num2);

    opera.postError("Before calculation");
    var result = num1 + num2;
```

```

        opera.postError("After calculation");

        opera.postError("Exiting sum()");
        return result;
    }

```

The `opera.postError()` method can be used to write out any type of information to the JavaScript console, despite its name.

Another option is to use LiveConnect, which is the ability to run Java code from JavaScript. Firefox, Safari, and Opera all support LiveConnect and may interact with a Java console. It's possible to write messages to the Java console using JavaScript via the following code:

```
java.lang.System.out.println("Your message");
```

This can be used in place of `console.log()` or `opera.postError()`, as shown in the following example:

```

function sum(num1, num2){
    java.lang.System.out.println("Entering sum(), arguments are " + num1 + ","
                                +
                                num2);

    java.lang.System.out.println("Before calculation");
    var result = num1 + num2;
    java.lang.System.out.println("After calculation");

    java.lang.System.out.println("Exiting sum()");
    return result;
}

```

Depending on system settings, the Java console may be displayed as soon as a LiveConnect call is made. The Java console is found in Firefox under the Tools menu and in Opera under the Tools ▾ Advanced menu. Safari doesn't have built-in support for opening the Java console; you must run it separately.

Since there is no true, cross-browser support for writing to the JavaScript console, the following function equalizes the interface:



```

function log(message){
    if (typeof console == "object"){
        console.log(message);
    } else if (typeof opera == "object"){
        opera.postError(message);
    } else if (typeof java == "object" && typeof java.lang == "object"){
        java.lang.System.out.println(message);
    }
}

```

[ConsoleLoggingExample01.htm](#)

The `log()` function detects which JavaScript console interface is available and uses the appropriate one. This function can safely be used in all browsers without causing any errors, as shown in the following example:

```
function sum(num1, num2){
    log("Entering sum(), arguments are " + num1 + "," + num2);

    log("Before calculation");
    var result = num1 + num2;
    log("After calculation");

    log("Exiting sum()");
    return result;
}
```

ConsoleLoggingExample01.htm

Logging messages to the JavaScript console is helpful in debugging code, but all messages should be removed when code goes to production. This can be done automatically, using a code-processing step in deployment, or manually.



Logging messages is considered a better debugging method than using alerts, because alerts interrupt program execution, which may affect the result of the code as timing of asynchronous processes are affected.

Logging Messages to the Page

Another common way to log debugging messages is to specify an area of the page that messages are written to. This may be an element that is included all the time but only used for debugging purposes, or an element that is created only when necessary. For example, the `log()` function may be changed to the following:



Available for
download on
[Wrox.com](#)

```
function log(message){
    var console = document.getElementById("debuginfo");
    if (console === null){
        console = document.createElement("div");
        console.id = "debuginfo";
        console.style.background = "#dedede";
        console.style.border = "1px solid silver";
        console.style.padding = "5px";
        console.style.width = "400px";
        console.style.position = "absolute";
        console.style.right = "0px";
        console.style.top = "0px";
    }
    console.log(message);
}
```

```

        document.body.appendChild(console);
    }
    console.innerHTML += "<p>" + message + "</p>";
}

```

[PageLoggingExample01.htm](#)

In this new version of `log()`, the code first checks to see if the debugging element already exists. If not, then a new `<div>` element is created and assigned stylistic information to separate it from the rest of the page. After that, the message is written into the `<div>` using `innerHTML`. The result is a small area that displays log information on the page. This approach may be useful when debugging code in Internet Explorer 7 and earlier or other browsers that don't support a JavaScript console.



As with console logging, page-logging code should be removed before the code is used in a production environment.

Throwing Errors

As mentioned earlier, throwing errors is an excellent way to debug code. If your error messages are specific enough, just seeing the error as it's reported may be enough to determine the error's source. The key to good error messages is for them to provide exact details about the cause of the error so that additional debugging is minimal. Consider the following function:

```

function divide(num1, num2) {
    return num1 / num2;
}

```

This simple function divides two numbers but will return `NaN` if either of the two arguments isn't a number. Simple calculations often cause problems in web applications when they return `NaN` unexpectedly. In this case, you can check that the type of each argument is a number before attempting the calculation. Consider the following example:

```

function divide(num1, num2) {
    if (typeof num1 != "number" || typeof num2 != "number") {
        throw new Error("divide(): Both arguments must be numbers.");
    }
    return num1 / num2;
}

```

Here, an error is thrown if either of the two arguments isn't a number. The error message provides the name of the function and the exact cause of the error. When the browser reports this error message, it immediately gives you a place to start looking for problems and a basic summary of the issue. This is much easier than dealing with a nonspecific browser error message.

In large applications, custom errors are typically thrown using an `assert()` function. Such a function takes a condition that should be true and throws an error if the condition is false. The following is a very basic `assert()` function:



```
function assert(condition, message) {
    if (!condition) {
        throw new Error(message);
    }
}
```

[AssertExample01.htm](#)

The `assert()` function can be used in place of multiple `if` statements in a function and can be a good location for error logging. This function can be used as follows:

```
function divide(num1, num2) {
    assert(typeof num1 == "number" && typeof num2 == "number",
           "divide(): Both arguments must be numbers.");
    return num1 / num2;
}
```

[AssertExample01.htm](#)

Using an `assert()` function reduces the amount of code necessary to throw custom errors and makes the code more readable compared to the previous example.

COMMON INTERNET EXPLORER ERRORS

Internet Explorer has traditionally been one of the most difficult browsers in which to debug JavaScript errors. The browser's error messages are generally short and confusing, with little or no context given. As Internet Explorer is the most popular web browser, its errors tend to get the most attention. The following sections provide a list of common and difficult-to-debug JavaScript errors that may occur.

Operation Aborted

Internet Explorer versions prior to 8 had perhaps one of the most confounding, annoying, and difficult-to-debug errors of all browsers: the operation aborted error. The operation aborted error occurs when part of the page that isn't yet fully loaded is being modified. The result is a modal dialog that says “Operation aborted.” When the OK button is clicked, the entire web page is unloaded and replaced with a blank screen, making it very difficult to debug. The following example page causes an operation aborted error:

```
<!DOCTYPE html>
<html>
<head>
    <title>Operation Aborted Example</title>
```

```

</head>
<body>
    <p>The following code should cause an Operation Aborted error in IE versions
       prior to 8.</p>
    <div>
        <script type="text/javascript">
            document.body.appendChild(document.createElement("div"));
        </script>
    </div>
</body>
</html>

```

OperationAbortedExample01.htm

The problems in this example are that the JavaScript code is attempting to modify `document.body` before it is fully loaded and the `<script>` element is not a direct child of the `<body>` element. To be more specific, this error will occur whenever a `<script>` node is contained within an element and the JavaScript code attempts to modify that element's parent or ancestors using `appendChild()`, `innerHTML`, or any other DOM method that assumes the element is fully loaded.

You can work around this problem either by waiting until the element is fully loaded before trying to manipulate it or by using a different manipulation method. For example, it's quite common to add overlays to `document.body` that will appear absolutely positioned on the page. These extra elements are typically added by using `appendChild()` but could easily be changed to use `insertBefore()`. The previous example could be rewritten to avoid an operation aborted error by changing just one line, as shown here:



```

<!DOCTYPE html>
<html>
<head>
    <title>Operation Aborted Example</title>
</head>
<body>
    <p>The following code should not cause an Operation Aborted error in IE
       versions prior to 8.</p>
    <div>
        <script type="text/javascript">
            document.body.insertBefore(document.createElement("div"),
                                         document.body.firstChild);
        </script>
    </div>
</body>
</html>

```

OperationAbortedExample02.htm

In this example, the new `<div>` element is added to the beginning of `document.body` instead of at the end. This won't cause an error, because all of the information needed to complete the operation is available when the script runs.

Another option is to move the `<script>` element so that it is a direct child of `<body>`. Consider the following example:



```
<!DOCTYPE html>
<html>
<head>
    <title>Operation Aborted Example</title>
</head>
<body>
    <p>The following code should not cause an Operation Aborted error in IE
    versions prior to 8.</p>
    <div>
    </div>
    <script type="text/javascript">
        document.body.appendChild(document.createElement("div"));
    </script>
</body>
</html>
```

OperationAbortedExample03.htm

Here, the operation aborted error doesn't occur because the script is modifying its immediate parent instead of an ancestor.

Internet Explorer 8 and later no longer throw operation aborted errors and instead throw a regular JavaScript error with the following message:

HTML Parsing Error: Unable to modify the parent container element before the child element is closed (KB927917).

The solution to the problem is the same even though the browser's reaction is different.

Invalid Character

The syntax of a JavaScript file must be made up of certain characters. When an invalid character is detected in a JavaScript file, IE throws the "invalid character" error. An invalid character is any character not defined as part of JavaScript syntax. For example, there is a character that looks like a minus sign but is represented by the Unicode value 8211 (\u2013). This character cannot be used in place of a regular minus sign (ASCII code of 45) because it's not part of JavaScript syntax. This special character is often automatically inserted into Microsoft Word documents, so you will get an illegal character error if you were to copy code written in Word to a text editor and then run it in Internet Explorer. Other browsers react similarly. Firefox throws an "illegal character" error, Safari reports a syntax error, and Opera reports a ReferenceError, because it interprets the character as an undefined identifier.

Member Not Found

As mentioned previously, all DOM objects in Internet Explorer are implemented as COM objects rather than in native JavaScript. This can result in some very strange behavior when it comes to

garbage collection. The "member not found" error is the direct result the mismatched garbage collection routines in Internet Explorer.

This error typically occurs when you're trying to assign a value to an object property after the object has already been destroyed. The object must be a COM object to get this specified error message. The best example of this occurs when you are using the event object. The Internet Explorer event object exists as a property of window and is created when the event occurs and destroyed after the last event handler has been executed. So if you were to use the event object in a closure that was to be executed later, any attempt to assign to a property of event will result in this error, as in the following example:

```
document.onclick = function(){
    var event = window.event;
    setTimeout(function(){
        event.returnValue = false;      //member not found error
    }, 1000);
};
```

In this code, a click handler is assigned to the document. It stores a reference to window.event in a local variable named event. This event variable is then referenced in a closure that is passed into setTimeout(). When the onclick event handler is exited, the event object is destroyed, so the reference in the closure is to an object whose members no longer exist. Assigning a value to returnValue causes the "member not found" error, because you cannot write to a COM object that has already destroyed its members.

Unknown Runtime Error

An unknown runtime error occurs when HTML is assigned using the innerHTML or outerHTML property in one of the following ways: if a block element is being inserted into an inline element or you're accessing either property on any part of a table (<table>, <tbody>, and so on). For example, a <p> tag cannot technically contain a block-level element such as a <div>, so the following code will cause an unknown runtime error:

```
p.innerHTML = "<div>Hi</div>";    //where p contains a <p> element
```

Other browsers attempt to error-correct when block elements are inserted in invalid places so that no error occurs, but Internet Explorer is much stricter in this regard.

Syntax Error

Often when Internet Explorer reports a syntax error, the cause is immediately apparent. You can usually trace back the error to a missing semicolon or an errant closing brace. However, there is another instance where a syntax error occurs that may not be immediately apparent.

If you are referencing an external JavaScript file that for some reason returns non-JavaScript code, Internet Explorer throws a syntax error. For example, if you set the src attribute of a <script> to point to an HTML file, a syntax error occurs. The syntax error is typically reported as the first line and first character of a script. Opera and Safari report a syntax error as well, but they will also

report the referenced file that caused the problem. Internet Explorer gives no such information, so you need to double-check every externally referenced JavaScript file. Firefox simply ignores any parsing errors in a non-JavaScript file that's included as if it were JavaScript.

This type of error typically occurs when JavaScript is being dynamically generated by a server-side component. Many server-side languages automatically insert HTML into the output if a runtime error occurs, and such output clearly breaks JavaScript syntax. If you're having trouble tracking down a syntax error, double-check each external JavaScript file to be sure that it doesn't contain HTML inserted by the server because of an error.

The System Cannot Locate the Resource Specified

Perhaps one of the least useful error messages is "The system cannot locate the resource specified." This error occurs when JavaScript is used to request a resource by URL and the URL is longer than Internet Explorer's maximum URL length of 2083 characters. This URL length limit applies not just to JavaScript but also to Internet Explorer in general. (Other browsers do not limit URL length so tightly.) There is also a URL path limit of 2048 characters. The following example causes this error:



Available for download on
Wrox.com

```
function createLongUrl(url){
    var s = "?";
    for (var i=0, len=2500; i < len; i++){
        s += "a";
    }

    return url + s;
}

var x = new XMLHttpRequest();
x.open("get", createLongUrl("http://www.somedomain.com/") , true);
x.send(null);
```

[LongURLErrorExample01.htm](#)

In this code, the `XMLHttpRequest` object attempts to make a request to a URL that exceeds the maximum URL limit. The error occurs when `open()` is called. One workaround for this type of error is to shorten the query string necessary for the request to succeed, either by decreasing the size of the named query string arguments or by eliminating unnecessary data. Another workaround is to change the request to a `POST` and send the data as the request body instead of in the query string. Ajax, the `XMLHttpRequest` object, and issues such as this are discussed fully in Chapter 21.

SUMMARY

Error handling in JavaScript is critical for today's complex web applications. Failing to anticipate where errors might occur and how to recover from them can lead to a poor user experience and possibly frustrated users. Most browsers don't report JavaScript errors to users by default, so you need to enable error reporting when developing and debugging. In production, however, no errors should ever be reported this way.

The following methods can be used to prevent the browser from reacting to a JavaScript error:

- The `try-catch` statement can be used where errors may occur, giving you the opportunity to respond to errors in an appropriate way instead of allowing the browser to handle the error.
- Another option is to use the `window.onerror` event handler, which receives all errors that are not handled by a `try-catch` (Internet Explorer, Firefox, and Chrome only).

Each web application should be inspected to determine where errors might occur and how those errors should be dealt with.

- A determination as to what constitutes a fatal error or a nonfatal error needs to be made ahead of time.
- After that, code can be evaluated to determine where the most likely errors will occur. Errors commonly occur in JavaScript because of the following factors:
 - Type coercion
 - Insufficient data type checking
 - Incorrect data being sent to or received from the server

Internet Explorer, Firefox, Chrome, Opera, and Safari each have JavaScript debuggers that either come with the browser or can be downloaded as an add-on. Each debugger offers the ability to set breakpoints, control code execution, and inspect the value of variables at runtime.

18

XML in JavaScript

WHAT'S IN THIS CHAPTER?

- ▶ Examining XML DOM support in browsers
- ▶ Understanding XPath in JavaScript
- ▶ Using XSLT processors

At one point in time, XML was the standard for structured data storage and transmission over the Internet. The evolution of XML closely mirrored the evolution of web technologies, as the DOM was developed for use not just in web browsers but also in desktop and server applications for dealing with XML data structures. Many developers started writing their own XML parsers in JavaScript to deal with the lack of built-in solutions. Since that time, all browsers have introduced native support for XML, the XML DOM, and many related technologies.

XML DOM SUPPORT IN BROWSERS

Since browser vendors began implementing XML solutions before formal standards were created, each offers not only different levels of support but also different implementations. DOM Level 2 was the first specification to introduce the concept of dynamic XML DOM creation. This capability was expanded on in DOM Level 3 to include parsing and serialization. By the time DOM Level 3 was finalized, however, most browsers had implemented their own solutions.

DOM Level 2 Core

As mentioned in Chapter 12, DOM Level 2 introduced the `createDocument()` method of `document.implementation`. Internet Explorer 9+, Firefox, Opera, Chrome, and Safari

support this method. You may recall that it's possible to create a blank XML document using the following syntax:

```
var xmldom = document.implementation.createDocument(namespaceUri, root, doctype);
```

When dealing with XML in JavaScript, the root argument is typically the only one that is used, because this defines the tag name of the XML DOM's document element. The namespaceUri argument is used sparingly, because namespaces are difficult to manage from JavaScript. The doctype argument is rarely, if ever, used.

To create a new XML document with document element of <root>, you can use the following code:



```
var xmldom = document.implementation.createDocument("", "root", null);
alert(xmldom.documentElement.tagName); // "root"
var child = xmldom.createElement("child");
xmldom.documentElement.appendChild(child);
```

[DOMLevel2CoreExample01.htm](#)

This example creates an XML DOM document with no default namespace and no doctype. Note that even though a namespace and doctype aren't needed, the arguments must still be passed in. An empty string is passed as the namespace URI so that no namespace is applied, and `null` is passed as the doctype. The `xmldom` variable contains an instance of the DOM Level 2 Document type, complete with all of the DOM methods and properties discussed in Chapter 12. In this example, the document element's tag name is displayed and then a new child element is created and added.

You can check to see if DOM Level 2 XML support is enabled in a browser by using the following line of code:

```
var hasXmlDom = document.implementation.hasFeature("XML", "2.0");
```

In practice, it is rare to create an XML document from scratch and then build it up systematically using DOM methods. It is much more likely that an XML document needs to be parsed into a DOM structure or vice versa. Because DOM Level 2 didn't provide for such functionality, a couple of de facto standards emerged.

The DOMParser Type

Firefox introduced the `DOMParser` type specifically for parsing XML into a DOM document, and it was later adopted by Internet Explorer 9, Safari, Chrome, and Opera. To use it, you must first create an instance of `DOMParser` and then call the `parseFromString()` method. This method accepts two arguments: the XML string to parse and a content type, which should always be `"text/xml"`. The return value is an instance of `Document`. Consider the following example:

```
var parser = new DOMParser();
var xmldom = parser.parseFromString("<root><child/></root>", "text/xml");

alert(xmldom.documentElement.tagName); // "root"
```

```

alert(xmlDom.documentElement.firstChild.tagName); // "child"

var anotherChild = xmlDom.createElement("child");
xmlDom.documentElement.appendChild(anotherChild);

var children = xmlDom.getElementsByTagName("child");
alert(children.length); // 2

```

[DOMParserExample01.htm](#)

In this example, a simple XML string is parsed into a DOM document. The DOM structure has <root> as the document element with a single <child> element as its child. You can then interact with the returned document using DOM methods.

The `DOMParser` can parse only well-formed XML and, as such, cannot parse HTML into an HTML document. Unfortunately, browsers behave differently when a parsing error occurs. When a parsing error occurs in Firefox, Opera, Safari, and Chrome, a `Document` object is still returned from `parseFromString()`, but its document element is <parsererror> and the content of the element is a description of the parsing error. Here is an example:

```

<parsererror xmlns="http://www.mozilla.org/newlayout/xml/parsererror.xml">XML
Parsing Error: no element found Location: file:///I:/My%20Writing/My%20Books/Professional%20JavaScript/Second%20Edition/Examples/Ch15/DOMParserExample2.htm Line Number
1, Column 7:<sourcetext>&lt;root&gt; -----^</sourcetext></parsererror>

```

Firefox and Opera both return documents in this format. Safari and Chrome return a document that has a <parsererror> element embedded at the point where the parsing error occurred. Internet Explorer 9 throws a parsing error at the point where `parseFromString()` is called. Because of these differences, the best way to determine if a parsing error has occurred is to use a `try-catch` block, and if there's no error, look for a <parsererror> element anywhere in the document via `getElementsByTagName()`, as shown here:



Available for
download on
Wrox.com

```

var parser = new DOMParser(),
    xmlDom,
    errors;
try {
    xmlDom = parser.parseFromString("<root>", "text/xml");
    errors = xmlDom.getElementsByTagName("parsererror");
    if (errors.length > 0){
        throw new Error("Parsing error!");
    }
} catch (ex) {
    alert("Parsing error!");
}

```

[DOMParserExample02.htm](#)

In this example, the string to be parsed is missing a closing </root> tag, which causes a parse error. In Internet Explorer 9+, this throws an error. In Firefox and Opera, the <parsererror> element will be the document element, whereas it's the first child <root> in Chrome and Safari. The call to

`getElementsByName("parsererror")` covers both cases. If any elements are returned by this method call, then an error has occurred and an alert is displayed. You could go one step further and extract the error information from the element as well.

The `XMLSerializer` Type

As a companion to `DOMParser`, Firefox also introduced the `XMLSerializer` type to provide the reverse functionality: serializing a DOM document into an XML string. Since that time, the `XMLSerializer` has been adopted by Internet Explorer 9+, Opera, Chrome, and Safari.

To serialize a DOM document, you must create a new instance of `XMLSerializer` and then pass the document into the `serializeToString()` method, as in this example:



```
var serializer = new XMLSerializer();
var xml = serializer.serializeToString(xmlDom);
alert(xml);
```

XMLSerializerExample01.htm

The value returned from `serializeToString()` is a string that is not pretty-printed, so it may be difficult to read with the naked eye.

The `XMLSerializer` is capable of serializing any valid DOM object, which includes individual nodes and HTML documents. When an HTML document is passed into `serializeToString()`, it is treated as an XML document, and so the resulting code is well-formed.



If a non-DOM object is passed into the `serializeToString()` method, an error is thrown.

XML in Internet Explorer 8 and Earlier

Internet Explorer was actually the first browser to implement native XML processing support, and it did so through the use of ActiveX objects. Microsoft created the MSXML library to provide desktop-application developers with XML processing capabilities, and instead of creating different objects for JavaScript, they just enabled access to the same objects through the browser.

In Chapter 8, you were introduced to the `ActiveXObject` type, which is used to instantiate ActiveX objects in JavaScript. An XML document instance is created using the `ActiveXObject` constructor and passing in the string identifier for the XML document version. There are six different XML document objects, as described here:

- `Microsoft.XmlDom` — Initial release with IE; should not be used.
- `MSXML2.DOMDocument` — Updated version for scripting purposes but considered an emergency fallback only.
- `MSXML2.DOMDocument.3.0` — Lowest recommended version for JavaScript usage.

- `MSXML2.DOMDocument.4.0` — Not considered safe for scripting, so attempting to use it may result in a security warning.
- `MSXML2.DOMDocument.5.0` — Also not considered safe for scripting and may cause a security warning.
- `MSXML2.DOMDocument.6.0` — The most recent version marked safe for scripting.

Of the six versions, Microsoft recommends using only `MSXML2.DOMDocument.6.0`, which is the most recent and robust version, or `MSXML2.DOMDocument.3.0`, which is the version that is available on most Windows computers. The last fallback is `MSXML2.DOMDocument`, which may be necessary for browsers earlier than Internet Explorer 5.5.

You can determine which version is available by attempting to create each and watching for errors. For example:



```
function createDocument(){
    if (typeof arguments.callee.activeXString != "string"){
        var versions = ["MSXML2.DOMDocument.6.0", "MSXML2.DOMDocument.3.0",
                       "MSXML2.DOMDocument"],
            i, len;

        for (i=0,len=versions.length; i < len; i++){
            try {
                new ActiveXObject(versions[i]);
                arguments.callee.activeXString = versions[i];
                break;
            } catch (ex){
                //skip
            }
        }
    }

    return new ActiveXObject(arguments.callee.activeXString);
}
```

[IEXmlDomExample01.htm](#)

In this function, a `for` loop is used to iterate over the possible ActiveX versions. If the version isn't available, the call to create a new `ActiveXObject` throws an error, in which case the `catch` statement catches the error and the loop continues. If an error doesn't occur, then the version is stored as the `activeXString` property of the function, so that this process needn't be repeated each time the function is called, and the created object is returned.

To parse an XML string, you must first create a DOM document and then call the `loadXML()` method. When the document is first created, it is completely empty and so cannot be interacted with. Passing an XML string into `loadXML()` parses the XML into the DOM document. Here's an example:

```
var xmldom = createDocument();
xmldom.loadXML("<root><child/></root>");

alert(xmldom.documentElement.tagName); // "root"
```

```

alert(xmlDom.documentElement.firstChild.tagName); // "child"

var anotherChild = xmlDom.createElement("child");
xmlDom.documentElement.appendChild(anotherChild);

var children = xmlDom.getElementsByTagName("child");
alert(children.length); // 2

```

IExmlDomExample01.htm

Once the DOM document is filled with XML content, it can be interacted with just like any other DOM document, including all methods and properties.

Parsing errors are represented by the `parseError` property, which is an object with several properties relating to any parsing issues. These properties are as follows:

- `errorCode` — Numeric code indicating the type of error that occurred, or 0 when there's no error.
- `filePos` — Position within the file where the error occurred.
- `line` — The line on which the error occurred.
- `linepos` — The character on the line where the error occurred.
- `reason` — A plain text explanation of the error.
- `srcText` — The code that caused the error.
- `url` — The URL of the file that caused the error (if available).

The `valueOf()` method for `parseError` returns the value of `errorCode`, so you can check to see if a parsing error occurred by using the following:

```

if (xmlDom.parseError != 0){
    alert("Parsing error occurred.");
}

```

An error code may be a positive or negative number, so you need only check to see if it's not equal to 0. The details of the parsing error are easily accessible and can be used to indicate more useful error information, as shown in the following example:



Available for download on Wrox.com

```

if (xmlDom.parseError != 0){
    alert("An error occurred:\nError Code: "
        + xmlDom.parseError.errorCode + "\n"
        + "Line: " + xmlDom.parseError.line + "\n"
        + "Line Pos: " + xmlDom.parseError.linepos + "\n"
        + "Reason: " + xmlDom.parseError.reason);
}

```

IExmlDomExample02.htm

You should check for parsing errors immediately after a call to `loadXML()` and before attempting to query the XML document for more information.

Serializing XML

XML serialization is built into the DOM document in Internet Explorer. Each node has an `xml` property that can be used to retrieve the XML string representing that node, as in this example:

```
alert(xmlDom.xml);
```

This simple serialization method is available on every node in the document, allowing you to serialize the entire document or a specific subtree.

Loading XML Files

The XML document object in Internet Explorer can also load files from a server. As with the DOM Level 3 functionality, XML documents must be located on the same server as the page running the JavaScript code. Also similar to DOM Level 3, documents can be loaded synchronously or asynchronously. To determine which method to use, set the `async` property to either `true` or `false` (it's `true` by default). Here's an example:

```
var xmldom = createDocument();
xmldom.async = false;
```

Once you've determined the mode to load the XML document in, a call to `load()` initiates the download process. This method takes a single argument, which is the URL of the XML file to load. When run in synchronous mode, a call to `load()` can immediately be followed by a check for parsing errors and other XML processing, such as this:



Available for
download on
Wrox.com

```
var xmldom = createDocument();
xmldom.async = false;
xmldom.load("example.xml");

if (xmldom.parseError != 0){
    //handle error
} else {
    alert(xmldom.documentElement.tagName); // "root"
    alert(xmldom.documentElement.firstChild.tagName); // "child"

    var anotherChild = xmldom.createElement("child");
    xmldom.documentElement.appendChild(anotherChild);

    var children = xmldom.getElementsByTagName("child");
    alert(children.length); // 2

    alert(xmldom.xml);
}
```

IEXmlDomExample03.htm

Because the XML file is being processed synchronously, code execution is halted until the parsing is complete, allowing a simple coding procedure. Although this may be convenient, it could also lead to a long delay if the download takes longer than expected. XML documents are typically loaded asynchronously to avoid such issues.

When an XML file is loaded asynchronously, you need to assign an `onreadystatechange` event handler to the XML DOM document. There are four different ready states:

- 1 — The DOM is loading data.
- 2 — The DOM has completed loading the data.
- 3 — The DOM may be used although some sections may not be available.
- 4 — The DOM is completely loaded and ready to be used.

Practically speaking, the only ready state of interest is 4, which indicates that the XML file has been completely downloaded and parsed into a DOM. You can retrieve the ready state of the XML document via the `readyState` property. Loading an XML file asynchronously typically uses the following pattern:



Available for
download on
Wrox.com

```
var xmldom = createDocument();
xmldom.async = true;

xmldom.onreadystatechange = function(){
    if (xmldom.readyState == 4){
        if (xmldom.parseError != 0){
            alert("An error occurred:\nError Code: "
                + xmldom.parseError.errorCode + "\n"
                + "Line: " + xmldom.parseError.line + "\n"
                + "Line Pos: " + xmldom.parseError.linepos + "\n"
                + "Reason: " + xmldom.parseError.reason);
        } else {
            alert(xmldom.documentElement.tagName); // "root"
            alert(xmldom.documentElement.firstChild.tagName); // "child"

            var anotherChild = xmldom.createElement("child");
            xmldom.documentElement.appendChild(anotherChild);

            var children = xmldom.getElementsByTagName("child");
            alert(children.length); // 2

            alert(xmldom.xml);
        }
    }
};

xmldom.load("example.xml");
```

IEXmlDomExample04.htm

Note that the assignment of the `onreadystatechange` event handler must happen before the call to `load()` to ensure that it gets called in time. Also note that inside of the event handler, you must use the name of the XML document variable, `xmldom`, instead of the `this` object. ActiveX controls disallow the use of `this` as a security precaution. Once the ready state of the document reaches 4, you can safely check to see if there's a parsing error and begin your XML processing.



Even though it's possible to load XML files via the XML DOM document object, it's generally accepted to use an XMLHttpRequest for this instead. The XMLHttpRequest object, and Ajax in general, is discussed in Chapter 21.

Cross-Browser XML Processing

Since there are very few developers with the luxury of developing for a single browser, it's frequently necessary to create browser-equalizing functions for XML processing. For XML parsing, the following function works in all of the major browsers:



Available for download on
Wrox.com

```
function parseXml(xml){
    var xmldom = null;

    if (typeof DOMParser != "undefined"){
        xmldom = (new DOMParser()).parseFromString(xml, "text/xml");
        var errors = xmldom.getElementsByTagName("parsererror");
        if (errors.length){
            throw new Error("XML parsing error: " + errors[0].textContent);
        }
    } else if (typeof ActiveXObject != "undefined"){
        xmldom = createDocument();
        xmldom.loadXML(xml);
        if (xmldom.parseError != 0){
            throw new Error("XML parsing error: " + xmldom.parseError.reason);
        }
    } else {
        throw new Error("No XML parser available.");
    }

    return xmldom;
}
```

[CrossBrowserXmlExample01.htm](#)

The `parseXml()` function accepts a single argument, the XML string to parse, and then uses capability detection to determine which XML parsing pattern to use. Since the `DOMParser` type is the most widely available solution, the function first tests to see if it is available. If so, a new `DOMParser` object is created and the XML string is parsed into the `xmldom` variable. Since `DOMParser` won't throw an error for parsing errors other than in Internet Explorer 9+, the returned document is checked for errors, and if one is found, an error is thrown with the message.

The last part of the function checks for ActiveX support and uses the `createDocument()` function defined earlier to create an XML document using the correct signature. As with `DOMParser`, the result is checked for parsing errors. If one is found, then an error is thrown indicating the reported description.

If no XML parser is available, then the function simply throws an error indicating that it could not continue.

This function can be used to parse any XML string and should always be wrapped in a `try-catch` statement just in case a parsing error occurs. Here's an example:



Available for
download on
Wrox.com

```
var xmldom = null;

try {
    xmldom = parseXml("<root><child/></root>");
} catch (ex){
    alert(ex.message);
}

//further processing
```

[CrossBrowserXmlExample01.htm](#)

For XML serialization, the same process can be followed to write a function that works in the four major browsers. For example:

```
function serializeXml(xmldom){

    if (typeof XMLSerializer != "undefined"){
        return (new XMLSerializer()).serializeToString(xmldom);

    } else if (typeof xmldom.xml != "undefined"){
        return xmldom.xml;

    } else {
        throw new Error("Could not serialize XML DOM.");
    }
}
```

[CrossBrowserXmlExample02.htm](#)

The `serializeXml()` function accepts a single argument, which is the XML DOM document to serialize. As with the `parseXml()` function, the first step is to check for the most widely available solution, which is `XMLSerializer`. If this type is available, then it is used to return the XML string for the document. Since the ActiveX approach simply uses the `xml` property, the function checks for that property specifically. If each of these three attempts fails, then the method throws an error indicating that serialization could not take place. Generally, serialization attempts shouldn't fail if you're using the appropriate XML DOM object for the browser, so it shouldn't be necessary to wrap a call to `serializeXml()` in a `try-catch`. Instead, you can simply use this:

```
var xml = serializeXml(xmldom);
```

Note that because of differences in serializing logic, you may not end up with exactly the same serialization results from browser to browser.

XPATH SUPPORT IN BROWSERS

XPath was created as a way to locate specific nodes within a DOM document, so it's important to XML processing. An API for XPath wasn't part of a specification until DOM Level 3, which introduced the DOM Level 3 XPath recommendation. Many browsers chose to implement this specification, but Internet Explorer decided to implement support in its own way.

DOM Level 3 XPath

The DOM Level 3 XPath specification defines interfaces to use for evaluating XPath expressions in the DOM. To determine if the browser supports DOM Level 3 XPath, use the following JavaScript code:

```
var supportsXPath = document.implementation.hasFeature("XPath", "3.0");
```

Although there are several types defined in the specification, the two most important ones are `xPathEvaluator` and `xPathResult`. The `xPathEvaluator` is used to evaluate XPath expressions within a specific context. This type has the following three methods:

- `createExpression(expression, nsresolver)` — Computes the XPath expression and accompanying namespace information into an `xPathExpression`, which is a compiled version of the query. This is useful if the same query is going to be run multiple times.
- `createNSResolver(node)` — Creates a new `xPathNSResolver` object based on the namespace information of `node`. An `xPathNSResolver` object is required when evaluating against an XML document that uses namespaces.
- `evaluate(expression, context, nsresolver, type, result)` — Evaluates an XPath expression in the given context and with specific namespace information. The additional arguments indicate how the result should be returned.

In Firefox, Safari, Chrome, and Opera, the `Document` type is typically implemented with the `xPathEvaluator` interface. So you can either create a new instance of `xPathEvaluator` or use the methods located on the `Document` instance (for both XML and HTML documents).

Of the three methods, `evaluate()` is the most frequently used. This method takes five arguments: the XPath expression, a context node, a namespace resolver, the type of result to return, and an `xPathResult` object to fill with the result (usually `null`, since the result is also returned as the function value). The third argument, the namespace resolver, is necessary only when the XML code uses an XML namespace. If namespaces aren't used, this should be set to `null`. The fourth argument, the type of result to return, is one of the following 10 constants values:

- `xPathResult.ANY_TYPE` — Returns the type of data appropriate for the XPath expression.
- `xPathResult.NUMBER_TYPE` — Returns a number value.
- `xPathResult.STRING_TYPE` — Returns a string value.
- `xPathResult.BOOLEAN_TYPE` — Returns a Boolean value.
- `xPathResult.UNORDERED_NODE_ITERATOR_TYPE` — Returns a node set of matching nodes, although the order may not match the order of the nodes within the document.

- `XPathResult.ORDERED_NODE_ITERATOR_TYPE` — Returns a node set of matching nodes in the order in which they appear in the document. This is the most commonly used result type.
- `XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE` — Returns a node set snapshot, capturing the nodes outside of the document so that any further document modification doesn't affect the node set. The nodes in the node set are not necessarily in the same order as they appear in the document.
- `XPathResult.ORDERED_NODE_SNAPSHOT_TYPE` — Returns a node set snapshot, capturing the nodes outside of the document so that any further document modification doesn't affect the result set. The nodes in the result set are in the same order as they appear in the document.
- `XPathResult.ANY_UNORDERED_NODE_TYPE` — Returns a node set of matching nodes, although the order may not match the order of the nodes within the document.
- `XPathResult.FIRST_ORDERED_NODE_TYPE` — Returns a node set with only one node, which is the first matching node in the document.

The type of result you specify determines how to retrieve the value of the result. Here's a typical example:



Available for
download on
Wrox.com

```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.ORDERED_NODE_ITERATOR_TYPE, null);

if (result !== null) {
    var element = result.iterateNext();
    while(element) {
        alert(element.tagName);
        node = result.iterateNext();
    }
}
```

[DomXPathExample01.htm](#)

This example uses the `XPathResult.ORDERED_NODE_ITERATOR_TYPE` result, which is the most commonly used result type. If no nodes match the XPath expression, `evaluate()` returns `null`; otherwise, it returns an `XPathResult` object. The `XPathResult` has properties and methods for retrieving results of specific types. If the result is a node iterator, whether it be ordered or unordered, the `iterateNext()` method must be used to retrieve each matching node in the result. When there are no further matching nodes, `iterateNext()` returns `null`.

If you specify a snapshot result type (either ordered or unordered), you must use the `snapshotItem()` method and `snapshotLength` property, as in the following example:

```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.ORDERED_NODE_SNAPSHOT_TYPE,
                             null);
if (result !== null) {
    for (var i=0, len=result.snapshotLength; i < len; i++) {
        alert(result.snapshotItem(i).tagName);
    }
}
```

[DomXPathExample02.htm](#)

In this example, `snapshotLength` returns the number of nodes in the snapshot, and `snapshotItem()` returns the node in a given position in the snapshot (similar to `length` and `item()` in a `NodeList`).

Single Node Results

The `XPathResult.FIRST_ORDERED_NODE_TYPE` result returns the first matching node, which is accessible through the `singleNodeValue` property of the result. For example:



Available for download on
Wrox.com

```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.FIRST_ORDERED_NODE_TYPE, null);

if (result !== null) {
    alert(result.singleNodeValue.tagName);
}
```

[DomXPathExample03.htm](#)

As with other queries, `evaluate()` returns `null` when there are no matching nodes. If a node is returned, it is accessed using the `singleNodeValue` property. This is the same for `XPathResult.FIRST_ORDERED_NODE_TYPE`.

Simple Type Results

It's possible to retrieve simple, nonnode data types from XPath as well, using the `XPathResult` types of Boolean, number, and string. These result types return a single value using the `booleanValue`, `numberValue`, and `stringValue` properties, respectively. For the Boolean type, the evaluation typically returns `true` if at least one node matches the XPath expression and returns `false` otherwise.

Consider the following:

```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.BOOLEAN_TYPE, null);
alert(result.booleanValue);
```

[DomXPathExample04.htm](#)

In this example, if any nodes match "`employee/name`", the `booleanValue` property is equal to `true`. For the number type, the XPath expression must use an XPath function that returns a number, such as `count()`, which counts all the nodes that match a given pattern. Here's an example:

```
var result = xmldom.evaluate("count(employee/name)", xmldom.documentElement,
                            null, XPathResult.NUMBER_TYPE, null);
alert(result.numberValue);
```

[DomXPathExample05.htm](#)

This code outputs the number of nodes that match "employee/name" (which is 2). If you try using this method without one of the special XPath functions, `numberValue` is equal to NaN.

For the string type, the `evaluate()` method finds the first node matching the XPath expression, and then returns the value of the first child node, assuming the first child node is a text node. If not, the result is an empty string. Here is an example:



```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.STRING_TYPE, null);
alert(result.stringValue);
```

[DomXPathExample06.htm](#)

In this example, the code outputs the string contained in the first text node under the first element matching "element/name".

Default Type Results

All XPath expressions automatically map to a specific result type. Setting the specific result type limits the output of the expression. You can, however, use the `XPathResult.ANY_TYPE` constant to allow the automatic result type to be returned. Typically, the result type ends up as a Boolean value, a number value, a string value, or an unordered node iterator. To determine which result type has been returned, use the `resultType` property on the evaluation result, as shown in this example:

```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.ANY_TYPE, null);

if (result !== null) {
    switch(result.resultType) {
        case XPathResult.STRING_TYPE:
            //handle string type
            break;

        case XPathResult.NUMBER_TYPE:
            //handle number type
            break;

        case XPathResult.BOOLEAN_TYPE:
            //handle boolean type
            break;

        case XPathResult.UNORDERED_NODE_ITERATOR_TYPE:
            //handle unordered node iterator type
            break;

        default:
            //handle other possible result types
    }
}
```

Using the `XPathResult.ANY_TYPE` constant allows more natural use of XPath but may also require extra processing code after the result is returned.

Namespace Support

For XML documents that make use of namespaces, the `XPathEvaluator` must be informed of the namespace information in order to make a proper evaluation. There are a number of ways to accomplish this. Consider the following XML code:

```
<?xml version="1.0" ?>
<wrox:books xmlns:wrox="http://www.wrox.com/">
    <wrox:book>
        <wrox:title>Professional JavaScript for Web Developers</wrox:title>
        <wrox:author>Nicholas C. Zakas</wrox:author>
    </wrox:book>
    <wrox:book>
        <wrox:title>Professional Ajax</wrox:title>
        <wrox:author>Nicholas C. Zakas</wrox:author>
        <wrox:author>Jeremy McPeak</wrox:author>
        <wrox:author>Joe Fawcett</wrox:author>
    </wrox:book>
</wrox:books>
```

In this XML document, all elements are part of the `http://www.wrox.com/` namespace, identified by the `wrox` prefix. If you want to use XPath with this document, you need to define the namespaces being used; otherwise the evaluation will fail.

The first way to handle namespaces is by creating an `XPathNSResolver` object via the `createNSResolver()` method. This method accepts a single argument, which is a node in the document that contains the namespace definition. In the previous example, this node is the document element `<wrox:books>`, which has the `xmlns` attribute defining the namespace. This node can be passed into `createNSResolver()`, and the result can then be used in `evaluate()` as follows:



Available for download on
Wrox.com

```
var nsresolver = xmldom.createNSResolver(xmldom.documentElement);
var result = xmldom.evaluate("wrox:book/wrox:author",
    xmldom.documentElement, nsresolver,
    XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);

alert(result.snapshotLength);
```

[DomXPathExample07.htm](#)

When the `nsresolver` object is passed into `evaluate()`, it ensures that the `wrox` prefix used in the XPath expression will be understood appropriately. Attempting to use this same expression without using an `XPathNSResolver` will result in an error.

The second way to deal with namespaces is by defining a function that accepts a namespace prefix and returns the associated URI, as in this example:

```
var nsresolver = function(prefix) {
    switch(prefix) {
        case "wrox": return "http://www.wrox.com/";
        //others here
    }
}
```

```

};

var result = xmldom.evaluate("count(wrox:book/wrox:author)",
    xmldom.documentElement, nsresolver, XPathResult.NUMBER_TYPE, null);

alert(result.numberValue);

```

DomXPathExample08.htm

Defining a namespace-resolving function is helpful when you're not sure which node of a document contains the namespace definitions. As long as you know the prefixes and URIs, you can define a function to return this information and pass it in as the third argument to `evaluate()`.

XPath in Internet Explorer

XPath support is built into the ActiveX-based XML DOM document object in Internet Explorer but not with the DOM object returned from a `DOMParser`. In order to use XPath in Internet Explorer through version 9, you must use the ActiveX implementation. The interface defines two additional methods on every node: `selectSingleNode()` and `selectNodes()`. The `selectSingleNode()` method accepts an XPath pattern and returns the first matching node if found or `null` if there are no nodes. For example:



Available for
download on
Wrox.com

```

var element = xmldom.documentElement.selectSingleNode("employee/name");

if (element !== null){
    alert(element.xml);
}

```

IEXPathExample01.htm

Here, the first node matching "employee/name" is returned. The context node is `xmldom.documentElement`, the node on which `selectSingleNode()` is called. Since it's possible to get a `null` value returned from this method, you should always check to ensure that the value isn't `null` before attempting to use node methods.

The `selectNodes()` method also accepts an XPath pattern as an argument, but it returns a `NodeList` of all nodes matching the pattern (if no nodes match, a `NodeList` with zero items is returned). Here is an example:

```

var elements = xmldom.documentElement.selectNodes("employee/name");
alert(elements.length);

```

IEXPathExample02.htm

In this example, all of the elements matching "employee/name" are returned as a `NodeList`. Since there is no possibility of a `null` value being returned, you can safely begin using the result. Remember that because the result is a `NodeList`, it is a dynamic collection that will constantly be updated every time it's accessed.

XPath support in Internet Explorer is very basic. It's not possible to get result types other than a node or NodeList.

Namespace Support in Internet Explorer

To deal with XPath expressions that contain namespaces in Internet Explorer, you'll need to know which namespaces you're using and create a string in the following format:

```
"xmlns:prefix1='uri1' xmlns:prefix2='uri2' xmlns:prefix3='uri3'"
```

This string then must be passed to a special method on the XML DOM document object in Internet Explorer called `setProperty()`, which accepts two arguments: the name of the property to set and the property value. In this case, the name of the property is "SelectionNamespaces", and the value is a string in the format mentioned previously. Therefore, the following code can be used to evaluate the XML document used in the DOM XPath namespaces example:



Available for download on Wrox.com

```
xmldom.setProperty("SelectionNamespaces", "xmlns:wrox='http://www.wrox.com/'");
```

```
var result = xmldom.documentElement.selectNodes("wrox:book/wrox:author");
alert(result.length);
```

[IEXPathExample03.htm](#)

As with the DOM XPath example, failing to provide the namespace resolution information results in an error when the expression is evaluated.

Cross-Browser XPath

Since XPath functionality is so limited in Internet Explorer, cross-browser XPath usage must be kept to evaluations that Internet Explorer can execute. This means, essentially, recreating the `selectSingleNode()` and `selectNodes()` methods in other browsers using the DOM Level 3 XPath objects. The first function is `selectSingleNode()`, which accepts three arguments: the context node, the XPath expression, and an optional `namespaces` object. The `namespaces` object should be a literal in the following form:

```
{
    prefix1: "uri1",
    prefix2: "uri2",
    prefix3: "uri3"
}
```

Providing the namespace information in this way allows for easy conversion into the browser-specific namespace-resolving format. The full code for `selectSingleNode()` is as follows:

```
function selectSingleNode(context, expression, namespaces) {
    var doc = (context.nodeType != 9 ? context.ownerDocument : context);

    if (typeof doc.evaluate != "undefined") {
        var nsresolver = null;
```

```
if (namespaces instanceof Object){
    nsresolver = function(prefix){
        return namespaces[prefix];
    };
}

var result = doc.evaluate(expression, context, nsresolver,
                         XPathResult.FIRST_ORDERED_NODE_TYPE, null);
return (result !== null ? result.singleNodeValue : null);

} else if (typeof context.selectSingleNode != "undefined"){

    //create namespace string
    if (namespaces instanceof Object){
        var ns = "";
        for (var prefix in namespaces){
            if (namespaces.hasOwnProperty(prefix)){
                ns += "xmlns:" + prefix + "=" + namespaces[prefix] + ' ';
            }
        }
        doc.setProperty("SelectionNamespaces", ns);
    }
    return context.selectSingleNode(expression);
} else {
    throw new Error("No XPath engine found.");
}
}
```

CrossBrowserXPathExample01.htm

The first step in this function is to determine the XML document on which to evaluate the expression. Since a context node can be a document, it's necessary to check the `nodeType` property. The variable `doc` holds a reference to the XML document after doing this check. At that point, you can check the document to see if the `evaluate()` method is present, indicating DOM Level 3 XPath support. If it is supported, the next step is to see if a `namespaces` object has been passed in. This is done by using the `instanceof` operator, because `typeof` returns "object" for `null` values and objects. The `nsresolver` variable is initialized to `null` and then overwritten with a function if namespace information is available. This function is a closure, using the passed-in `namespaces` object to return namespace URIs. After that, the `evaluate()` method is called and the result is inspected to determine whether or not a node was returned before returning a value.

The Internet Explorer branch of the function checks for the existence of the `selectSingleNode()` method on the `context` node. As with the DOM branch, the first step is to construct namespace information for the selection. If a `namespaces` object is passed in, then its properties are iterated over to create a string in the appropriate format. Note the use of the `hasOwnProperty()` method to ensure that any modifications to `Object.prototype` are not picked up by this function. The native `selectSingleNode()` method is then called and the result is returned.

If neither of the two methods is supported, then the function throws an error indicating that there's no XPath engine available. The `selectSingleNode()` function can be used as follows:



```
var result = selectSingleNode(xmlDom.documentElement, "wrox:book/wrox:author",
{ wrox: "http://www.wrox.com/" });
alert(serializeXml(result));
```

[CrossBrowserXPathExample01.htm](#)

A cross-browser `selectNodes()` function is created in a very similar fashion. The function accepts the same three arguments as the `selectSingleNode()` function and much of its logic is similar. For ease of reading, the following highlights the differences between the functions:

```
function selectNodes(context, expression, namespaces){
    var doc = (context.nodeType != 9 ? context.ownerDocument : context);

    if (typeof doc.evaluate != "undefined"){
        var nsresolver = null;
        if (namespaces instanceof Object){
            nsresolver = function(prefix){
                return namespaces[prefix];
            };
        }

        var result = doc.evaluate(expression, context, nsresolver,
            XPathResult.ORDERED_NODE_SNAPSHOT_TYPE,
            null);
        var nodes = new Array();

        if (result !== null){
            for (var i=0, len=result.snapshotLength; i < len; i++){
                nodes.push(result.snapshotItem(i));
            }
        }
    }

    return nodes;
} else if (typeof context.selectNodes != "undefined"){

    //create namespace string
    if (namespaces instanceof Object){
        var ns = "";
        for (var prefix in namespaces){
            if (namespaces.hasOwnProperty(prefix)){
                ns += "xmlns:" + prefix + "=" + namespaces[prefix] + ' ';
            }
        }
        doc.setProperty("SelectionNamespaces", ns);
    }

    var result = context.selectNodes(expression);
    var nodes = new Array();

    for (var i=0, len=result.length; i < len; i++) {
```

```
        nodes.push(result[i]);
    }

    return nodes;
} else {
    throw new Error("No XPath engine found.");
}
}
```

CrossBrowserXPathExample02.htm

As you can see, much of the same logic is used from `selectSingleNode()`. In the DOM portion of the code, an ordered snapshot result type is used and then stored in an array. To match the Internet Explorer implementation, the function should return an array even if no results were found, so the `nodes` array is always returned. In the Internet Explorer branch of the code, the `selectNodes()` method is called and then the results are copied into an array. Since Internet Explorer returns a `NodeList`, it's best to copy the nodes over into an array, so the function returns the same type regardless of the browser being used. This function can then be used as follows:



Available for
download on
Wrox.com

```
var result = selectNodes(xmlDom.documentElement, "wrox:book/wrox:author",
                        { wrox: "http://www.wrox.com/" });
alert(result.length);
```

CrossBrowserXPathExample02.htm

For the best cross-browser compatibility, it's best to use these two methods exclusively for XPath processing in JavaScript.

XSLT SUPPORT IN BROWSERS

XSLT is a companion technology to XML that makes use of XPath to transform one document representation into another. Unlike XML and XPath, XSLT has no formal API associated with it and is not represented in the formal DOM at all. This left browser vendors to implement support in their own way. The first browser to add XSLT processing in JavaScript was Internet Explorer.

XSLT in Internet Explorer

As with the rest of the XML functionality in Internet Explorer, XSLT support is provided through the use of ActiveX objects. Beginning with MSXML 3.0 (shipped with Internet Explorer 6), full XSLT 1.0 support is available via JavaScript. There is no XSLT support for DOM documents created using a `DOMParser` in Internet Explorer 9.

Simple XSLT Transformations

The simplest way to transform an XML document using an XSLT style sheet is to load each into a DOM document and then use the `transformNode()` method. This method exists on every node in a document and accepts a single argument, which is the document containing an XSLT style

sheet. The `transformNode()` method returns a string containing the transformation. Here is an example:



```
//load the XML and XSLT (IE-specific)
xmldom.load("employees.xml");
xsltDom.load("employees.xslt");

//transform
var result = xmldom.transformNode(xsltDom);
```

[IEXsltExample01.htm](#)

This example loads a DOM document with XML and a DOM document with the XSLT style sheet. Then, `transformNode()` is called on the XML document node, passing in the XSLT. The variable `result` is then filled with a string resulting from the transformation. Note that the transformation began at the document node level, because that's where `transformNode()` was called. XSLT transformations can also take place anywhere in the document by calling `transformNode()` on the node at which you want the transformations to begin. Here is an example:

```
result = xmldom.documentElement.transformNode(xsltDom);
result = xmldom.documentElement.childNodes[1].transformNode(xsltDom);
result = xmldom.getElementsByTagName("name")[0].transformNode(xsltDom);
result = xmldom.documentElement.firstChild.lastChild.transformNode(xsltDom);
```

If you call `transformNode()` from anywhere other than the `document` element, you start the transformation at that spot. The XSLT style sheet, however, still has access to the full XML document from which that node came.

Complex XSLT Transformations

The `transformNode()` method gives basic XSLT transformation capabilities, but there are more complex ways to use the language. To do so, you must use an XSL template and an XSL processor. The first step is to load the XSLT style sheet into a thread-safe version of an XML document. This is done by using the `MSXML2.FreeThreadedDOMDocument` ActiveX object, which supports all of the same interfaces as a normal DOM document in Internet Explorer. This object needs to be created using the most up-to-date version as well. For example:

```
function createThreadSafeDocument(){
    if (typeof arguments.callee.activeXString != "string"){
        var versions = ["MSXML2.FreeThreadedDOMDocument.6.0",
                      "MSXML2.FreeThreadedDOMDocument.3.0",
                      "MSXML2.FreeThreadedDOMDocument"],
            i, len;

        for (i=0,len=versions.length; i < len; i++){
            try {
                new ActiveXObject(versions[i]);
                arguments.callee.activeXString = versions[i];
                break;
            } catch (ex){
```

```

        //skip
    }
}
}

return new ActiveXObject(arguments.callee.activeXString);
}

```

IEXsltExample02.htm

Aside from the different signature, using a thread-safe XML DOM document is the same as using the normal kind, as shown here:

```

var xsldom = createThreadSafeDocument();
xsldom.async = false;
xsldom.load("employees.xslt");

```

After the free-threaded DOM document is created and loaded, it must be assigned to an XSL template, which is another ActiveX object. The template is used to create an XSL processor object that can then be used to transform an XML document. Once again, the most appropriate version must be created, like this:



```

function createXSLTemplate(){
    if (typeof arguments.callee.activeXString != "string") {
        var versions = ["MSXML2.XSLTemplate.6.0",
                       "MSXML2.XSLTemplate.3.0",
                       "MSXML2.XSLTemplate"],
            i, len;

        for (i=0,len=versions.length; i < len; i++) {
            try {
                new ActiveXObject(versions[i]);
                arguments.callee.activeXString = versions[i];
                break;
            } catch (ex) {
                //skip
            }
        }
    }

    return new ActiveXObject(arguments.callee.activeXString);
}

```

IEXsltExample02.htm

You can use the `createXSLTemplate()` function to create the most recent version of the object, as in this example:

```

var template = createXSLTemplate();
template.stylesheet = xsldom;

var processor = template.createProcessor();

```

```

processor.input = xmldom;
processor.transform();

var result = processor.output;

```

[IEXsltExample02.htm](#)

When the XSL processor is created, the node to transform must be assigned to the `input` property. This value may be a document or any node within a document. The call to `transform()` executes the transformations and stores the result in the `output` property as a string. This code duplicates the functionality available with `transformNode()`.



There is a significant difference between the 3.0 and 6.0 versions of the XSL template object. In 3.0, the `input` property must be a complete document; using a node throws an error. In 6.0, you may use any node in a document.

Using the XSL processor allows extra control over the transformation and provides support for more advanced XSLT features. For example, XSLT style sheets accept parameters that can be passed in and used as local variables. Consider the following style sheet:



Available for download on
Wrox.com

```

<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="html" />

    <xsl:param name="message" />

    <xsl:template match="/">
        <ul>
            <xsl:apply-templates select="*" />
        </ul>
        <p>Message: <xsl:value-of select="$message" /></p>
    </xsl:template>

    <xsl:template match="employee">
        <li><xsl:value-of select="name" />, <em><xsl:value-of select="@title" /></em></li>
    </xsl:template>

</xsl:stylesheet>

```

[employees.xslt](#)

This style sheet defines a parameter named `message` and then outputs that parameter into the transformation result. To set the value of `message`, you use the `addParameter()` method before calling `transform()`. The `addParameter()` method takes two arguments: the name of the



```
processor.input = xmldom.documentElement;
processor.addParameter("message", "Hello World!");
processor.transform();
```

IEXsltExample03.htm

When you set a value for the parameter, the output will reflect the value.

Another advanced feature of the XSL processor is the capability to set a mode of operation. In XSLT, it's possible to define a mode for a template using the `mode` attribute. When a mode is defined, the template isn't run unless `<xsl:apply-templates>` is used with a matching `mode` attribute. Consider the following example:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="html" />

    <xsl:param name="message" />

    <xsl:template match="/">
        <ul>
            <xsl:apply-templates select="*" />
        </ul>
        <p>Message: <xsl:value-of select="$message" /></p>
    </xsl:template>

    <xsl:template match="employee">
        <li><xsl:value-of select="name" />, <em><xsl:value-of select="@title" /></em></li>
    </xsl:template>

    <xsl:template match="employee" mode="title-first">
        <li><em><xsl:value-of select="@title" /></em>, <xsl:value-of select="name" /></li>
    </xsl:template>
</xsl:stylesheet>
```

employees3.xslt

This style sheet defines a template with its `mode` attribute set to `"title-first"`. Inside of this template, the employee's title is output first, and the employee name is output second. In order to use this template, the `<xsl:apply-templates>` element must have its mode set to `"title-first"` as well. If you use this style sheet, it has the same output as the previous one by default, displaying the employee name first and the position second. If, however, you use this style sheet and set the mode to `"title-first"` using JavaScript, it outputs the employee's title first. This can be done in JavaScript using the `setStartMode()` method as shown here:



```
processor.input = xmldom;
processor.addParameter("message", "Hello World!");
processor.setStartMode("title-first");
processor.transform();
```

[IEXsltExample05.htm](#)

The `setStartMode()` method accepts only one argument, which is the mode to set the processor to. Just as with `addParameter()`, this must be called before `transform()`.

If you are going to do multiple transformations using the same style sheet, you can reset the processor after each transformation. When you call the `reset()` method, the input and output properties are cleared, as well as the start mode and any specified parameters. The syntax for this method is as follows:

```
processor.reset(); //prepare for another use
```

Because the processor has compiled the XSLT style sheet, it is faster to make repeat transformations versus using `transformNode()`.



MSXML supports only XSLT 1.0. Development on MSXML has stopped since Microsoft's focus has shifted to the .NET Framework. It is expected that, at some point in the future, JavaScript will have access to the XML and XSLT .NET objects.

The XSLTProcessor Type

Mozilla implemented JavaScript support for XSLT in Firefox by creating a new type. The `XSLTProcessor` type allows developers to transform XML documents by using XSLT in a manner similar to the XSL processor in Internet Explorer. Since it was first implemented, Chrome, Safari, and Opera have copied the implementation, making `XSLTProcessor` into a de facto standard for JavaScript-enabled XSLT transformations.

As with the Internet Explorer implementation, the first step is to load two DOM documents, one with the XML and the other with the XSLT. After that, create a new `XSLTProcessor` and use the `importStylesheet()` method to assign the XSLT to it, as shown in this example:

```
var processor = new XSLTProcessor();
processor.importStylesheet(xsltdom);
```

[XsltProcessorExample01.htm](#)

The last step is to perform the transformation. This can be done in two different ways. If you want to return a complete DOM document as the result, call `transformToDocument()`. You can also get a document fragment object as the result by calling `transformToFragment()`. Generally speaking,

the only reason to use `transformToFragment()` is if you intend to add the results to another DOM document.

When using `transformToDocument()`, just pass in the XML DOM and use the result as another completely different DOM. Here's an example:



Available for
download on
[Wrox.com](#)

```
var result = processor.transformToDocument(xmlDom);
alert(serializeXml(result));
```

[XsltProcessorExample01.htm](#)

The `transformToFragment()` method accepts two arguments: the XML DOM to transform and the document that should own the resulting fragment. This ensures that the new document fragment is valid in the destination document. You can, therefore, create the fragment and add it to the page by passing in `document` as the second argument. Consider the following example:

```
var fragment = processor.transformToFragment(xmlDom, document);
var div = document.getElementById("divResult");
div.appendChild(fragment);
```

[XsltProcessorExample02.htm](#)

Here, the processor creates a fragment owned by the `document` object. This enables the fragment to be added to a `<div>` element that exists in the page.

When the output format for an XSLT style sheet is either "xml" or "html", creating a document or document fragment makes perfect sense. When the output format is "text", however, you typically just want the text result of the transformation. Unfortunately, there is no method that returns text directly. Calling `transformToDocument()` when the output is "text" results in a full XML document being returned, but the contents of that document are different from browser to browser. Safari, for example, returns an entire HTML document, whereas Opera and Firefox return a one-element document with the output as the element's text.

The solution is to call `transformToFragment()`, which returns a document fragment that has a single child node containing the result text. You can, therefore, get the text by using the following code:

```
var fragment = processor.transformToFragment(xmlDom, document);
var text = fragment.firstChild.nodeValue;
alert(text);
```

This code works the same way for each of the supporting browsers and correctly returns just the text output from the transformation.

Using Parameters

The `XSLTProcessor` also allows you to set XSLT parameters using the `setParameter()` method, which accepts three arguments: a namespace URI, the parameter local name, and the value to set. Typically, the namespace URI is `null`, and the local name is simply the parameter's name. This

method must be called prior to `transformToDocument()` or `transformToFragment()`. Here's an example:



Available for download on
Wrox.com

```
var processor = new XSLTProcessor();
processor.importStylesheet(xsltdom);
processor.setParameter(null, "message", "Hello World!");
var result = processor.transformToDocument(xmlDom);
```

[XsltProcessorExample03.htm](#)

There are two other methods related to parameters, `getParameter()` and `removeParameter()`, which are used to get the current value of a parameter and remove the parameter value, respectively. Each method takes the namespace URI (once again, typically `null`) and the local name of the parameter. For example:

```
var processor = new XSLTProcessor();
processor.importStylesheet(xsltdom);
processor.setParameter(null, "message", "Hello World!");

alert(processor.getParameter(null, "message")); //outputs "Hello World!"
processor.removeParameter(null, "message");

var result = processor.transformToDocument(xmlDom);
```

These methods aren't used often and are provided mostly for convenience.

Resetting the Processor

Each `XSLTProcessor` instance can be reused multiple times for multiple transformations with different XSLT style sheets. The `reset()` method removes all parameters and style sheets from the processor, allowing you to once again call `importStylesheet()` to load a different XSLT style sheet, as in this example:

```
var processor = new XSLTProcessor();
processor.importStylesheet(xsltdom);

//do some transformations

processor.reset();
processor.importStylesheet(xsltdom2);

//do more transformations
```

Reusing a single `XSLTProcessor` saves memory when using multiple style sheets to perform transformations.

Cross-Browser XSLT

The Internet Explorer XSLT transformation is quite different from the `XSLTProcessor` approach, so recreating all of the functionality available in each is not realistic. The easiest cross-browser

technique for XSLT transformations is to return a string result. For Internet Explorer, this means simply calling `transformNode()` on the context node, whereas other browsers need to serialize the result of a `transformToDocument()` operation. The following function can be used in Internet Explorer, Firefox, Chrome, Safari, and Opera:



```
function transform(context, xslt){
    if (typeof XSLTProcessor != "undefined"){
        var processor = new XSLTProcessor();
        processor.importStylesheet(xslt);

        var result = processor.transformToDocument(context);
        return (new XMLSerializer()).serializeToString(result);

    } else if (typeof context.transformNode != "undefined") {
        return context.transformNode(xslt);
    } else {
        throw new Error("No XSLT processor available.");
    }
}
```

CrossBrowserXsltExample01.htm

The `transform()` function accepts two arguments: the context node on which to perform the transformation and the XSLT document object. First, the code checks to see if the `XSLTProcessor` type is defined, and if so, it uses that to process the transformation. The `transformToDocument()` method is called and the result is serialized into a string to be returned. If the context node has a `transformNode()` method, then that is used to return the result. As with the other cross-browser functions in this chapter, `transform()` throws an error if there is no XSLT processor available. This function is used as follows:

```
var result = transform(xmlDom, xsltDom);
```

Using the Internet Explorer `transformNode()` method ensures that you don't need to use a thread-safe DOM document for the transformation.



Note that because of different XSLT engines in browsers, the results you receive from a transformation may vary slightly or greatly from browser to browser. You should never depend on an absolute transformation result using XSLT in JavaScript.

SUMMARY

There is a great deal of support for XML and related technologies in JavaScript. Unfortunately, because of an early lack of specifications, there are several different implementations for common functionality. DOM Level 2 provides an API for creating empty XML documents but not for

parsing or serialization. Because of this lack of functionality, browser vendors began creating their own approaches. Internet Explorer took the following approach:

- Internet Explorer introduced XML support through ActiveX objects, the same objects that could be used to build desktop applications.
- The MSXML library ships with Windows and is accessible from JavaScript.
- This library includes support for basic XML parsing and serialization and for complementary technologies such as XPath and XSLT.

Firefox, on the other hand, implemented two new types to deal with XML parsing and serialization as follows:

- The `DOMParser` type is a simple object that parses an XML string into a DOM document.
- The `XMLSerializer` type performs the opposite operation, serializing a DOM document into an XML string.

Because of the simplicity and popularity of these objects, Internet Explorer 9, Opera, Chrome, and Safari duplicated the functionality, and these types are de facto standards in web development.

DOM Level 3 introduced a specification for an XPath API that has been implemented by Firefox, Safari, Chrome, and Opera. The API enables JavaScript to run any XPath query against a DOM document and retrieve the result regardless of its data type. Internet Explorer implemented its own XPath support in the form of two methods: `selectSingleNode()` and `selectNodes()`. Although much more limited than the DOM Level 3 API, these methods provide basic XPath functionality to locate a node or set of nodes in a DOM document.

The last related technology is XSLT, which has no public specification defining an API for its usage. Firefox created the `XSLTProcessor` type to handle transformations via JavaScript and was soon copied by Safari, Chrome, and Opera. Internet Explorer implemented its own solution, both with the simple `transformNode()` method and through a more complicated template/processor approach.

XML is now well supported in Internet Explorer, Firefox, Chrome, Safari, and Opera. Even though the implementations vary wildly between Internet Explorer and the other browsers, there's enough commonality to create reasonable cross-browser functionality.

19

ECMAScript for XML

WHAT'S IN THIS CHAPTER?

- ▶ Additional types introduced by E4X
- ▶ Using E4X for XML manipulation
- ▶ Syntax changes

In 2002, a group of companies led by BEA Systems proposed an extension to ECMAScript to add native XML support to the language. In June 2004, ECMAScript for XML (E4X) was released as ECMA-357, which was revised in December 2005. E4X is not its own language; rather, it is an optional extension to the ECMAScript language. As such, E4X introduces new syntax for dealing with XML, as well as for XML-specific objects.

Though browser adoption has been slow, Firefox versions 1.5 and later support almost the entire E4X standard. This chapter focuses on the Firefox implementation.

E4X TYPES

As an extension to ECMAScript, E4X introduces the following new global types:

- ▶ `XML` — Any single part of an XML structure
- ▶ `XMLElement` — A collection of XML objects
- ▶ `Namespace` — Mapping between a namespace prefix and a namespace URI
- ▶ `QName` — A qualified name made up of a local name and a namespace URI

Using these four types, E4X is capable of representing all parts of an XML document by mapping each type, specifically `XML` and `XMLElement`, to multiple DOM types.

The XML Type

The `XML` type is the most important new type introduced in E4X, because it can represent any single part of an XML structure. An instance of `XML` can represent an element, an attribute, a comment, a processing instruction, or a text node. The `XML` type inherits from the `Object` type, so it inherits all of the default properties and methods of all objects. There are a few ways to create a new `XML` object, the first of which is to call the constructor like this:

```
var x = new XML();
```

This line creates an empty `XML` object that can be filled with data. You can also pass in an XML string to the constructor, as shown in this example:

```
var x = new XML("<employee position=\"Software Engineer\"><name>Nicholas " +
    "Zakas</name></employee>");
```

The XML string passed into the constructor is parsed into an object hierarchy of `XML` objects. Additionally, you can pass in a DOM document or node to the constructor as follows and have its data represented in E4X:

```
var x = new XML(xmlDom);
```

Even though these methods of construction are useful, the most powerful and interesting method is direct assignment of XML data into a variable via an *XML literal*. XML literals are nothing more than XML code embedded into JavaScript code. Here's an example:



```
var employee = <employee position="Software Engineer">
    <name>Nicholas C. Zakas</name>
</employee>;
```

[XMLTypeExample01.htm](#)

Here, an XML data structure is assigned directly to a variable. This augmented syntax creates an `XML` object and assigns it to the `employee` variable.



The Firefox implementation of E4X doesn't support the parsing of XML prologs. If <?xml version="1.0"?> is present, either in text that is passed to the XML constructor or as part of the XML literal, a syntax error occurs.

The `toXMLString()` method returns an XML string representing the object and its children. The `toString()` method, on the other hand, behaves differently based on the contents of the `XML` object. If the contents are simple (plain text), then the text is returned; otherwise `toString()` acts the same as `toXMLString()`. Consider the following example:

```
var data = <name>Nicholas C. Zakas</name>;
alert(data.toString());      // "Nicholas C. Zakas"
alert(data.toXMLString());  // "<name>Nicholas C. Zakas</name>"
```

Between these two methods, most XML serialization needs can be met.

The XMLList Type

The `XMLList` type represents ordered collections of XML objects. The DOM equivalent of `XMLList` is `NodeList`, although the differences between `XML` and `XMLList` are intentionally small as compared to the differences between `Node` and `NodeList`. To create a new `XMLList` object explicitly, you can use the following `XMLList` constructor:

```
var list = new XMLList();
```

As with the `XML` constructor, you can pass in an XML string to be parsed. The string need not contain a single document element, as the following code illustrates:

```
var list = new XMLList("<item/><item/>");
```

[XMLListTypeExample01.htm](#)

Here the `list` variable is filled with an `XMLList` containing two `XML` objects, one for each `<item/>` element.

An `XMLList` object also can be created by combining two or more `XML` objects via the plus (+) operator. This operator has been overloaded in E4X to perform `XMLList` creation, as in this example:

```
var list = <item/> + <item/>;
```

This example combines two XML literals into an `XMLList` by using the plus operator. The same can be accomplished by using the special `<>` and `</>` syntax and omitting the plus operator, as shown here:

```
var list = <><item/><item/></>;
```

Although it's possible to create stand-alone `XMLList` objects, they are typically created as part of parsing a larger XML structure. Consider the following:



```
var employees = <employees>
    <employee position="Software Engineer">
        <name>Nicholas C. Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>;
```

[XMLListTypeExample02.htm](#)

This code defines an `employees` variable that becomes an `XML` object representing the `<employees/>` element. Since there are two child `<employee/>` elements, an `XMLElement` object is created and stored in `employees.employee`. It's then possible to access each element using bracket notation and its position like this:

```
var firstEmployee = employees.employee[0];
var secondEmployee = employees.employee[1];
```

Each `XMLElement` object also has a `length()` method that returns the number of items it contains. For example:

```
alert(employees.employee.length()); //2
```

Note that `length()` is a method, not a property. This is intentionally different from arrays and `NodeLists`.

One interesting part of E4X is the intentional blurring of the `XML` and `XMLElement` types. In fact, there is no discernible difference between an `XML` object and an `XMLElement` containing a single `XML` object. To minimize these differences, each `XML` object also has a `length()` method and a property referenced by `[0]` (which returns the `XML` object itself).

The compatibilities between `XML` and `XMLElement` allow for much easier E4X usage, because some methods may return either type.

The `toString()` and `toXMLString()` methods for an `XMLElement` object return the same string value, which is a concatenated serialization of all `XML` objects it contains.

The Namespace Type

Namespaces are represented in E4X by `Namespace` objects. A `Namespace` object generally is used to map a namespace prefix to a namespace URI, although a prefix is not always necessary. You can create a `Namespace` object by using the `Namespace` constructor as follows:

```
var ns = new Namespace();
```

You can also initialize a `Namespace` object with either a URI or a prefix and URI, as shown here:



```
var ns = new Namespace("http://www.wrox.com/"); //no prefix namespace
var wrox = new Namespace("wrox", "http://www.wrox.com/"); //wrox namespace
```

[NamespaceTypeExample01.htm](#)

The information in a `Namespace` object can be retrieved using the `prefix` and `uri` properties like this:

```
alert(ns.uri);           //"http://www.wrox.com/"
alert(ns.prefix);        //undefined
alert(wrox.uri);         //"http://www.wrox.com/"
alert(wrox.prefix);       // "wrox"
```

[NamespaceTypeExample01.htm](#)

Whenever a prefix isn't assigned as part of the Namespace object, its `prefix` property is set to `undefined`. To create a default namespace, you should set the prefix to an empty string.

If an XML literal contains a namespace or if an XML string containing namespace information is parsed via the `XML` constructor, a Namespace object is created automatically. You can then retrieve a reference to the Namespace object by using the `namespace()` method and specifying the prefix. Consider the following example:



Available for download on
Wrox.com

```
var xml = <wrox:root xmlns:wrox="http://www.wrox.com/">
    <wrox:message>Hello World!</wrox:message>
</wrox:root>

var wrox = xml.namespace("wrox");
alert(wrox.uri);
alert(wrox.prefix);
```

[NamespaceTypeExample02.htm](#)

In this example, an XML fragment containing a namespace is created as an XML literal. The Namespace object from the `wrox` namespace can be retrieved via `namespace ("wrox")`, after which point you can access the `uri` and `prefix` properties. If the XML fragment has a default namespace, that can be retrieved by passing an empty string into the `namespace()` method.

The `toString()` method for a Namespace object always returns the namespace URI.

The QName Type

The `QName` type represents qualified names of XML objects, which are the combination of a namespace and a local name. You can create a new `QName` object manually using the `QName` constructor and passing in either a name or a Namespace object and a name, as shown here:

```
var wrox = new Namespace("wrox", "http://www.wrox.com/");
var wroxMessage = new QName(wrox, "message"); //represents "wrox:message"
```

[QNameTypeExample01.htm](#)

After the object is created, it has two properties that can be accessed: `uri` and `localName`. The `uri` property returns the URI of the namespace specified when the object is created (or an empty string if no namespace is specified), and the `localName` property returns the local name part of the qualified name, as the following example shows:

```
alert(wroxMessage.uri); // "http://www.wrox.com/"
alert(wroxMessage.localName); // "message"
```

[QNameTypeExample01.htm](#)

These properties are read-only and cause an error if you try to change their values. The `QName` object overrides the `toString()` object to return a string in the form `uri::localName`, such as "`http://www.wrox.com/:message`" in the previous example.



```
var xml = <wrox:root xmlns:wrox="http://www.wrox.com/">
    <wrox:message>Hello World!</wrox:message>
</wrox:root>

var wroxRoot = xml.name();
alert(wroxRoot.uri);           // "http://www.wrox.com/"
alert(wroxRoot.localName);     // "root"
```

QNameTypeExample02.htm

A `QName` object is created for each element and attribute in an XML structure even when no namespace information is specified.

You can change the qualified name of an `XML` object by using the `setName()` method and passing in a new `QName` object, as shown here:

```
xml.setName(new QName("newroot"));
```

This method typically is used when changing the tag name of an element or an attribute name that is part of a namespace. If the name isn't part of a namespace, you can change the local name by simply using the `setLocalName()` method like this:

```
xml.setLocalName("newtagname");
```

GENERAL USAGE

When an XML object, elements, attributes, and text are assembled into an object hierarchy, you can then navigate the structure by using dot notation along with attribute and tag names. Each child element is represented as a property of its parent, with the property name being equal to the child element's local name. If that child element contains only text, then it is returned whenever the property is accessed, as in the following example:

```
var employee = <employee position="Software Engineer">
    <name>Nicholas C. Zakas</name>
</employee>;
alert(employee.name); // "Nicholas C. Zakas"
```

The `<name/>` element in this code contains only text. That text is retrieved via `employee.name`, which navigates to the `<name/>` element and returns it. Since the `toString()` method is called implicitly when passed into an alert, the text contained within `<name/>` is displayed. This ability makes it trivial to access text data contained within an XML document. If there's more than one element with the same tag name, an `XMLElement` is returned. Consider the following example:

```

var employees = <employees>
    <employee position="Software Engineer">
        <name>Nicholas C. Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>

alert(employees.employee[0].name);      //Nicholas C. Zakas"
alert(employees.employee[1].name);      //Jim Smith"

```

This example accesses each `<employee>` element and outputs the value of their `<name>` elements. If you aren't sure of a child element's local name or if you want to retrieve all child elements regardless of their name, you can use an asterisk (*), as shown here:



Available for download on
Wrox.com

```

var allChildren = employees.*;           //return all children regardless of local name
alert(employees.*[0].name);            //"Nicholas C. Zakas"

```

[UsageExample01.htm](#)

As with other properties, the asterisk may return either a single XML object or an XMMLList object, depending on the XML structure.

The `child()` method behaves in the exact same way as property access. Any property name or index can be passed into the `child()` method and it will return the same value. Consider this example:

```

var firstChild = employees.child(0);          //same as employees.*[0]
var employeeList = employees.child("employee"); //same as employees.employee
var allChildren = employees.child("*");         //same as employees.*

```

For added convenience, a `children()` method is provided that always returns all child elements. Here's an example:

```
var allChildren = employees.children(); //same as employees.*
```

There is also an `elements()` method, which behaves similar to `child()` with the exception that it will return only XML objects that represent elements. For example:

```

var employeeList = employees.elements("employee"); //same as employees.employee
var allChildren = employees.elements("*");           //same as employees.*

```

These methods provide a more familiar syntax for JavaScript developers to access XML data.

Child elements can be removed by using the `delete` operator, as shown here:

```

delete employees.employee[0];
alert(employees.employee.length()); //1

```

This is one of the major advantages of treating child nodes as properties.

Accessing Attributes

Attributes can also be accessed using dot notation, although the syntax is slightly augmented. To differentiate an attribute name from a child-element tag name, you must prepend an “at” character (@) before the name. This syntax is borrowed from XPath, which also uses @ to differentiate between attributes and character names. The result is a syntax that looks a little strange, as you can see in this example:



```
var employees = <employees>
    <employee position="Software Engineer">
        <name>Nicholas C. Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>

alert(employees.employee[0].@position); // "Software Engineer"
```

AttributesExample01.htm

As with elements, each attribute is represented as a property that can be accessed using this shorthand notation. An `XML` object representing the attribute is returned and its `toString()` method always returns the attribute value. To get the attribute name, use the `name()` method of the object.

You can also use the `child()` method to access an attribute by passing in the name of the attribute prefixed with `@`, as shown here:

```
alert(employees.employee[0].child("@position")); // "Software Engineer"
```

AttributesExample01.htm

Since any `XML` object property name can be used with `child()`, the `@` character is necessary to distinguish between tag names and attribute names.

It’s possible to access only attributes by using the `attribute()` method and passing in the name of the attribute. Unlike `child()`, there is no need to prefix the attribute name with an `@` character. Here’s an example:

```
alert(employees.employee[0].attribute("position")); // "Software Engineer"
```

AttributesExample01.htm

These three ways of accessing properties are available on both `XML` and `XMLElement` types. When used on an `XML` object, an `XML` object representing the attribute is returned; when used on an `XMLElement` object, an `XMLElement` is returned containing attribute `XML` objects for all elements in the list. In the previous example, for instance, `employees.employee.@position` will return an `XMLElement`

containing two objects — one for the `position` attribute on the first `<employee/>` element and one for the second.

To retrieve all attributes in an `XML` or `XMLElement` object, you can use the `attributes()` method. This method returns an `XMLElement` of objects representing all attributes. This is the same as using the `@*` pattern, as illustrated in this example:

```
//both lines get all attributes
var atts1 = employees.employee[0].@*;
var atts2 = employees.employee[0].attributes();
```

Changing attribute values in E4X is as simple as changing a property value. Simply assign a new value to the property like this:

```
employees.employee[0].@position = "Author"; //change position attribute
```

The change is then reflected internally, so when you serialize the `XML` object, the attribute value is updated. This same technique can be used to add new attributes, as shown in the following example:

```
employees.employee[0].@experience = "8 years"; //add experience attribute
employees.employee[0].@manager = "Jim Smith"; //add manager attribute
```

Since attributes act like any other ECMAScript properties, you can also remove attributes by using the `delete` operator as follows:

```
delete employees.employee[0].@position; //delete position attribute
```

Property access for attributes allows for very simple interaction with the underlying XML structure.

Other Node Types

E4X is capable of representing all parts of an XML document, including comments and processing instructions. By default, E4X will not parse comments or processing instructions, so they won't show up in the object hierarchy. To force the parser to recognize them, you must set the following two properties on the `XML` constructor:

```
XML.ignoreComments = false;
XML.ignoreProcessingInstructions = false;
```

With these flags set, E4X parses comments and processing instructions into the XML structure.

Since the `XML` type represents all of the node types, it's necessary to have a way to tell them apart. The `nodeKind()` method indicates what type of node an `XML` object represents and returns `"text"`, `"element"`, `"comment"`, `"processing-instruction"`, or `"attribute"`. Consider the following `XML` object:



```
var employees = <employees>
    <?Dont forget the donuts?>
    <employee position="Software Engineer">
        <name>Nicholas C. Zakas</name>
    </employee>
    <!-- just added -->
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>;
```

Given this XML, the following table shows what `nodeKind()` returns, depending on which node is in scope.

STATEMENT	RETURNS
<code>employees.nodeKind()</code>	"element"
<code>employees.*[0].nodeKind()</code>	"processing-instruction"
<code>employees.employee[0].@position.nodeKind()</code>	"attribute"
<code>employees.employee[0].nodeKind()</code>	"element"
<code>employees.*[2].nodeKind()</code>	"comment"
<code>employees.employee[0].name.*[0].nodeKind()</code>	"text"

The `nodeKind()` method can't be called on an `XMLElement` that has more than one `XML` object in it; doing so throws an error.

It's possible to retrieve just the nodes of a particular type by using one of the following methods:

- `attributes()` — Returns all the attributes of an `XML` object.
- `comments()` — Returns all the child comments of an `XML` object.
- `elements(tagName)` — Returns all the child elements of an `XML` object. You can filter the results by providing the `tagName` of the elements you want to return.
- `processingInstructions(name)` — Returns all the child processing instructions of an `XML` object. You can filter the results by providing the `name` of the processing instructions to return.
- `text()` — Returns all text node children of an `XML` object.

Each of these methods returns an `XMLElement` containing the appropriate `XML` objects.

You can determine if an `XML` object contains just text or more complex content by using the `hasSimpleContent()` and `hasComplexContent()` methods. The former returns `true` if there are only text nodes as children, whereas the latter returns `true` if there are any child nodes that aren't text nodes. Here's an example:

```
alert(employees.employee[0].hasComplexContent()); //true
alert(employees.employee[0].hasSimpleContent()); //false
```

```
alert(employees.employee[0].name.hasComplexContent()); //false
alert(employees.employee[0].name.hasSimpleContent()); //true
```

These methods, used in conjunction with the others, can aid in the querying of an XML structure for relevant data.

Querying

In truth, E4X provides a querying syntax that is similar in many ways to XPath. The simple act of retrieving element or attribute values is a basic type of query. The XML objects that represent various parts of an XML structure aren't created until a query is made. In effect, all properties of XML or XMLList objects are simply parts of a query. This means referencing a property that doesn't represent a part of the XML structure still returns an XMLList, it just has nothing in it. For example, if the following code is run against the previous XML example, nothing will be returned:



Available for
download on
[Wrox.com](#)

```
var cats = employees.cat;
alert(cats.length()); //0
```

[QueryingExample01.htm](#)

This query looks for `<cat/>` elements under `<employees/>`. The first line returns an XMLList with nothing in it. Such behavior allows for querying without worrying about exceptions occurring.

Most of the previous example dealt with direct children of nodes using dot notation. You can expand the query to all descendants by using two dots, as shown here:

```
var allDescendants = employees..*; //get all descendants of <employees/>
```

In this code, all descendants of the `<employees/>` element are returned. The results are limited to elements, text, comments, and processing instructions, with the latter two included based only on the flags specified on the XML constructor (discussed in the preceding section); attributes will not be included. To retrieve only elements of a specific tag name, replace the asterisk with the actual tag name as follows:

```
var allNames = employees..name; //get all <name/> descendants of <employees/>
```

The same queries can be executed using the `descendants()` method. When used without any arguments, this method returns all descendants (which is the same as using `..*`), or you can also supply a name as an argument to limit the results. Here are examples of both:

```
var allDescendants = employees.descendants(); //all descendants
var allNames = employees.descendants("name"); //all <name/> descendants
```

It is possible to retrieve all attributes of all descendants using either of the following:

```
var allAttributes = employees..@*; //get all attributes on descendants
var allAttributes2 = employees.descendants("@*"); //same
```

As with element descendants, you can limit the results by supplying a full attribute name instead of an asterisk. For example:

```
var allAttributes = employees..@position;           //get all position attributes
var allAttributes2 = employees.descendants("@position"); //same
```

In addition to accessing descendants, you can specify a condition that must be met. For instance, to return all `<employee>` elements where the `position` attribute is "salesperson", you can use the following query:

```
var salespeople = employees.employee.(@position == "Salesperson");
```

This syntax can also be used to change parts of the XML structure. For example, you can change the `position` attribute of the first salesperson to "Senior Salesperson" with just the following line:

```
employees.employee.(@position == "Salesperson") [0].@position= "Senior Salesperson";
```

Note that the expression in parentheses returns an `XMLElement` containing the results, so the brackets return the first item upon which the `@position` property is written.

You can travel back up the XML structure by using the `parent()` method, which returns the `XML` object representing the `XML` object's parent. If called on an `XMLElement`, the `parent()` method returns the common parent of all objects in the list. Consider this example:

```
var employees2 = employees.employee.parent();
```

Here, the `employees2` variable contains the same value as the `employees` variable. The `parent()` method is most useful when dealing with an `XML` object of unknown origin.

XML Construction and Manipulation

There are numerous options for getting XML data into an `XML` object. As discussed earlier, you can pass in an XML string to the `XML` constructor or use an XML literal. XML literals can be made more useful by embedding JavaScript variables within curly braces: `{}`. A variable can be used anywhere within an XML literal, such as in this example:



```
var tagName = "color";
var color = "red";
var xml = <{tagName}>{color}</{tagName}>

alert(xml.toXMLString());      //"<color>red</color>"
```

XMLConstructionExample01.htm

In this code, both the tag name and the text value of the XML literal are specified using variables inserted with curly braces. This capability makes it easy to build up XML structures without string concatenation.

E4X also makes it easy to build up an entire XML structure using standard JavaScript syntax. As mentioned previously, most operations are queries and won't throw an error even if the elements or attributes don't exist. Taking that one step further, if you assign a value to a nonexistent element or attribute, E4X will create the underlying structure first and then do the assignment. Here's an example:

```
var employees = <employees/>;
employees.employee.name = "Nicholas C. Zakas";
employees.employee.@position = "Software Engineer";
```

[XMLConstructionExample02.htm](#)

This example begins with an `<employees/>` element and then builds on it. The second line creates an `<employee/>` element and a `<name/>` element inside it, assigning a text value. The next line adds the `position` attribute and assigns a value to it. In the end, the structure is as follows:

```
<employees>
  <employee position="Software Engineer">
    <name>Nicholas C. Zakas</name>
  </employee>
</employees>
```

It's then possible to add a second `<employee/>` element using the `+` operator, like this:



Available for
download on
Wrox.com

```
employees.employee += <employee position="Salesperson">
  <name>Jim Smith</name>
</employee>;
```

[XMLConstructionExample02.htm](#)

This results in a final XML structure, as follows:

```
<employees>
  <employee position="Software Engineer">
    <name>Nicholas C. Zakas</name>
  </employee>
  <employee position="Salesperson">
    <name>Jim Smith</name>
  </employee>
</employees>
```

Aside from this basic XML construction syntax, the following DOM-like methods are also available:

- `appendChild(child)` — Appends the given `child` to the end of the `XMLLIST` representing the node's children.
- `copy()` — Returns a duplicate of the `XML` object.
- `insertChildAfter(refNode, child)` — Inserts `child` after `refNode` in the `XMLLIST` representing the node's children.

- `insertChildBefore(refNode, child)` — Inserts `child` before `refNode` in the `XMLEList` representing the node's children.
- `prependChild(child)` — Inserts the given `child` at the beginning of the `XMLEList` representing the node's children.
- `replace(propertyName, value)` — Replaces the property named `propertyName`, which may be an element or an attribute, with the given `value`.
- `setChildren(children)` — Replaces all current children with `children`, which may be an XML object or an `XMLEList` object.

These methods are incredibly useful and easy to use. The following code illustrates some of these methods:



```
var employees = <employees>
    <employee position="Software Engineer">
        <name>Nicholas C. Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>

employees.appendChild(<employee position="Vice President">
    <name>Benjamin Anderson</name>
</employee>);

employees.prependChild(<employee position="User Interface Designer">
    <name>Michael Johnson</name>
</employee>);

employees.insertChildBefore(employees.child(2),
    <employee position="Human Resources Manager">
        <name>Margaret Jones</name>
    </employee>);

employees.setChildren(<employee position="President">
    <name>Richard McMichael</name>
</employee> +
<employee position="Vice President">
    <name>Rebecca Smith</name>
</employee>);
```

[XMLConstructionExample03.htm](#)

First, the code adds a vice president named Benjamin Anderson to the bottom of the list of employees. Second, a user interface designer named Michael Johnson is added to the top of the list of employees. Third, a human resources manager named Margaret Jones is added just before the employee in position 2, which at this point is Jim Smith (because Michael Johnson and Nicholas C. Zakas now come before him). Finally, all the children are replaced with the president, Richard McMichael, and the new vice president, Rebecca Smith. The resulting XML looks like this:

```
<employees>
    <employee position="President">
        <name>Richard McMichael</name>
    </employee>
    <employee position="Vice President">
        <name>Rebecca Smith</name>
    </employee>
</employees>
```

Using these techniques and methods, you can perform any DOM-style operation using E4X.

Parsing and Serialization Options

The way E4X parses and serializes data is controlled by several settings on the `XML` constructor. The following three settings are related to XML parsing:

- `ignoreComments` — Indicates that the parser should ignore comments in the markup. This is set to `true` by default.
- `ignoreProcessingInstructions` — Indicates that the parser should ignore processing instructions in the markup. This is set to `true` by default.
- `ignoreWhitespace` — Indicates that the parser should ignore white space in between elements rather than create text nodes to represent it. This is set to `true` by default.

These three settings affect parsing of XML strings passed into the `XML` constructor, as well as XML literals.

Additionally, the following two settings are related to the serialization of XML data:

- `prettyIndent` — Indicates the number of spaces used per indent when serializing XML. The default is 2.
- `prettyPrinting` — Indicates that the XML should be output in a human-readable format, with each element on a new line and children indented. This is set to `true` by default.

These settings affect the output from `toString()` and `toXMLString()`.

All five of the settings are stored in a `settings` object that can be retrieved using the `settings()` method of the `XML` constructor, as in this example:



```
var settings = XML.settings();
alert(settings.ignoreWhitespace);      //true
alert(settings.ignoreComments);       //true
```

[ParsingAndSerializationExample01.htm](#)

Multiple settings can be assigned at once by passing an object into the `setSettings()` method containing all five settings. This is useful when you want to change settings temporarily, as in the following example:

```
var settings = XML.settings();
XML.prettyIndent = 8;
```

```
XML.ignoreComments = false;  
  
//do some processing  
  
XML.setSettings(settings); //reset to previous settings
```

You can always get an object containing the default settings by using the `defaultSettings()` method, so you can reset the settings at any time using the following line:

```
XML.setSettings(XML.defaultSettings());
```

Namespaces

E4X makes namespaces quite easy to use. As discussed previously, you can retrieve a Namespace object for a particular prefix using the `namespace()` method. You can also set the namespace for a given element by using the `setNamespace()` method and passing in a Namespace object. Here's an example:

```
var messages = <messages>  
    <message>Hello world!</message>  
</messages>;  
messages.setNamespace(new Namespace("wrox", "http://www.wrox.com/"));
```

When `setNamespace()` is called, the namespace gets applied to only the element on which it was called. Serializing the `messages` variable results in the following:

```
<wrox:messages xmlns:wrox="http://www.wrox.com/">  
    <message>Hello world!</message>  
</wrox:messages>
```

The `<messages/>` element gets prefixed with the `wrox` namespace because of the call to `setNamespace()`, whereas the `<message/>` element remains unchanged.

To simply add a namespace declaration without changing the element, use the `addNamespace()` method and pass in a Namespace object, as in this example:

```
messages.addNamespace(new Namespace("wrox", "http://www.wrox.com/"));
```

When this code is applied to the original `messages` XML, the following XML structure is created:

```
<messages xmlns:wrox="http://www.wrox.com/">  
    <message>Hello world!</message>  
</messages>
```

By calling `removeNamespace()` and passing in a Namespace object, you can remove the namespace declaration for the namespace with the given namespace prefix and URI; it is not necessary to pass in the exact Namespace object representing the namespace. Consider this example:

```
messages.removeNamespace(new Namespace("wrox", "http://www.wrox.com/"));
```

This code removes the `wrox` namespace. Note that qualified names referencing the prefix will not be affected.

There are two methods that return an array of the `Namespace` object related to a node. The first is `namespaceDeclarations()`, which returns an array of all namespaces that are declared on the given node. The second is `inScopeNamespaces()`, which returns an array of all namespaces that are in the scope of the given node, meaning they have been declared either on the node itself or on an ancestor node. Consider this example:

```
var messages = <messages xmlns:wrox="http://www.wrox.com/">
    <message>Hello world!</message>
</messages>

alert(messages.namespaceDeclarations());      // "http://www.wrox.com"
alert(messages.inScopeNamespaces());          // ",http://www.wrox.com"

alert(messages.message.namespaceDeclarations()); // ""
alert(messages.message.inScopeNamespaces());    // ",http://www.wrox.com"
```

Here, the `<message/>` element returns an array containing one namespace when `namespaceDeclarations()` is called, and an array with two namespaces when `inScopeNamespaces()` is called. The two in-scope namespaces are the default namespace (represented by an empty string) and the `wrox` namespace. When these methods are called on the `<message/>` element, `namespaceDeclarations()` returns an empty array, whereas `inScopeNamespaces()` returns the same results.

A `Namespace` object can also be used to query an XML structure for elements in a specific namespace by using the double colon (`::`). For example, to retrieve all `<message/>` elements contained in the `wrox` namespace, you could use the following:

```
var messages = <messages xmlns:wrox="http://www.wrox.com/">
    <wrox:message>Hello world!</wrox:message>
</messages>;
var wroxNS = new Namespace("wrox", "http://www.wrox.com/");
var wroxMessages = messages.wroxNS::message;
```

The double colon indicates the namespace in which the element to be returned should exist. Note that it is the name of the JavaScript variable that is used, not the namespace prefix.

You can also set the default namespace for all `XML` objects created within a given scope. To do so, use the `default xml namespace` statement and assign either a `Namespace` object or simply a namespace URI. Here's an example:

```
default xml namespace = "http://www.wrox.com/";

function doSomething(){

    //set default namespace just for this function
    default xml namespace = new Namespace("your", "http://www.yourdomain.com");

}
```

The default XML namespace for the global scope is not set. This statement is useful when all XML data within a given scope will be using a specific namespace, avoiding constant references to the namespace itself.

OTHER CHANGES

To work seamlessly with standard ECMAScript, E4X makes some changes to the base language. One change is the introduction of the `for-each-in` loop. As opposed to the `for-in` loop, which iterates over each property and returns the property name, the `for-each-in` loop iterates over each property and returns the value of the property, as this example illustrates:



```
var employees = <employees>
    <employee position="Software Engineer">
        <name>Nicholas C. Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>

for each (var child in employees) {
    alert(child.toXMLString());
}
```

ForEachInExample01.htm

The `for-each-in` loop in this example fills the `child` variable with each child node of `<employees/>`, which may include comments, processing instructions, and/or text nodes. Attributes aren't returned in the loop unless you use an `XMLEList` of attributes, such as the following:

```
for each (var attribute in employees.@*){ //iterate over attributes
    alert(attribute);
}
```

Even though the `for-each-in` loop is defined as part of E4X, it can also be used on normal arrays and objects, as shown here:

```
var colors = ["red", "green", "blue"];
for each(var color in colors){
    alert(color);
}
```

ForEachInExample01.htm

For arrays, the `for-each-in` loop returns each array item. For non-`XML` objects, it returns the value of each property.

E4X also adds a global function called `isXMLName()` that accepts a string and returns `true` if the name is a valid local name for an element or attribute. This is provided as a convenience to developers who may be using unknown string data to construct XML data structures. Here's an example:

```
alert(isXMLName("color"));      //true
alert(isXMLName("hello world")); //false
```

If you are unsure of the origin of a string that should be used as a local name, it's best to use `isXMLName()` first to determine if the string is valid or will cause an error.

The last change to standard ECMAScript is to the `typeof` operator. When used on an `XML` object or an `XMLElement` object, `typeof` returns the string "`xml`". This differs from when it is used on other objects, in which case it returns "`object`", as shown here:

```
var xml = new XML();
var list = new XMLElement();
var object = {};

alert(typeof xml);      //"xml"
alert(typeof list);     //"xml"
alert(typeof object);   //"object"
```

In most cases, it is unnecessary to distinguish between `XML` and `XMLElement` objects. Since both types are considered primitives in E4X, you cannot use the `instanceof` operator to make this distinction either.

ENABLING FULL E4X

Because E4X does many things differently than standard JavaScript, Firefox enables only the parts that work best when E4X is intermixed with other code. To fully enable E4X, you need to set the `type` attribute of the `<script>` tag to "`text/javascript;e4x=1`", as in this example:

```
<script type="text/javascript;e4x=1" src="e4x_file.js"></script>
```

When this switch is turned on, full E4X support is enabled, including the proper parsing of embedded comments and CData sections in E4X literals. Using comments and/or CData sections without full E4X enabled results in syntax errors.

SUMMARY

ECMAScript for XML (E4X) is an extension to ECMAScript defined in the ECMA-357 specification. The purpose of E4X is to provide syntax for working with XML data that is more like that of standard ECMAScript. E4X has the following characteristics:

- Unlike the DOM, there is only one type to represent all of the different node types present in XML.
- The `XML` object encapsulates data and the behavior necessary for all nodes. To represent a collection of multiple nodes, the specification defines an `XMLElement` object.

- Two other types, `Namespace` and `QName`, are present to represent namespaces and qualified names, respectively.

E4X changes standard ECMAScript syntax as follows to allow for easier querying of an XML structure:

- Using two dots (..) indicates that all descendants should be matched, whereas using the @ character indicates that one or more attributes should be returned.
- The asterisk character (*) represents a wildcard that can match any node of the given type.
- All of these queries can also be accomplished via a series of methods that perform the same operation.

By the end of 2011, Firefox was the only browser to support E4X. Though no other browser vendors have committed to implementing E4X, it has gained a certain amount of popularity on the server with the BEA Workshop for WebLogic and YQL from Yahoo!.

20

JSON

WHAT'S IN THIS CHAPTER?

- ▶ Understanding JSON syntax
- ▶ JSON parsing
- ▶ JSON serialization

There was a time when XML was the de facto standard for transmitting structured data over the Internet. The first iteration of web services was largely XML-based, highlighting its target of server-to-server communication. XML was not, however, without its detractors. Some believed that the language was overly verbose and redundant. Several solutions arose to counter these problems, but the Web had already started moving in a new direction.

Douglas Crockford first specified JavaScript Object Notation (JSON) as IETF RFC 4627 in 2006 even though it was in use as early as 2001. JSON is a strict subset of JavaScript, making use of several patterns found in JavaScript to represent structured data. Crockford put forth JSON as a better alternative to XML for accessing structured data in JavaScript, since it could be passed directly to `eval()` and didn't require the creation of a DOM.

The most important thing to understand about JSON is that it is a data format, not a programming language. JSON is not a part of JavaScript even though they share syntax. JSON is also not solely used by JavaScript, since it is a data format. There are parsers and serializers available in many programming languages.

SYNTAX

JSON syntax allows the representation of three types of values:

- ▶ **Simple Values** — Strings, numbers, Booleans, and `null` can all be represented in JSON using the same syntax as JavaScript. The special value `undefined` is not supported.

- **Objects** — The first complex data type, objects represent ordered key-value pairs. Each value may be a primitive type or a complex type.
- **Arrays** — The second complex data type, arrays represent an ordered list of values that are accessible via a numeric index. The values may be of any type, including simple values, objects, and even other arrays.

There are no variables, functions, or object instances in JSON. JSON is all about representing structured data, and although it shares syntax with JavaScript, it should not be confused with JavaScript paradigms.

Simple Values

In its simplest form, JSON represents a small number of simple values. For example, the following is valid JSON:

```
5
```

This is JSON that represents the number 5. Likewise, the following is also valid JSON representing a string:

```
"Hello world!"
```

The big difference between JavaScript strings and JSON strings is that JSON strings must use double quotes to be valid (single quotes causes a syntax error).

Boolean values and `null` are valid exactly as they are as stand-alone JSON. In practice, however, JSON is most often used to represent more complex data structures of which simple values represent just part of the overall information.

Objects

Objects are represented using a slight modification of object literal notation. Object literals in JavaScript look like this:

```
var person = {  
    name: "Nicholas",  
    age: 29  
};
```

While this is the standard way that developers create object literals, it's the quoted property format that is used in JSON. The following is exactly the same as the previous example:

```
var object = {  
    "name": "Nicholas",  
    "age": 29  
};
```

The JSON representation of this same object is then:

```
{
  "name": "Nicholas",
  "age": 29
}
```

There are a couple of differences from the JavaScript example. First, there is no variable declaration (variables don't exist in JSON). Second, there is no trailing semicolon (not needed since this isn't a JavaScript statement). Once again, the quotes around the property name are required to be valid JSON. The value can be any simple or complex value, allowing you to embed objects within objects, such as:

```
{
  "name": "Nicholas",
  "age": 29,
  "school": {
    "name": "Merrimack College",
    "location": "North Andover, MA"
  }
}
```

This example embeds school information into the top-level object. Even though there are two properties called "name", they are in two different objects and so are allowed. You do want to avoid having two properties of the same name in the same object.

Unlike JavaScript, object property names in JSON must always be double-quoted. It's a common mistake to hand-code JSON without these double quotes or using single quotes.

Arrays

The second complex type in JSON is the array. Arrays are represented in JSON using array literal notation from JavaScript. For example, this is an array in JavaScript:

```
var values = [25, "hi", true];
```

You can represent this same array in JSON using a similar syntax:

```
[25, "hi", true]
```

Note once again the absence of a variable or a semicolon. Arrays and objects can be used together to represent more complex collections of data, such as:

```
[
  {
    "title": "Professional JavaScript",
    "authors": [
      "Nicholas C. Zakas"
    ],
    edition: 3,
    year: 2011
  },
  {
    "title": "JavaScript: The Good Parts",
    "authors": [
      "John Resig"
    ],
    edition: 1,
    year: 2008
  }
]
```

```
{
    "title": "Professional JavaScript",
    "authors": [
        "Nicholas C. Zakas"
    ],
    edition: 2,
    year: 2009
},
{
    "title": "Professional Ajax",
    "authors": [
        "Nicholas C. Zakas",
        "Jeremy McPeak",
        "Joe Fawcett"
    ],
    edition: 2,
    year: 2008
},
{
    "title": "Professional Ajax",
    "authors": [
        "Nicholas C. Zakas",
        "Jeremy McPeak",
        "Joe Fawcett"
    ],
    edition: 1,
    year: 2007
},
{
    "title": "Professional JavaScript",
    "authors": [
        "Nicholas C. Zakas"
    ],
    edition: 1,
    year: 2006
}
]
```

This array contains a number of objects representing books. Each object has several keys, one of which is "authors", which is another array. Objects and arrays are typically top-level parts of a JSON data structure (even though this is not required) and can be used to create a large number of data structures.

PARSING AND SERIALIZATION

JSON's rise to popularity was not necessarily because it used familiar syntax. More so, it was because the data could be parsed into a usable object in JavaScript. This stood in stark contrast to XML that was parsed into a DOM document, making extraction of data into a bit of a chore for JavaScript developers. For example, the JSON code in the previous section contains a list of books, and you can easily get the title of the third book via:

```
books[2].title
```

This assumes that the data structure was stored in a variable named `books`. Compare this to a typical walk through a DOM structure:

```
doc.getElementsByTagName("book")[2].getAttribute("title")
```

With all of the extra method calls, it's no wonder that JSON became incredibly popular with JavaScript developers. After that, JSON went on to become the de facto standard for web services.

The JSON Object

Early JSON parsers did little more than use JavaScript's `eval()` function. Since JSON is a subset of JavaScript's syntax, `eval()` could parse, interpret, and return the data as JavaScript objects and arrays. ECMAScript 5 formalized JSON parsing under a native global called `JSON`. This object is supported in Internet Explorer 8+, Firefox 3.5+, Safari 4+, Chrome, and Opera 10.5+. A shim for older browsers can be found at <https://github.com/douglascrockford/JSON-js>. It's important not to use `eval()` alone for evaluating JSON in older browsers because of the risk of executable code. The `JSON` shim is the best option for browsers without native JSON parsing.

The `JSON` object has two methods: `stringify()` and `parse()`. In simple usage, these methods serialize JavaScript objects into a JSON string and parse JSON into a native JavaScript value, respectively. For example:



Available for
download on
Wrox.com

```
var book = {
    title: "Professional JavaScript",
    authors: [
        "Nicholas C. Zakas"
    ],
    edition: 3,
    year: 2011
};

var jsonText = JSON.stringify(book);
```

[JSONStringifyExample01.htm](#)

This example serializes a JavaScript object into a JSON string using `JSON.stringify()` and stores it in `jsonText`. By default, `JSON.stringify()` outputs a JSON string without any extra white space or indentation, so the value stored in `jsonText` is:

```
{"title": "Professional JavaScript", "authors": ["Nicholas C. Zakas"], "edition": 3,  
"year": 2011}
```

When serializing a JavaScript object, all functions and prototype members are intentionally omitted from the result. Additionally, any property whose value is `undefined` is also skipped. You're left with just a representation of the instance properties that are one of the JSON data types.

A JSON string can be passed directly into `JSON.parse()` and it creates an appropriate JavaScript value. For example, you can create an object similar to the `book` object using this code:

```
var bookCopy = JSON.parse(jsonText);
```

Note that `book` and `bookCopy` are each separate objects without any relationship to one another even though they do share the same properties.

An error is thrown if the text passed into `JSON.parse()` is not valid JSON.

Serialization Options

The `JSON.stringify()` method actually accepts two arguments in addition to the object to serialize. These arguments allow you to specify alternate ways to serialize a JavaScript object. The first argument is a filter, which can be either an array or a function, and the second argument is an option for indenting the resulting JSON string. When used separately or together, this provides some very useful functionality for controlling JSON serialization.

Filtering Results

If the argument is an array, then `JSON.stringify()` will include only object properties that are listed in the array. Consider the following:



```
var book = {
    "title": "Professional JavaScript",
    "authors": [
        "Nicholas C. Zakas"
    ],
    edition: 3,
    year: 2011
};

var jsonText = JSON.stringify(book, ["title", "edition"]);
```

[JSONStringifyExample01.htm](#)

The second argument to `JSON.stringify()` is an array with two strings: `"title"` and `"edition"`. These correspond to properties in the object being serialized, and so only those properties appear in the resulting JSON string:

```
{"title": "Professional JavaScript", "edition": 3}
```

When the second argument is a function, the behavior is slightly different. The provided function receives two arguments: the property key name and the property value. You can look at the key to determine what to do with the property. The key is always a string but might be an empty string if a value isn't part of a key-value pair.

In order to change the serialization of the object, return the value that should be included for that key. Keep in mind that returning `undefined` will result in the property being omitted from the result. Here's an example:



```

var book = {
    "title": "Professional JavaScript",
    "authors": [
        "Nicholas C. Zakas"
    ],
    edition: 3,
    year: 2011
};

var jsonText = JSON.stringify(book, function(key, value){
    switch(key){
        case "authors":
            return value.join(", ")

        case "year":
            return 5000;

        case "edition":
            return undefined;

        default:
            return value;
    }
});

```

[JSONStringifyExample02.htm](#)

The function filters based on the key. The "authors" key is translated from an array to a string, the "year" key is set to 5000, and the "edition" key is removed altogether by returning `undefined`. It's important to provide a default behavior that returns the passed-in value so that all other values are passed through to the result. The first call to this function actually has `key` equal to an empty string and the `value` set to the `book` object. The resulting JSON string is:

```
{"title": "Professional JavaScript", "authors": "Nicholas C. Zakas", "year": 5000}
```

Keep in mind that filters apply to all objects contained in the object to be serialized, so an array of multiple objects with these properties will result in every object including only the "title" and "edition" properties.

Firefox 3.5–3.6 had a bug in its implementation of `JSON.stringify()` when a function was used as the second argument. The function can act only as a filter: returning `undefined` means the property is skipped, while returning anything else causes the property to be included. This behavior was fixed in Firefox 4.

String Indentation

The third argument of `JSON.stringify()` controls indentation and white space. When this argument is a number, it represents the number of spaces to indent each level. For example, to indent each level by four spaces, use the following:



```
var book = {
    "title": "Professional JavaScript",
    "authors": [
        "Nicholas C. Zakas"
    ],
    "edition": 3,
    "year": 2011
};

var jsonText = JSON.stringify(book, null, 4);
```

JSONStringifyExample03.htm

The string stored in jsonText is:

```
{
    "title": "Professional JavaScript",
    "authors": [
        "Nicholas C. Zakas"
    ],
    "edition": 3,
    "year": 2011
}
```

You may have noticed that `JSON.stringify()` also inserts new lines into the JSON string for easier reading. This happens for all valid indentation argument values. (Indentation without new lines isn't very useful.) The maximum numeric indentation value is 10; passing in a value larger than 10 automatically sets the value to 10.

If the indentation argument is a string instead of a number, then the string is used as the indentation character for the JSON string instead of a space. Using a string, you can set the indentation character to be a tab or something completely arbitrary like two dashes:

```
var jsonText = JSON.stringify(book, null, " - -");
```

The jsonText value then becomes:

```
{
--"title": "Professional JavaScript",
--"authors": [
----"Nicholas C. Zakas"
--],
--"edition": 3,
--"year": 2011
}
```

There is a ten-character limit on the indentation string to use. If a string longer than ten characters is used, then it is truncated to the first ten characters.

The `toJSON()` Method

Sometimes objects need custom JSON serialization above and beyond what `JSON.stringify()` can do. In those cases, you can add a `toJSON()` method to the object and have it return the

proper JSON representation for itself. In fact, the native `Date` object has a `toJSON()` method that automatically converts JavaScript `Date` objects into an ISO 8601 date string (essentially, the same as calling `toISOString()` on the `Date` object).

A `toJSON()` method can be added to any object, for example:



```
var book = {
    "title": "Professional JavaScript",
    "authors": [
        "Nicholas C. Zakas"
    ],
    edition: 3,
    year: 2011,
    toJSON: function(){
        return this.title;
    }
};

var jsonText = JSON.stringify(book);
```

[JSONStringifyExample05.htm](#)

This code defines a `toJSON()` method on the `book` object that simply returns the title of the book. Similar to the `Date` object, this object is serialized to a simple string instead of an object. You can return any serialization value from `toJSON()`, and it will work appropriately. Returning `undefined` causes the value to become null if the object is embedded in another object or else is just `undefined` if the object is top-level.

The `toJSON()` method can be used in addition to the filter function, and so it's important to understand the order in which the various parts of a serialization process take place. When an object is passed into `JSON.stringify()`, the following steps are taken:

1. Call the `toJSON()` method if it's available to retrieve the actual value. Use the default serialization otherwise.
2. If the second argument is provided, apply the filter. The value that is passed into a filter function will be the value returned from step 1.
3. Each value from step 2 is serialized appropriately.
4. If the third argument is provided, format appropriately.

It's important to understand this order when deciding whether to create a `toJSON()` method or to use a filter function or to do both.

Parsing Options

The `JSON.parse()` method also accepts an additional argument, which is a function that is called on each key-value pair. The function is called a *reviver* function to distinguish it from the *replacer* (filter) function that `JSON.stringify()` accepts, even though the format is exactly the same: the function receives two arguments, the key and the value, and needs to return a value.



If the reviver function returns `undefined`, then the key is removed from the result; if it returns any other value, that value is inserted into the result. A very common use of the reviver function is to turn date strings into `Date` objects. For example:

```
var book = {
    "title": "Professional JavaScript",
    "authors": [
        "Nicholas C. Zakas"
    ],
    edition: 3,
    year: 2011,
    releaseDate: new Date(2011, 11, 1)
};

var jsonText = JSON.stringify(book);

var bookCopy = JSON.parse(jsonText, function(key, value){
    if (key == "releaseDate"){
        return new Date(value);
    } else {
        return value;
    }
});

alert(bookCopy.releaseDate.getFullYear());
```

[JSONParseExample02.htm](#)

This code starts with the addition of a `releaseDate` property to the `book` object, which is a `Date`. The object is serialized to get a valid JSON string and then parsed back into an object, `bookCopy`. The reviver function looks for the `"releaseDate"` key and, when found, creates a new `Date` object based on that string. The resulting `bookCopy.releaseDate` property is then a `Date` object so the `getFullYear()` method can be called.

SUMMARY

JSON is a lightweight data format designed to easily represent complex data structures. The format uses a subset of JavaScript syntax to represent objects, arrays, strings, numbers, Booleans, and `null`. Even though XML can handle the same job, JSON is less verbose and has better support in JavaScript.

ECMAScript 5 defines a native `JSON` object that is used for serialization of objects into JSON format and for parsing JSON data into JavaScript objects. The `JSON.stringify()` and `JSON.parse()` methods are used for these two operations, respectively. Both methods have a number of options that allow you to change the default behavior to filter or otherwise modify the process.

The native `JSON` object is well supported across browsers, including Internet Explorer 8+, Firefox 3.5+, Safari 4+, Opera 10.5, and Chrome.

21

Ajax and Comet

WHAT'S IN THIS CHAPTER?

- ▶ Using the XMLHttpRequest object
- ▶ Working with XMLHttpRequest events
- ▶ Cross-domain Ajax restrictions

In 2005, Jesse James Garrett penned an online article titled “Ajax: A New Approach to Web Applications” (www.adaptivepath.com/ideas/essays/archives/000385.php). This article outlined a technique that he referred to as *Ajax*, short for *Asynchronous JavaScript+XML*. The technique consisted of making server requests for additional data without unloading the web page, resulting in a better user experience. Garrett explained how this technique could be used to change the traditional click-and-wait paradigm that the Web had been stuck in since its inception.

The key technology pushing Ajax forward was the XMLHttpRequest (XHR) object, first invented by Microsoft and then duplicated by other browser vendors. Prior to the introduction of XHR, Ajax-style communication had to be accomplished through a number of hacks, mostly using hidden frames or iframes. XHR introduced a streamlined interface for making server requests and evaluating the responses. This allowed for asynchronous retrieval of additional information from the server, meaning that a user click didn’t have to refresh the page to retrieve more data. Instead, an XHR object could be used to retrieve the data and then the data could be inserted into the page using the DOM. And despite the mention of XML in the name, Ajax communication is format-agnostic; the technique is about retrieving data from the server without refreshing a page, not necessarily about XML.

The technique that Garrett referred to as Ajax had, in fact, been around for some time. Typically called *remote scripting* prior to Garrett’s article, such browser-server communication has been possible since 1998 using different techniques. Early on, server requests could be made from JavaScript through an intermediary, such as a Java applet or Flash movie. The XHR

object brought native browser communication capabilities to developers, reducing the amount of work necessary to achieve the result.

Renamed as Ajax, the popularity of browser-server communication exploded in late 2005 and early 2006. A renewed interest in JavaScript and the Web in general brought new techniques and patterns for using these capabilities. Therefore, the XHR object is now a necessary tool in every web developer's tool kit.

THE XMLHttpRequest OBJECT

Internet Explorer 5 was the first browser to introduce the XHR object. It did so through the use of an ActiveX object included as part of the MSXML library. As such, three versions of the XHR object may be used in the browser: `MSXML2.XMLHttp`, `MSXML2.XMLHttp.3.0`, and `MSXML2.XMLHttp.6.0`. Using an XHR object with the MSXML library requires a function similar to the one used for creating XML documents in Chapter 18, as shown in the following example:



```
//function for IE versions prior to 7
function createXHR(){
    if (typeof arguments.callee.activeXString != "string") {
        var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
                       "MSXML2.XMLHttp"],
            i, len;

        for (i=0,len=versions.length; i < len; i++) {
            try {
                new ActiveXObject(versions[i]);
                arguments.callee.activeXString = versions[i];
                break;
            } catch (ex) {
                //skip
            }
        }
    }

    return new ActiveXObject(arguments.callee.activeXString);
}
```

This function tries to create the most recent version of the XHR object that is available on Internet Explorer.

Internet Explorer 7+, Firefox, Opera, Chrome, and Safari all support a native XHR object that can be created using the `XMLHttpRequest` constructor as follows:

```
var xhr = new XMLHttpRequest();
```

If you need only support Internet Explorer versions 7 and later, then you can forego the previous function in favor of using the native XHR implementation. If, on the other hand, you must extend support to earlier versions of Internet Explorer, the `createXHR()` function can be augmented to check for the native XHR object, as shown here:



```

function createXHR(){
    if (typeof XMLHttpRequest != "undefined"){
        return new XMLHttpRequest();
    } else if (typeof ActiveXObject != "undefined"){
        if (typeof arguments.callee.activeXString != "string"){
            var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
                            "MSXML2.XMLHttp"],
                i, len;

            for (i=0,len=versions.length; i < len; i++){
                try {
                    new ActiveXObject(versions[i]);
                    arguments.callee.activeXString = versions[i];
                    break;
                } catch (ex){
                    //skip
                }
            }
        }
        return new ActiveXObject(arguments.callee.activeXString);
    } else {
        throw new Error("No XHR object available.");
    }
}

```

XHRExample01.htm

The new code in this function first checks for the native XHR object and, if found, returns a new instance. If the native object isn't found, then it checks for ActiveX support. An error is thrown if neither option is available. You can then create an XHR object using the following code in all browsers:

```
var xhr = createXHR();
```

Since the XHR implementation in each browser is compatible with the original Internet Explorer version, you can use the created `xhr` object the same way in all browsers.

XHR Usage

To begin using an XHR object, you will first call the method `open()`, which accepts three arguments: the type of request to be sent ("get", "post", and so on), the URL for the request, and a Boolean value indicating if the request should be sent asynchronously. Here's an example:

```
xhr.open("get", "example.php", false);
```

This line opens a synchronous GET request for `example.php`. There are a couple of things to note about this code. First, the URL is relative to the page on which the code is called, although an absolute path can be given as well. Second, the call to `open()` does not actually send the request; it simply prepares a request to be sent.



You can access only URLs that exist on the same origin, which means the same domain, using the same port, and with the same protocol. If the URL specifies any of these differently than the page making the request, a security error is thrown.

To send the specified request, you must call the `send()` method as follows:



Available for download on Wrox.com

```
xhr.open("get", "example.txt", false);
xhr.send(null);
```

XHRExample01.htm

The `send()` method accepts a single argument, which is data to be sent as the body of the request. If no body data needs to be sent, you must pass in `null`, because this argument is required for some browsers. Once `send()` is called, the request is dispatched to the server.

Since this request is synchronous, the JavaScript code will wait for the response to return before continuing execution. When a response is received, the XHR object properties are filled with data. The relevant properties are as follows:

- `responseText` — The text that was returned as the body of the response.
- `responseXML` — Contains an XML DOM document with the response data if the response has a content type of `"text/xml"` or `"application/xml"`.
- `status` — The HTTP status of the response.
- `statusText` — The description of the HTTP status.

When a response is received, the first step is to check the `status` property to ensure that the response was returned successfully. Generally, HTTP status codes in the 200s are considered successful and some content will be available in `responseText` and possibly in `responseXML` if the content type is correct. In addition, the status code of 304 indicates that a resource hasn't been modified and is being served from the browser's cache, which also means a response is available. To ensure that a proper response was received, you should check for all of these statuses, as shown here:

```
xhr.open("get", "example.txt", false);
xhr.send(null);

if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
    alert(xhr.responseText);
} else {
    alert("Request was unsuccessful: " + xhr.status);
}
```

XHRExample01.htm

This code displays either the content returned from the server or an error message, depending on the status code that was returned. It's recommended to always check the `status` property to determine

the best course of action and to avoid using `statusText` for this purpose, because the latter has proven to be unreliable across browsers. The `responseText` property is always filled with the body of the response, regardless of the content type, whereas `responseXML` will be `null` for non-XML data.



Several browsers incorrectly report a 204 status code. ActiveX versions of XHR in Internet Explorer set status to 1223 when a 204 is retrieved, and native XHR objects in Internet Explorer normalize 204 to 200. Opera reports a status of 0 when a 204 is retrieved.

Although it's possible to make synchronous requests such as this one, most of the time it's better to make asynchronous requests that allow JavaScript code execution to continue without waiting for the response. The XHR object has a `readyState` property that indicates what phase of the request/response cycle is currently active. The possible values are as follows:

- 0 — Uninitialized. The `open()` method hasn't been called yet.
- 1 — Open. The `open()` method has been called but `send()` has not been called.
- 2 — Sent. The `send()` method has been called but no response has been received.
- 3 — Receiving. Some response data has been retrieved.
- 4 — Complete. All of the response data has been retrieved and is available.

Whenever the `readyState` changes from one value to another, the `readystatechange` event is fired. You can use this opportunity to check the value of `readyState`. Generally speaking, the only `readyState` of interest is 4, which indicates that all of the data is ready. The `onreadystatechange` event handler should be assigned prior to calling `open()` for cross-browser compatibility. Consider the following example:



Available for download on
Wrox.com

```
var xhr = createXHR();
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
};
xhr.open("get", "example.txt", true);
xhr.send(null);
```

[XHRAsyncExample01.htm](#)

Note that this code uses the DOM Level 0 style of attaching an event handler to the `XHR` object, because not all browsers support the DOM Level 2 style of event attachment. Unlike other event

handlers, no event object is passed into the `onreadystatechange` event handler. Instead you must use the `XHR` object itself to determine what to do next.



This example uses the `xhr` object inside the `onreadystatechange` event handler instead of the `this` object because of scoping issues with the `onreadystatechange` event handler. Using this may cause the function to fail or cause an error, depending on the browser being used, so it's safer to use the actual XHR object-instance variable.

You can cancel an asynchronous request before a response is received by calling the `abort()` method like this:

```
xhr.abort();
```

Calling this method makes the `XHR` object stop firing events and prevents access to any of the response-related properties on the object. Once a request has been aborted, the `XHR` object should be dereferenced. Because of memory issues, it's not recommended to reuse an `XHR` object.

HTTP Headers

Every HTTP request and response sends along with it a group of header information that may or may not be of interest to the developer. The `XHR` object exposes both types of headers — those on the request and those on the response — through several methods.

By default, the following headers are sent when an `XHR` request is sent:

- `Accept` — The content types that the browser can handle.
- `Accept-Charset` — The character sets that the browser can display.
- `Accept-Encoding` — The compression encodings handled by the browser.
- `Accept-Language` — The languages the browser is running in.
- `Connection` — The type of connection the browser is making with the server.
- `Cookie` — Any cookies set on the page.
- `Host` — The domain of the page making the request.
- `Referer` — The URI of the page making the request. Note that this header is spelled incorrectly in the HTTP specification and so must be spelled incorrectly for compatibility purposes. (The correct spelling of this word is “referrer”.)
- `User-Agent` — The browser’s user-agent string.

Although the exact request headers sent vary from browser to browser, these are the ones that are generally sent. You can set additional request headers by using the `setRequestHeader()` method. This method accepts two arguments: the name of the header and the value of the header. For request

headers to be sent, `setRequestHeader()` must be called after `open()` but before `send()`, as in the following example:



```
var xhr = createXHR();
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
};

xhr.open("get", "example.php", true);
xhr.setRequestHeader("MyHeader", "MyValue");
xhr.send(null);
```

XHRRRequestHeadersExample01.htm

The server can read these custom request headers to determine an appropriate course of action. It's advisable to always use custom header names rather than those the browser normally sends, because using the default ones may affect the server response. Some browsers will allow overwriting default headers, but others will not.

You can retrieve the response headers from an XHR object by using the `getResponseHeader()` method and passing in the name of the header to retrieve. It's also possible to retrieve all headers as a long string by using the `getAllResponseHeaders()` method. Here's an example of both methods:

```
var myHeader = xhr.getResponseHeader("MyHeader");
var allHeaders = xhr.getAllResponseHeaders();
```

Headers can be used to pass additional, structured data from the server to the browser. The `getAllResponseHeaders()` method typically returns something along the lines of the following:

```
Date: Sun, 14 Nov 2004 18:04:03 GMT
Server: Apache/1.3.29 (Unix)
Vary: Accept
X-Powered-By: PHP/4.3.8
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

This output allows you to parse the response headers to find all of the header names that were sent rather than check for the existence of each one individually.

GET Requests

The most common type of request to execute is a GET, which is typically made when the server is being queried for some sort of information. If necessary, query-string arguments can be appended to the end of the URL to pass information to the server. For XHR, this query string must be present and encoded correctly on the URL that is passed into the `open()` method.

One of the most frequent errors made with GET requests is to have an improperly formatted query string. Each query-string name and value must be encoded using `encodeURIComponent()` before being attached to the URL, and all of the name-value pairs must be separated by an ampersand, as in this example:

```
xhr.open("get", "example.php?name1=value1&name2=value2", true);
```

The following function helps to add query-string arguments to the end of an existing URL:

```
function addURLParam(url, name, value) {
    url += (url.indexOf "?" == -1 ? "?" : "&") +
    url += encodeURIComponent(name) + "=" + encodeURIComponent(value);
    return url;
}
```

The `addURLParam()` function takes three arguments: the URL to add the parameters to, the parameter name, and the parameter value. First, the function checks to see if the URL already contains a question mark (to determine if other parameters already exist). If it doesn't, then the function appends a question mark; otherwise it adds an ampersand. Next the name and value are encoded and appended to the end of the URL. The last step is to return the updated URL.

This function can be used to build up a URL for a request, as shown in the following example:

```
var url = "example.php";

//add the arguments
url = addURLParam(url, "name", "Nicholas");
url = addURLParam(url, "book", "Professional JavaScript");

//initiate request
xhr.open("get", url, false);
```

Using the `addURLParam()` function here ensures that the query string is properly formed for use with the XHR object.

POST Requests

The second most frequent type of request is POST, which is typically used to send data to the server that should save data. Each POST request is expected to have data submitted as the body of the request, whereas GET requests traditionally do not. The body of a POST request can contain a very large amount of data, and that data can be in any format. You can initiate a POST request by specifying `post` as the first argument to the `open()` method. For example:

```
xhr.open("post", "example.php", true);
```

The second part is to pass some data to the `send()` method. Since XHR was originally designed to work primarily with XML, you can pass in an XML DOM document that will be serialized and submitted as the request body. You can also pass in any string to send to the server.

By default, a POST request does not appear the same to the server as a web-form submission. Server logic will need to read the raw post data to retrieve your data. You can, however, mimic a form

submission using XHR. The first step is to set the Content-Type header to application/x-www-form-urlencoded, which is the content type set when a form is submitted. The second step is to create a string in the appropriate format. As discussed in Chapter 14, post data is sent in the same format as a query string. If a form already on the page should be serialized and sent to the server via XHR, you can use the serialize() function from Chapter 14 to create the string, as shown here:



```
function submitData(){
    var xhr = createXHR();
    xhr.onreadystatechange = function(){
        if (xhr.readyState == 4){
            if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
                alert(xhr.responseText);
            } else {
                alert("Request was unsuccessful: " + xhr.status);
            }
        }
    };

    xhr.open("post", "postexample.php", true);
    xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    var form = document.getElementById("user-info");
    xhr.send(serialized(form));
}
```

[XHRPostExample01.htm](#)

In this function, form data from a form with the ID "user-info" is serialized and sent to the server. The PHP file postexample.php can then retrieve the posted data via \$_POST. Consider this example:

```
<?php
    header("Content-Type: text/plain");
    echo <<<EOF
Name: ${_POST['user-name']}
Email: ${_POST['user-email']}
EOF;
?>
```

[postexample.php](#)

Without including the Content-Type header, the data will not appear in the \$_POST superglobal — you'd need to use \$HTTP_RAW_POST_DATA to access it.



POST requests have more overhead associated with them than do GET requests. In terms of performance, GET requests can be up to two times faster than POST requests sending the same amount of data.

XMLHttpRequest LEVEL 2

The popularity of the XMLHttpRequest object as a de facto standard led to the creation of official specifications from the W3C to govern its behavior. XMLHttpRequest Level 1 simply defined the already existing implementation details of the XMLHttpRequest object. XMLHttpRequest Level 2 went on to evolve the XMLHttpRequest object further. Not all browsers have implemented all parts of the Level 2 specification, but all browsers have implemented some of the functionality.

The FormData Type

The serialization of form data is frequently needed in modern web applications, and so the XMLHttpRequest Level 2 specification introduces the `FormData` type. The `FormData` type makes it easy to both serialize existing forms and create data in the same format as a form for easy transmission via XHR. The following creates a `FormData` object and populates it with some data:

```
var data = new FormData();
data.append("name", "Nicholas");
```

The `append()` method accepts two arguments, a key and a value, essentially the name of a form field and the value that the field contains. You can add as many of these pairs as you would like. It's also possible to prepopulate the key-value pairs with data that exists in a form element by passing in the form element to the `FormData` constructor:

```
var data = new FormData(document.forms[0]);
```

Once you have an instance of `FormData`, it can be passed directly into the XHR `send()` method, as in this example:



```
var xhr = createXHR();
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
};

xhr.open("post", "postexample.php", true);
var form = document.getElementById("user-info");
xhr.send(new FormData(form));
```

[XHRFormDataExample01.htm](#)

One of the conveniences of the `FormData` type is that you don't need to explicitly set any request headers on the XMLHttpRequest object. The XMLHttpRequest object recognizes the passed-in data type as an instance of `FormData` and configures the headers appropriately.

The `FormData` type is supported in Firefox 4+, Safari 5+, Chrome, and WebKit for Android 3+.

Timeouts

In Internet Explorer 8, the `XHR` object was augmented to include a `timeout` property that indicates the number of milliseconds the request should wait for a response before aborting. When the `timeout` property is set to a number and the response is not received within that number of milliseconds, a `timeout` event is fired and the `ontimeout` event handler is called. This functionality was later added into the XMLHttpRequest Level 2 specification. Here's an example:



Available for download on Wrox.com

```

var xhr = createXHR();
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        try {
            if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
                alert(xhr.responseText);
            } else {
                alert("Request was unsuccessful: " + xhr.status);
            }
        } catch (ex){
            //assume handled by ontimeout
        }
    }
};

xhr.open("get", "timeout.php", true);
xhr.timeout = 1000; //set timeout for 1 second (IE 8+ only)
xhr.ontimeout = function(){
    alert("Request did not return in a second.");
};
xhr.send(null);

```

[XHRTIMEOUTExample01.htm](#)

This example illustrates the use of the `timeout` property. Setting it equal to 1000 milliseconds means that if the request doesn't return in 1 second or less, the request is aborted. When that happens, the `ontimeout` event handler is called. The `readyState` is still changed to 4, which means the `onreadystatechange` event handler is called. However, an error occurs if you try to access the `status` property after a timeout has occurred. To protect against this, encapsulate the code that checks the `status` property in a `try-catch` statement.

Internet Explorer 8+ is still, as of the time of this writing, the only browser to support timeouts.

The `overrideMimeType()` Method

Firefox first introduced `overrideMimeType()` as a way to override the MIME type of an XHR response. This was later added to XMLHttpRequest Level 2. Since the returned MIME type for a

response determines how the response is handled by the XHR object, having a way to override the type returned by the server is a useful addition.

Consider the case where the server sends a MIME type of `text/plain` that actually contains XML. This would result in the `responseXML` property being `null` even though the data is actually XML. By calling `overrideMimeType()`, you can ensure the response is treated as XML instead of plain text:

```
var xhr = createXHR();
xhr.open("get", "text.php", true);
xhr.overrideMimeType("text/xml");
xhr.send(null);
```

This example forces the XHR object to treat the response as XML instead of plain text. The call to `overrideMimeType()` must happen before the call to `send()` in order to correctly override the response MIME type.

The `overrideMimeType()` method is supported in Firefox, Safari 4+, Opera 10.5+, and Chrome.

PROGRESS EVENTS

The Progress Events specification is a W3C Working Draft defining events for client-server communication. These events were first targeted at XHR explicitly but have now also made their way into other similar APIs. There are six progress events:

- `loadstart` — Fires when the first byte of the response has been received.
- `progress` — Fires repeatedly as a response is being received.
- `error` — Fires when there was an error attempting the request.
- `abort` — Fires when the connection was terminated by calling `abort()`.
- `load` — Fires when the response has been fully received.
- `loadend` — Fires when the communication is complete and after firing `error`, `abort`, or `load`.

Each request begins with the `loadstart` event being fired; followed by one or more `progress` events; then one of `error`, `abort`, or `load`; finally ending with `loadend`.

The first five events are supported in Firefox 3.5+, Safari 4+, Chrome, Safari for iOS, and WebKit for Android. Opera, as of version 11, and Internet Explorer 8+ support only the `load`. No browsers currently support the `loadend` event.

Most of these events are straightforward, but there are two that have some subtleties to be aware of.

The `load` Event

When Firefox first implemented a version of the XHR object, it sought to simplify the interaction model. To that end, the `load` event was introduced as a replacement for the `readystatechange`

event. The `load` event fires as soon as the response has been completely received, eliminating the need to check the `readyState` property. The `onload` event handler receives an event object whose `target` property is set to the `XHR` object instance, and all of the `XHR` object properties and methods are available from within. However, not all browsers properly implement the `event` object for this event, necessitating the use of the `XHR` object variable itself, as shown in the following example:



Available for download on
Wrox.com

```
var xhr = createXHR();
xhr.onload = function(){
    if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
        alert(xhr.responseText);
    } else {
        alert("Request was unsuccessful: " + xhr.status);
    }
};
xhr.open("get", "altevents.php", true);
xhr.send(null);
```

XHRProgressEventExample01.htm

As long as a response is received from the server, regardless of the status, the `load` event will fire. This means you must check the `status` property to determine if the appropriate data is available. The `load` event is supported by Firefox, Opera, Chrome, and Safari.

The progress Event

Another XHR innovation from Mozilla is the `progress` event, which fires periodically as the browser receives new data. The `onprogress` event listener receives an event object whose `target` is the `XHR` object and contains three additional properties: `lengthComputable`, a Boolean indicating if progress information is available; `position`, which is the number of bytes that have already been received; and `totalSize`, which is the total number of expected bytes as defined by the `Content-Length` response header. With that information, you can provide a progress indicator to the user. The following code includes an example of how this is done:

```
var xhr = createXHR();
xhr.onload = function(event){
    if ((xhr.status >= 200 && xhr.status < 300) ||
        xhr.status == 304){
        alert(xhr.responseText);
    } else {
        alert("Request was unsuccessful: " + xhr.status);
    }
};
xhr.onprogress = function(event){
    var divStatus = document.getElementById("status");
    if (event.lengthComputable){
        divStatus.innerHTML = "Received " + event.position + " of " +
        event.totalSize +
        " bytes";
    }
}
```

```
};

xhr.open("get", "altevents.php", true);
xhr.send(null);
```

[XHRProgressEventExample01.htm](#)

For proper execution, the `onprogress` event handler must be attached prior to calling `open()`. In the preceding example, an HTML element is filled with status information every time the `progress` event is fired. Assuming that the response has a `Content-Length` header, you can also use this information to calculate the percentage of the response that has already been received.

CROSS-ORIGIN RESOURCE SHARING

One of the major limitations of Ajax communication via XHR is the cross-origin security policy. By default, XHR objects can access resources only on the domain from which the containing web page originates. This security feature prevents some malicious behavior. However, the need for legitimate cross-origin access was great enough for solutions to begin appearing in browsers.

Cross-Origin Resource Sharing (CORS) is a W3C Working Draft that defines how the browser and server must communicate when accessing sources across origins. The basic idea behind CORS is to use custom HTTP headers to allow both the browser and the server to know enough about each other to determine if the request or response should succeed or fail.

For a simple request, one that uses either GET or POST with no custom headers and whose body is `text/plain`, the request is sent with an extra header called `Origin`. The `Origin` header contains the origin (protocol, domain name, and port) of the requesting page so that the server can easily determine whether or not it should serve a response. An example `Origin` header might look like this:

```
Origin: http://www.nczonline.net
```

If the server decides that the request should be allowed, it sends an `Access-Control-Allow-Origin` header echoing back the same origin that was sent or "*" if it's a public resource. For example:

```
Access-Control-Allow-Origin: http://www.nczonline.net
```

If this header is missing, or the origins don't match, then the browser disallows the request. If all is well, then the browser processes the request. Note that neither the requests nor the responses include cookie information.

CORS in Internet Explorer

Microsoft introduced the `XDomainRequest` (XDR) type in Internet Explorer 8. This object works in a manner similar to XHR but in a way that is safe and secure for cross-domain communication. The XDR object implements part of the CORS specification. Here are some of the ways that XDR differs from XHR:

- Cookies are neither sent with requests nor received with responses.
- There is no access to set request headers other than `Content-Type`.
- There is no access to response headers.
- Only GET and POST requests are supported.

These changes mitigate issues related to *cross-site request forgery* (CSRF) and cross-site scripting (XSS) attacks. The resource being requested can dynamically decide whether to set the `Access-Control-Allow-Origin` header based on any data it deems appropriate: user-agent, referrer, and so on. As part of the request, an `Origin` header is sent with a value indicating the origin domain of the request, allowing the remote resource to recognize an XDR request explicitly.

XDR object usage looks very similar to XHR object use. You create a new instance of `XDomainRequest`, call the `open()` method, and then call the `send()` method. Unlike the `open()` method on XHR objects, the one on XDR objects accepts only two arguments: the request type and the URL.

All XDR requests are executed asynchronously, and there is no way to create a synchronous request. When a request has returned, a `load` event fires and the `responseText` property is filled with the response, as follows:



```
var xhr = new XDomainRequest();
xhr.onload = function(){
    alert(xhr.responseText);
};
xhr.open("get", "http://www.somewhere-else.com/page/");
xhr.send(null);
```

[XDomainRequestExample01.htm](#)

When the response is received, you have access to only the raw text of the response; there is no way to determine the status code of the response. The `load` event is fired for all valid responses and an `error` event is fired for all failures, including the lack of an `Access-Control-Allow-Origin` header on the response. Unfortunately, you receive no additional information about the error that occurred, so just knowing that the request was unsuccessful must be enough. To detect an error, assign an `onerror` event handler, as shown in this example:

```
var xhr = new XDomainRequest();
xhr.onload = function(){
    alert(xhr.responseText);
};
xhr.onerror = function(){
    alert("An error occurred.");
};
xhr.open("get", "http://www.somewhere-else.com/page/");
xhr.send(null);
```

[XDomainRequestExample01.htm](#)



Because there are so many ways an XDR request can fail, you should always use an onerror event handler to capture the occurrence; otherwise it will fail silently.

You can stop a request before it returns by calling `abort()` as follows:

```
xdr.abort(); //stop the request
```

Also similar to XHR, the XDR object supports the `timeout` property and the `ontimeout` event handler. Here's an example:

```
var xhr = new XDomainRequest();
xhr.onload = function(){
    alert(xhr.responseText);
};
xhr.onerror = function(){
    alert("An error occurred.");
};
xhr.timeout = 1000;
xhr.ontimeout = function(){
    alert("Request took too long.");
};
xhr.open("get", "http://www.somewhere-else.com/page/");
xhr.send(null);
```

This example times out after one second, at which point the `ontimeout` event handler is called.

To allow for POST requests, the XDR object exposes a `contentType` property that can be used to indicate the format of the posted data, as shown in this example:

```
var xhr = new XDomainRequest();
xhr.onload = function(){
    alert(xhr.responseText);
};
xhr.onerror = function(){
    alert("An error occurred.");
};
xhr.open("post", "http://www.somewhere-else.com/page/");
xhr.contentType = "application/x-www-form-urlencoded";
xhr.send("name1=value1&name2=value2");
```

This property is the only access to header information through the XDR object.

CORS in Other Browsers

Firefox 3.5+, Safari 4+, Chrome, Safari for iOS, and WebKit for Android all support CORS natively through the `XMLHttpRequest` object. When attempting to open a resource on a different origin, this behavior automatically gets triggered without any extra code. To make a request to a resource on

another domain, the standard XHR object is used with an absolute URL specified in `open()`, such as this:

```
var xhr = createXHR();
xhr.onreadystatechange = function() {
    if (xhr.readyState == 4) {
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
};

xhr.open("get", "http://www.somewhere-else.com/page/", true);
xhr.send(null);
```

Unlike the XDR object in Internet Explorer, the cross-domain XHR object allows access to the `status` and `statusText` properties and allows synchronous requests. There are some additional limitations on a cross-domain XHR object that are necessary for security purposes. They are as follows:

- Custom headers cannot be set using `setRequestHeader()`.
- Cookies are neither sent nor received.
- The `getAllResponseHeaders()` method always returns an empty string.

Since the same interface is used for both same- and cross-domain requests, it's best to always use a relative URL when accessing a local resource, and an absolute URL when accessing a remote resource. This disambiguates the use case and can prevent problems such as limiting access to header and/or cookie information for local resources.

Preflighted Requests

CORS allows the use of custom headers, methods other than GET or POST, and different body content types through a transparent mechanism of server verification called *preflighted requests*. When you try to make a request with one of the advanced options, a “preflight” request is made to the server. This request uses the OPTIONS method and sends the following headers:

- `Origin` — Same as in simple requests.
- `Access-Control-Request-Method` — The method that the request wants to use.
- `Access-Control-Request-Headers` — (Optional) A comma-separated list of the custom headers being used.

Here's an example assuming a POST request with a custom header called NCZ:

```
Origin: http://www.nczonline.net
Access-Control-Request-Method: POST
Access-Control-Request-Headers: NCZ
```

During this request, the server can determine whether or not it will allow requests of this type. The server communicates this to the browser by sending the following headers in the response:

- Access-Control-Allow-Origin — Same as in simple requests.
- Access-Control-Allow-Methods — A comma-separated list of allowed methods.
- Access-Control-Allow-Headers — A comma-separated list of headers that the server will allow.
- Access-Control-Max-Age — The amount of time in seconds that this preflight request should be cached for.

For example:

```
Access-Control-Allow-Origin: http://www.nczonline.net
Access-Control-Allow-Methods: POST, GET
Access-Control-Allow-Headers: NCZ
Access-Control-Max-Age: 1728000
```

Once a preflight request has been made, the result is cached for the period of time specified in the response; you'll only incur the cost of an extra HTTP request the first time a request of this type is made.

Firefox 3.5+, Safari 4+, and Chrome all support preflighted requests; Internet Explorer through version 10 does not.

Credentialed Requests

By default, cross-origin requests do not provide credentials (cookies, HTTP authentication, and client-side SSL certificates). You can specify that a request should send credentials by setting the `withCredentials` property to `true`. If the server allows credentialed requests, then it responds with the following HTTP header:

```
Access-Control-Allow-Credentials: true
```

If a credentialed request is sent and this header is not sent as part of the response, then the browser doesn't pass the response to JavaScript (`responseText` is an empty string, `status` is 0, and `onerror()` is invoked). Note that the server can also send this HTTP header as part of the preflight response to indicate that the origin is allowed to send credentialed requests.

Firefox 3.5+, Safari 4+, and Chrome all support the `withCredentials` property. Internet Explorer through version 10 still does not.

Cross-Browser CORS

Even though all browsers don't natively support the same level of CORS, all supporting browsers do support simple (nonpreflighted, noncredentialed) requests, so it makes sense to have a cross-browser solution. The easiest way to determine if the XHR object supports CORS is to check for the existence of the `withCredentials` property. You can then couple with the existence of the `XDomainRequest` object to cover all browsers:



```

function createCORSRequest(method, url){
    var xhr = new XMLHttpRequest();
    if ("withCredentials" in xhr){
        xhr.open(method, url, true);
    } else if (typeof XDomainRequest != "undefined"){
        xhr = new XDomainRequest();
        xhr.open(method, url);
    } else {
        xhr = null;
    }
    return xhr;
}

var request = createCORSRequest("get", "http://www.somewhere-else.com/page/");
if (request){
    request.onload = function(){
        //do something with request.responseText
    };
    request.send();
}

```

CrossBrowserCORSRequestExample01.htm

The `XMLHttpRequest` object in Firefox, Safari, and Chrome has similar enough interfaces to the Internet Explorer `XDomainRequest` object that this pattern works fairly well. The common interface properties/methods are:

- `abort()` — Use to stop a request that's already in progress.
- `onerror` — Use instead of `onreadystatechange` to detect errors.
- `onload` — Use instead of `onreadystatechange` to detect successes.
- `responseText` — Use to get contents of response.
- `send()` — Use to send the request.

Each of these can be used on the object returned from `createCORSRequest()` and will work the same in each browser.

ALTERNATE CROSS-DOMAIN TECHNIQUES

Before CORS came about, achieving cross-domain Ajax communication was a bit trickier. Developers came to rely on parts of the DOM that could perform cross-domain requests as a simple way to make certain types of requests without using the XHR object. Despite the ubiquity of CORS, these techniques are still popular because they don't involve changes on the server.

Image Pings

One of the first techniques for cross-domain communication was through the use of the `` tag. Images can be loaded cross-domain by any page without worrying about restrictions. This

is the main way that online advertisements track views. As discussed in Chapter 13, you can also dynamically create images and use their `onload` and `onerror` event handlers to tell you when the response has been received.

Dynamically creating images is often used for *image pings*. Image pings are simple, cross-domain, one-way communication with the server. The data is sent via query-string arguments and the response can be anything, though typically it's a pixel image or a 204 response. The browser can't get any specific data back from an image ping but it can tell when the response has been received by listening for the `load` and `error` events. Here's a simple example:



Available for download on
Wrox.com

```
var img = new Image();
img.onload = img.onerror = function(){
    alert("Done!");
};
img.src = "http://www.example.com/test?name=Nicholas";
```

ImagePingExample01.htm

This example creates a new instance of `Image` and then assigns both the `onload` and the `onerror` event handlers to the same function. This ensures that regardless of the response, you'll be notified when the request has completed. The request begins when the `src` property is set and this example is sending along a `name` parameter.

Image pings are frequently used for tracking user clicks on a page or dynamic ad impressions. The two main downsides to image pings are that you can only send GET requests and you cannot access the response text from the server. This is why image pings are best used for one-way communication between the browser and the server.

JSONP

JSONP is short for “JSON with padding” and is a special variant of JSON that has become popular for web services. JSONP looks just like JSON except that the data is wrapped within what looks like a function call. For example:

```
callback({ "name": "Nicholas" });
```

The JSONP format is made up of two parts: the callback and the data. The callback is the function that should be called on the page when the response has been received. Typically the name of the callback is specified as part of the request. The data is simply the JSON data to pass to the function. A typical JSONP request looks like this:

```
http://freegeoip.net/json/?callback=handleResponse
```

This URL is for a JSONP geolocation service. It's quite common to have the callback parameter specified as query-string argument for JSONP services, and in this case, I've specified the callback function name to be `handleResponse()`.

JSONP is used through dynamic `<script>` elements (see Chapter 13 for details), assigning the `src` to a cross-domain URL. The `<script>` element, similar to ``, is capable of loading resources

from other domains without restriction. Because JSONP is valid JavaScript, the JSONP response gets pulled into the page and executed immediately upon completion. Here's an example:



```
function handleResponse(response) {
    alert("You're at IP address " + response.ip + ", which is in " +
          response.city + ", " + response.region_name);
}

var script = document.createElement("script");
script.src = "http://freegeoip.net/json/?callback=handleResponse";
document.body.insertBefore(script, document.body.firstChild);
```

[JSONPExample01.htm](#)

This example displays your IP address and location information from the geolocation service.

JSONP is very popular for web developers because of its simplicity and ease of use. Its advantage over image pings is that you can access the response text directly, allowing bidirectional communication between browser and server. There are, however, a couple of downsides to using JSONP.

First, you're pulling executable code into your page from another domain. If that domain isn't trusted, it could very easily swap the response for something more malicious, and you would have no recourse aside from removing the JSONP call altogether. When using a web service that you don't operate, make sure that it comes from a trusted source.

The second downside is that there is no easy way to determine that a JSONP request has failed. Although HTML5 has specified an `onerror` event handler for `<script>` elements, it hasn't yet been implemented by any browser. Developers frequently used timers to see if a response has or has not been received within a set amount of time, but even this is tricky because not every user has the same connection speed and bandwidth.

Comet

Comet is a term coined by Alex Russell to describe a more advanced Ajax technique sometimes referred to as *server push*. Whereas Ajax is described as the page requesting data from the server, Comet is described as the server pushing data to the page. This approach allows information to come into the page in a manner closer to real time, making it ideal for information such as sports scores or stock market prices.

There are two popular approaches to Comet: *long polling* and *streaming*. Long polling is a new spin on traditional polling (also called *short polling*) where the browser sends a request to the server in regular intervals to see if there's any data. Figure 21-1 shows a timeline of how short polling works.

Long polling flips short polling around. The page initiates a request to the server and the server holds that connection open until it has data to send. Once the data is sent, the connection is closed by the browser and a new connection is

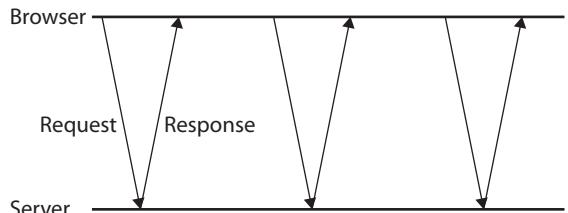


FIGURE 21-1

immediately opened up to the server. This process continues for as long as the page is open in the browser. Figure 21-2 shows a timeline of how long polling works.

In both short polling and long polling, the browser needs to initiate the connection to the server before data can be received. The big difference is how the server handles sending data back. Short polling sends a response

immediately regardless of the data availability, while long polling waits to send a response. The advantage of polling techniques is that all browsers support this through the XHR object and using `setTimeout()`. You just need to manage when the requests are sent.

The second popular approach for Comet is HTTP streaming. Streaming is different than either polling technique, because it uses a single HTTP connection for the entire lifetime of the page. The browser sends a request to the server and the server holds that connection open, periodically sending data through the connection to the server. For example, a PHP server might have a script that looks like this:

```
<?php
    $i = 0;
    while(true) {

        //output some data and then flush the output buffer immediately
        echo "Number is $i";
        flush();

        //wait a few seconds
        sleep(10);

        $i++;
    }
}
```

All server-side languages support the notion of printing to the output buffer and then flushing (sending the contents of the output buffer to the client). This is the core of HTTP streaming.

The XHR object can be used to achieve HTTP streaming in Firefox, Safari, Opera, and Chrome by listening for the `readystatechange` event and focusing on `readyState` 3. A `readyState` of 3 will fire periodically in all of these browsers as data is being received from the server. At that point, the `responseText` property contains all of the data received, which means you need to slice off the newest piece by keeping track of what was sent previously. An HTTP streaming implementation using XHR looks like this:



Available for
download on
Wrox.com

```
function createStreamingClient(url, progress, finished) {
    var xhr = new XMLHttpRequest(),
        received = 0;

    xhr.open("get", url, true);
    xhr.onreadystatechange = function() {

```

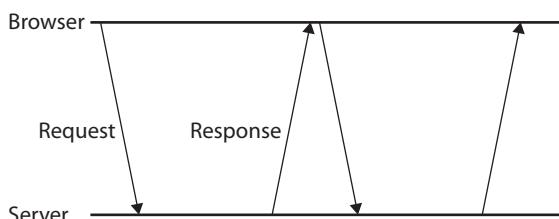


FIGURE 21-2

```

var result;

if (xhr.readyState == 3){

    //get only the new data and adjust counter
    result = xhr.responseText.substring(received);
    received += result.length;

    //call the progress callback
    progress(result);

} else if (xhr.readyState == 4){
    finished(xhr.responseText);
}

};

xhr.send(null);
return xhr;
}

var client = createStreamingClient("streaming.php", function(data) {
    alert("Received: " + data);
}, function(data){
    alert("Done!");
});
```

[HTTPStreamingExample01.htm](#)

The `createStreamingClient()` function accepts three arguments: the URL to connect to, the function to call when more data has been received, and the function to call when the connection has closed. You may or may not want to restart a connection once it has been closed, so it's good to keep track of that.

Whenever the `readystatechange` event is fired and the `readyState` is 3, the `responseText` is sliced to return only the most recent data. The `received` variable keeps track of how many characters have already been processed, incrementing each time `readyState` 3 is processed. Then, the `progress` callback function is executed and the new data is passed in. When the `readyState` is 4, the `finished` callback is executed and the entire content of the response is passed in.

Even though this example is relatively simple and works in most browsers (Internet Explorer being the exception), the management of connections for Comet patterns is easy to get wrong and requires time to perfect. The browser community believes that Comet is an important part of the Web's future, and so two new interfaces were created to make Comet easier.

Server-Sent Events

Server-Sent Events (SSE) is an API and pattern for read-only Comet interactions. The SSE API creates a one-way HTTP connection to the server through which the server can pass as much or as little information as necessary. The server response must have a MIME type of `text/event-stream` and outputs the information in a specific format that the browser API consumes and makes available through JavaScript. SSE supports short polling, long polling, and HTTP streaming and, as

such, automatically determines when to reconnect to the server if it gets disconnected. The result is an extremely simple and useful API that makes Comet easy.

Server-Sent Events are supported in Firefox 6+, Safari 5+, Opera 11+, Chrome, and Safari for iOS 4+.

The API

The JavaScript API for SSE is similar to other recent JavaScript messaging APIs. To subscribe to a new event stream, you start by creating a new `EventSource` object and pass in the entry point:

```
var source = new EventSource("myevents.php");
```

Note that the referenced URL must be on the same origin (scheme, domain, and port) as the page in which the object is created. The `EventSource` instance has a `readyState` property that is set to 0 to indicate it's connecting to the server, 1 to indicate an open connection, and 2 to indicate a closed connection.

There are also three events:

- `open`, which is fired when the connection is established
- `message`, which is fired when a new event is received from the server
- `error`, which is fired when no connection can be made

In general usage, the `onmessage` event handler is likely the one you'll use the most:

```
source.onmessage = function(event){  
    var data = event.data;  
    //do something with the data  
};
```

Information sent back from the server is returned via `event.data` as a string.

By default, the `EventSource` object will attempt to keep the connection alive with the server. If the connection is closed, a reconnect is attempted. This means that Server-Sent Events work with long polling and HTTP streaming. You can force the object to disconnect immediately and stop attempting to reconnect by calling the `close()` method:

```
source.close();
```

The Event Stream

The server events are sent along a long-lasting HTTP response with a MIME type of `text/event-stream`. The format of the response is plain text and, in its simplest form, is made up of the prefix `data:` followed by text, such as:

```
data: foo  
  
data: bar  
  
data: foo  
data: bar
```

The first part of this stream fires a message event with event.data set to “foo”; the second part fires a message event with event.data set to “bar”; the third fires a message event with event.data set to “foo\nbar” (note the newline character in between). When there are two or more consecutive lines beginning with data:, it is interpreted as a multiline piece of data and the values are concatenated together with a newline character. The message event is never fired until a blank line is encountered after a line containing data:, so be certain to include that extra newline when generating the event stream on the server.

You can also associate an ID with a particular event by including an id: line before or after the data: line(s):

```
data: foo  
id: 1
```

By setting an ID, the EventSource object keeps track of the last event fired. If the connection is dropped, a special HTTP header called Last-Event-ID is sent along with the request so that the server can determine which event is appropriate to fire next. This is important for keeping sequential pieces of data in the correct order over multiple connections.

Web Sockets

Web Sockets is one of the most talked-about new browser APIs. The goal of Web Sockets is to provide full-duplex, bidirectional communication with the server over a single, long-lasting connection. When a Web Socket is created in JavaScript, an HTTP request is sent to the server to initiate a connection. When the server responds, the connection uses HTTP upgrade to switch from HTTP to the Web Socket protocol. This means that Web Sockets cannot be implemented with a standard HTTP server and must use a specialized server supporting the protocol to work properly.

Since Web Sockets uses a custom protocol, the URL scheme is slightly different. Instead of using the http:// or https:// schemes, there are ws:// for an unsecured connection and wss:// for a secured connection. When specifying a Web Socket URL, you must include the scheme since other schemes may be supported in the future.

The advantage of using a custom protocol over HTTP is that very small amounts of data, unencumbered by the byte overhead of HTTP, can be sent between the client and the server. Using smaller data packets makes Web Sockets ideal for mobile applications where bandwidth and latency are a problem. The disadvantage of using a custom protocol is that it has taken longer to define protocol than the JavaScript API. Web Sockets has been stalled repeatedly as people have found issues with the protocol, in terms of both consistency and security. Firefox 4 and Opera 11 both had Web Sockets enabled by default but disabled it just before release because of security concerns. Web Sockets are now supported in Firefox 6+, Safari 5+, Chrome, and Safari for iOS 4+.

The API

To create a new Web Socket, instantiate a `WebSocket` object and pass in the URL that will provide the connection:

```
var socket = new WebSocket("ws://www.example.com/server.php");
```

Note that you must pass in an absolute URL to the `WebSocket` constructor. The same-origin policy does not apply to Web Sockets, so you can open a connection to any site. It is completely up to the server whether or not it will communicate with a page from a particular origin. (It can determine from where the request originated using information in the handshake.)

The browser attempts to create the connection as soon as the `WebSocket` object is instantiated. Similar to XHR, `WebSocket` has a `readyState` property that indicates the current state. The values, however, are different from those for XHR and are as follows:

- `WebSocket.OPENING` (0) — The connection is being established.
- `WebSocket.OPEN` (1) — The connection has been established.
- `WebSocket.CLOSING` (2) — The connection is beginning to close.
- `WebSocket.CLOSE` (3) — The connection is closed.

There is no `readystatechange` event for `WebSocket`; however, there are other events that correspond to the various states. The `readyState` always starts at 0.

You can close a Web Socket connection at any time using the `close()` method:

```
socket.close();
```

Upon calling `close()`, the `readyState` immediately changes to 2 (closing) and will transition to 3 when complete.

Sending/Receiving Data

Once a Web Socket is open, you can both send data over and receive data from the connection. To send data to the server, use the `send()` method and pass in any string, for example:

```
var socket = new WebSocket("ws://www.example.com/server.php");
socket.send("Hello world!");
```

Since Web Sockets can only send plain text over the connection, you'll need to serialize more complex data structures before sending them over the connection. The following serializes data into a JSON string and then sends the string to the server:

```
var message = {
  time: new Date(),
  text: "Hello world!",
  clientId: "asdfp8734rew"
};

socket.send(JSON.stringify(message));
```

The server would then need to parse the returned JSON to access the data.

When the server sends a message to the client, a `message` event is fired on the `WebSocket` object. The `message` event works similar to other messaging protocols, with the payload available through the `event.data` property:

```
socket.onmessage = function(event) {
    var data = event.data;

    //do something with data
};
```

As with data that is sent to the server via `send()`, data returned in `event.data` is always a string. If you are expecting another data format, then you must manually parse the data.

Other Events

The `WebSocket` object has three more events that fire during the lifetime of the connection:

- `open` — Fires when the connection has been successfully made.
- `error` — Fires when an error occurs. The connection is unable to persist.
- `close` — Fires when the connection is closed.

The `WebSocket` object doesn't support DOM Level 2 event listeners, so you need to use DOM Level 0 style event handlers for each:

```
var socket = new WebSocket("ws://www.example.com/server.php");

socket.onopen = function(){
    alert("Connection established.");
};

socket.onerror = function(){
    alert("Connection error.");
};

socket.onclose = function(){
    alert("Connection closed.");
};
```

Of these three events, only the `close` event has additional information on the `event` object. There are three additional properties on the `event` object: `wasClean`, a Boolean indicating if the connection was closed cleanly; `code`, a numeric status code sent from the server; and `reason`, a string containing a message sent from the server. You may want to use this information either to display to the user or to log for analytics:

```
socket.onclose = function(event) {
    console.log("Was clean? " + event.wasClean + " Code=" + event.code + " Reason=" + event.reason);
};
```

SSE versus Web Sockets

When determining whether to use SSE or Web Sockets for a particular use case, you can take several factors into account. First, do you have the flexibility to set up a Web Socket server? Since the Web Socket protocol is not HTTP, your existing web servers may not be capable of Web Socket

communication. SSE works over normal HTTP, so you may be able to use your existing web servers to perform this type of communication.

The second question to ask is whether you need bidirectional communication. If the use case requires only read-only access to server data (such as sports scores), then SSE may be easier to implement. If the use case requires full bidirectional support (such as a chat room), then Web Sockets may be a better choice. Keep in mind that you can still implement bidirectional communication with a combination of XHR and SSE should Web Sockets not be an option for you.

SECURITY

There has been a lot published about Ajax and Comet security; in fact, there are entire books dedicated to the topic. Security considerations for large-scale Ajax applications are vast, but there are some basic things to understand about Ajax security in general.

First, any URL that can be accessed via XHR can also be accessed by a browser or a server. For example, consider the following URL:

```
/getuserinfo.php?id=23
```

If a request is made to this URL, it will presumably return some data about a user whose ID is 23. There is nothing to stop someone from changing the URL to a user ID of 24 or 56 or any other value. The `getuserinfo.php` file must know whether the requestor actually has access to the data that is being requested; otherwise you have left the server wide open to relay data about anyone.

When an unauthorized system is able to access a resource, it is considered a cross-site request forgery (CSRF) attack. The unauthorized system is making itself appear to be legitimate to the server handling the request. Ajax applications, large and small, have been affected by CSRF attacks ranging from benign proof-of-vulnerability attacks to malicious data-stealing or data-destroying attacks.

The prevailing theory of how to secure URLs accessed via XHR is to validate that the sender has access to the resource. This can be done in the following ways:

- Requiring SSL to access resources that can be requested via XHR.
- Requiring a computed token to be sent along with every request.

Please recognize that the following are ineffective against CSRF attacks:

- Requiring a POST instead of a GET — This is easily changed.
- Using the referrer as a determination of origin — Referrers are easily spoofed.
- Validating based on cookie information — Also easily spoofed.

The XHR object offers something that seems secure at first glance but ultimately is quite insecure. The `open()` method actually has two more arguments: a username and a password that should be sent along with the request. This can be used to send requests to pages via SSL on a server, as in this example:

```
xhr.open("get", "example.php", true, "username", "password"); //AVOID!!!!
```



Even though this username/password feature is possible, you should avoid using this feature. Storing usernames and passwords in JavaScript is highly insecure, because anyone with a JavaScript debugger can view what is stored in the variables, exposing your username and password in plain text.

SUMMARY

Ajax is a method for retrieving data from the server without refreshing the current page. Ajax has the following characteristics:

- The central object responsible for the growth of Ajax is the XMLHttpRequest (XHR) object.
- This object was created by Microsoft and first introduced in Internet Explorer 5 as a way to retrieve XML data from the server in JavaScript.
- Since that time, Firefox, Safari, Chrome, and Opera have all duplicated the implementation, and the W3C has written a specification defining the XHR behavior, making XHR a Web standard.
- Though there are some differences in implementations, the basic usage of the XHR object is relatively normalized across all browsers and can therefore safely be used in web applications.

One of the major constraints on XHR is the same-origin policy that limits communication to the same domain, using the same port, and with the same protocol. Any attempts to access resources outside of these restrictions cause a security error, unless an approved cross-domain solution is used. The solution is called Cross-Origin Resource Sharing (CORS) and is supported in Internet Explorer 8+ through the XDomainRequest object and other browsers natively through the XHR object.

Image pings and JSONP are other techniques for cross-domain communication, though they are less robust than CORS.

Comet is an extension of Ajax where the server is able to push data to the client almost in real time. There are two primary approaches to Comet: long polling and HTTP streaming. All browsers support long polling, while only some natively support HTTP streaming. Server-Sent Events (SSE) is a browser API for Comet interactions that supports both long polling and HTTP streaming.

Web Sockets are a full-duplex, bidirectional communication channel with the server. Unlike other solutions, Web Sockets do not use HTTP but rather use a custom protocol designed to deliver small pieces of data quickly. This requires a different web server but gives a speed advantage.

The buzz around Ajax and Comet encouraged more developers to learn JavaScript and helped usher in a resurgence of interest in web development. Ajax-related concepts are still relatively new and will undoubtedly continue to evolve.



The topic of Ajax is substantial, and a full discussion is beyond the scope of this book. For further information on this topic, read Professional Ajax, 2nd Edition (Wiley, 2007; ISBN: 978-0-470-10949-6).

22

Advanced Techniques

WHAT'S IN THIS CHAPTER?

- Using advanced functions
- Tamper-proofing your objects
- Yielding with timers

JavaScript is an incredibly flexible language that can be used in a variety of styles. Typically, JavaScript is used in either a procedural manner or an object-oriented one. The language, however, is capable of much more intricate and interesting patterns because of its dynamic nature. These techniques make use of ECMAScript language features, BOM extensions, and DOM functionality to achieve powerful results.

ADVANCED FUNCTIONS

Functions are one of the most interesting parts of JavaScript. They can be quite simple and procedural in nature, or they can be quite complex and dynamic. Additional functionality can be achieved through the use of closures. Furthermore, function pointers are very easy to work with, since all functions are objects. All of this makes JavaScript functions both interesting and powerful. The following sections outline some of the advanced ways that functions can be used in JavaScript.

Safe Type Detection

JavaScript's built-in type detection mechanisms aren't foolproof and, in fact, can sometimes give both false positives and false negatives. The `typeof` operator, for example, has several quirks that can make it unreliable in detecting certain types of data. Since Safari (through version 4) returns "function" when `typeof` is applied to a regular expression, it's difficult to definitively determine if a value is a function.

The `instanceof` operator is also problematic in that it's difficult to use when multiple global scopes are present, such as when there are multiple frames. The classic example of this problem, as mentioned in Chapter 5, is attempting to identify an object as an array using the following code:

```
var isArray = value instanceof Array;
```

This code returns `true` only if `value` is an array and was created in the same global scope as the `Array` constructor. (Remember, `Array` is a property of `window`.) If `value` is an array from another frame, this code returns `false`.

Yet another problem with type detection comes when trying to determine if an object is a native implementation or a developer-defined one. This problem came to the forefront as browsers began to natively implement the `JSON` object. Since many were already using Douglas Crockford's `JSON` library, which defined a global `JSON` object, developers struggled to determine which object was present on the page.

The solution to all of these problems is the same. The native `toString()` method of `Object` can be called with any value to return a string in the format “[object NativeConstructorName]”. Each object has an internal `[[Class]]` property that specifies the constructor name that is returned as part of this string. For example:

```
alert(Object.prototype.toString.call(value)); // "[object Array]"
```

Since the native constructor name for arrays is the same regardless of the global context in which it was created, using `toString()` returns a consistent value. This allows you to create a function such as:

```
function isArray(value) {
    return Object.prototype.toString.call(value) == "[object Array]";
}
```

The same approach can be used to determine if a value is a native function or regular expression:

```
function isFunction(value) {
    return Object.prototype.toString.call(value) == "[object Function]";
}
function isRegExp(value) {
    return Object.prototype.toString.call(value) == "[object RegExp]";
}
```

Note that `isFunction()` will return `false` in Internet Explorer for any functions that are implemented as COM objects rather than native JavaScript functions (see Chapter 10 for further details).

This technique is also largely in use for identifying the native `JSON` object. The `toString()` method of `Object` can't determine constructor names for nonnative constructors, so any objects that are instances of developer-defined constructors return “[object Object]”. Several JavaScript libraries contain code similar to the following:

```
var isNativeJSON = window.JSON && Object.prototype.toString.call(JSON) ==
    "[object JSON]";
```



Being able to discern the difference between native and nonnative JavaScript objects is very important in web development to ensure you know the available capabilities of an object. This technique can be used on any object to definitively make this determination.



Keep in mind that it's possible to assign `Object.prototype.toString()` to a different value. The technique discussed in this section assumes that `Object.prototype.toString()` is the native version and has not been overwritten by a developer.

Scope-Safe Constructors

Chapter 6 covered the definition and usage of constructors for defining custom objects. You'll recall that a constructor is simply a function that is called using the `new` operator. When used in this way, the `this` object used inside the constructor points to the newly created object instance, as in this example:



Available for
download on
[Wrox.com](#)

```
function Person(name, age, job){  
    this.name = name;  
    this.age = age;  
    this.job = job;  
}  
  
var person = new Person("Nicholas", 29, "Software Engineer");
```

[ScopeSafeConstructorsExample01.htm](#)

In this example, the `Person` constructor assigns three properties using the `this` object: `name`, `age`, and `job`. When used with the `new` operator, a new `Person` object is created, and the properties are assigned onto it. The problem occurs when the constructor is called without the `new` operator. Since the `this` object is bound at runtime, calling `Person()` directly maps `this` to the global object (`window`), resulting in accidental augmentation of the wrong object. For example:

```
var person = Person("Nicholas", 29, "Software Engineer");  
alert(window.name);      // "Nicholas"  
alert(window.age);       // 29  
alert(window.job);       // "Software Engineer"
```

[ScopeSafeConstructorsExample01.htm](#)

Here, the `window` object has been augmented with the three properties intended for a `Person` instance, because the constructor was called as a regular function, omitting the `new` operator. This issue occurs as a result of late binding of the `this` object, which was resolved to `window` in this case. Since the `name` property of `window` is used to identify link targets and frames, this accidental overwriting of the property could lead to other errors on the page. The solution is to create a *scope-safe constructor*.



Available for
download on
Wrox.com

```
function Person(name, age, job) {
    if (this instanceof Person) {
        this.name = name;
        this.age = age;
        this.job = job;
    } else {
        return new Person(name, age, job);
    }
}

var person1 = Person("Nicholas", 29, "Software Engineer");
alert(window.name);      // ""
alert(person1.name);     // "Nicholas"

var person2 = new Person("Shelby", 34, "Ergonomist");
alert(person2.name);     // "Shelby"
```

ScopeSafeConstructorsExample02.htm

The `Person` constructor in this code adds an `if` statement that checks to ensure that the `this` object is an instance of `Person`, which indicates that either the `new` operator was used or the constructor was called in the context of an existing `Person` instance. In either case, the object initialization continues as usual. If `this` is not an instance of `Person`, then the constructor is called again with the `new` operator and that value is returned. The result is that calling the `Person` constructor either with or without the `new` operator returns a new instance of `Person`, avoiding any accidental property setting on the global object.

There is a caveat to scope-safe constructors. By implementing this pattern, you are locking down the context in which the constructor can be called. If you're using the constructor-stealing pattern of inheritance without also using prototype chaining, your inheritance may break. Here is an example:

```
function Polygon(sides) {
    if (this instanceof Polygon) {
        this.sides = sides;
        this.getArea = function() {
            return 0;
        };
    } else {
        return new Polygon(sides);
    }
}

function Rectangle(width, height) {
    Polygon.call(this, 2);
    this.width = width;
    this.height = height;
    this.getArea = function() {
        return this.width * this.height;
    };
}
```

```
var rect = new Rectangle(5, 10);
alert(rect.sides); //undefined
```

ScopeSafeConstructorsExample03.htm

In this code, the `Polygon` constructor is scope-safe, whereas the `Rectangle` constructor is not. When a new instance of `Rectangle` is created, it should inherit the `sides` property from `Polygon` through the use of `Polygon.call()`. However, since the `Polygon` constructor is scope-safe, the `this` object is not an instance of `Polygon`, so a new `Polygon` object is created and returned. The `this` object in the `Rectangle` constructor is not augmented, and the value returned from `Polygon.call()` is not used, so there is no `sides` property on the `Rectangle` instance.

This issue resolves itself if prototype chaining or parasitic combination is used with constructor stealing. Consider the following example:



```
function Polygon(sides) {
    if (this instanceof Polygon) {
        this.sides = sides;
        this.getArea = function(){
            return 0;
        };
    } else {
        return new Polygon(sides);
    }
}

function Rectangle(width, height){
    Polygon.call(this, 2);
    this.width = width;
    this.height = height;
    this.getArea = function(){
        return this.width * this.height;
    };
}

Rectangle.prototype = new Polygon();

var rect = new Rectangle(5, 10);
alert(rect.sides); //2
```

ScopeSafeConstructorsExample04.htm

In this rewritten code, an instance of `Rectangle` is also an instance of `Polygon`, so `Polygon.call()` works as it should, ultimately adding a `sides` property to the `Rectangle` instance.

Scope-safe constructors are helpful in environments where multiple developers are writing JavaScript code to run on the same page. In that context, accidental changes to the global object may result in errors that are often difficult to track down. Scope-safe constructors are recommended as a best practice unless you're implementing inheritance based solely on constructor stealing.

Lazy Loading Functions

Because of differences in browser behavior, most JavaScript code contains a significant amount of if statements that fork execution toward code that should succeed. Consider the following `createXHR()` function from the previous chapter:

```
function createXHR(){
    if (typeof XMLHttpRequest != "undefined"){
        return new XMLHttpRequest();
    } else if (typeof ActiveXObject != "undefined"){
        if (typeof arguments.callee.activeXString != "string"){
            var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
                "MSXML2.XMLHttp"],
                i, len;

            for (i=0,len=versions.length; i < len; i++){
                try {
                    new ActiveXObject(versions[i]);
                    arguments.callee.activeXString = versions[i];
                    break;
                } catch (ex){
                    //skip
                }
            }
        }
        return new ActiveXObject(arguments.callee.activeXString);
    } else {
        throw new Error("No XHR object available.");
    }
}
```

Every time `createXHR()` is called, it goes through and checks which capability is supported for the browser. First it checks for native XHR, then it tests for ActiveX-based XHR, and finally it throws an error if neither is found. This happens each and every time the function is called, even though the result of this branching won't change from call to call: if the browser supports native XHR, it supports native XHR always, so the test becomes unnecessary. Code going through even a single if statement is slower than code with no if statements, so the code could run faster if the if statement weren't necessary every time. The solution is a technique called *lazy loading*.

Lazy loading means that the branching of function execution happens only once. There are two ways to accomplish lazy loading, the first is by manipulating the function the first time it is called. During that first call, the function is overwritten with another function that executes in the appropriate way such that any future calls to the function needn't go through the execution branch. For example, the `createXHR()` function can be rewritten to use lazy loading in this way:



Available for download on Wrox.com

```
function createXHR(){
    if (typeof XMLHttpRequest != "undefined"){
        createXHR = function(){
            return new XMLHttpRequest();
        };
    }
}
```

```

    } else if (typeof ActiveXObject != "undefined") {
        createXHR = function(){
            if (typeof arguments.callee.activeXString != "string"){
                var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
                               "MSXML2.XMLHttp"],
                    i, len;

                for (i=0,len=versions.length; i < len; i++){
                    try {
                        new ActiveXObject(versions[i]);
                        arguments.callee.activeXString = versions[i];
                        break;
                    } catch (ex){
                        //skip
                    }
                }
            }

            return new ActiveXObject(arguments.callee.activeXString);
        };
    } else {
        createXHR = function(){
            throw new Error("No XHR object available.");
        };
    }
}

return createXHR();
}

```

[LazyLoadingExample01.htm](#)

In the lazy loading version of `createXHR()`, each branch of the `if` statement assigns a different function to the `createXHR` variable, effectively overwriting the original function. The last step is then to call the newly assigned function. The next time `createXHR()` is called, it will call the assigned function directly so the `if` statements won't be reevaluated.

The second lazy loading pattern is to assign the appropriate function immediately when the function is declared. So instead of taking a slight performance hit when the function is called for the first time, there's a slight performance hit when the code is loaded for the first time. Here's the previous example written using this pattern:



```

var createXHR = (function(){
    if (typeof XMLHttpRequest != "undefined"){
        return function(){
            return new XMLHttpRequest();
        };
    } else if (typeof ActiveXObject != "undefined"){
        return function(){
            if (typeof arguments.callee.activeXString != "string"){
                var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
                               "MSXML2.XMLHttp"],

```

```
i, len;

for (i=0,len=versions.length; i < len; i++){
    try {
        new ActiveXObject(versions[i]);
        arguments.callee.activeXString = versions[i];
        break;
    } catch (ex) {
        //skip
    }
}
}

return new ActiveXObject(arguments.callee.activeXString);
};

} else {
    return function(){
        throw new Error("No XHR object available.");
    };
}
})();
});
```

[LazyLoadingExample02.htm](#)

The approach used in this example is to create an anonymous, self-executing function that determines which of the different function implementations should be used. Note that the logic is exactly the same. The only things that have changed are the first line (using `var` to define the function), the addition of the self-executing anonymous function, and that each branch now returns the correct function definition so that it gets assigned to `createXHR()` immediately.

Lazy loading functions have an advantage in that you pay a performance penalty just once for forking the code. Which pattern is right for you is up to your unique requirements, but both patterns offer the advantage of ensuring that unnecessary code isn't being executed all the time.

Function Binding

An advanced technique that has become increasingly popular is *function binding*. Function binding involves creating a function that calls another function with a specific `this` value and with specific arguments. This technique is often used in conjunction with callbacks and event handlers to preserve code execution context while passing functions around as variables. Consider the following example:

```
var handler = {
    message: "Event handled",

    handleClick: function(event){
        alert(this.message);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", handler.handleClick);
```

In this example, an object called `handler` is created. The `handler.handleClick()` method is assigned as an event handler to a DOM button. When the button is clicked, the function is called, and an alert is displayed. Even though it may seem as if the alert should display "Event handled", it actually displays "undefined". The problem is that the context of `handler.handleClick()` is not being saved, so the `this` object ends up pointing to the DOM button instead of `handler` in most browsers. (In Internet Explorer through version 8, this points to `window`.) You can fix this problem using a closure, as shown in the following example:

```
var handler = {
    message: "Event handled",

    handleClick: function(event){
        alert(this.message);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", function(event){
    handler.handleClick(event);
});
```

This solution uses a closure to call `handler.handleClick()` directly inside the `onclick` event handler. Of course, this is a very specific solution to this specific piece of code. Creating multiple closures can lead to code that is difficult to understand and debug. Therefore, many JavaScript libraries have implemented a function that can bind a function to a specific context. Typically, this function is called `bind()`.

A simple `bind()` function takes a function and a context, returning a function that calls the given function in the given context with all arguments intact. The syntax is as follows:



Available for
download on
[Wrox.com](#)

```
function bind(fn, context){
    return function(){
        return fn.apply(context, arguments);
    };
}
```

[FunctionBindingExample01.htm](#)

This function is deceptively simple but is actually quite powerful. A closure is created within `bind()` that calls the passed-in function by using `apply()` and passing in the `context` object and the arguments. Note that the `arguments` object, as used here, is for the inner function, not for `bind()`. When the returned function is called, it executes the passed-in function in the given context and passes along all arguments. The `bind()` function is used as follows:

```
var handler = {
    message: "Event handled",

    handleClick: function(event){
        alert(this.message);
    }
};
```

```

};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", bind(handler.handleClick, handler));

```

FunctionBindingExample01.htm

In this example, the `bind()` function is used to create a function that can be passed into `EventUtil.addHandler()`, maintaining the context. The event object is also passed through to the function, as shown here:



```

var handler = {
    message: "Event handled",
    handleClick: function(event){
        alert(this.message + ":" + event.type);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", bind(handler.handleClick, handler));

```

FunctionBindingExample01.htm

The `handler.handleClick()` method gets passed the `event` object as usual, since all arguments are passed through the bound function directly to it.

ECMAScript 5 introduced a native `bind()` method on all functions to make this process even easier. Instead of defining your own `bind()` function, you can call the method directly on the function itself. For example:

```

var handler = {
    message: "Event handled",

    handleClick: function(event){
        alert(this.message + ":" + event.type);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", handler.handleClick.bind(handler));

```

FunctionBindingExample02.htm

The native `bind()` method works similarly to the one previously described in that you pass in the object that should be the value of `this`. The native `bind()` method is available in Internet Explorer 9+, Firefox 4+, and Chrome.

Bound functions are useful whenever a function pointer must be passed as a value and that function needs to be executed in a particular context. They are most commonly used for event handlers and

with `setTimeout()` and `setInterval()`. However, bound functions have more overhead than regular functions — they require more memory and are slightly slower because of multiple function calls — so it's best to use them only when necessary.

Function Currying

A topic closely related to function binding is *function currying*, which creates functions that have one or more arguments already set (also called *partial function application*). The basic approach is the same as function binding: use a closure to return a new function. The difference with currying is that this new function also sets some arguments to be passed in when the function is called.

Consider the following example:

```
function add(num1, num2) {
    return num1 + num2;
}

function curriedAdd(num2) {
    return add(5, num2);
}

alert(add(2, 3));      //5
alert(curriedAdd(3)); //8
```

This code defines two functions: `add()` and `curriedAdd()`. The latter is essentially a version of `add()` that sets the first argument to 5 in all cases. Even though `curriedAdd()` is not technically a curried function, it demonstrates the concept quite well.

Curried functions are typically created dynamically by calling another function and passing in the function to curry and the arguments to supply. The following function is a generic way to create curried functions:



Available for
download on
Wrox.com

```
function curry(fn) {
    var args = Array.prototype.slice.call(arguments, 1);
    return function() {
        var innerArgs = Array.prototype.slice.call(arguments),
            finalArgs = args.concat(innerArgs);
        return fn.apply(null, finalArgs);
    };
}
```

[FunctionCurryingExample01.htm](#)

The `curry()` function's primary job is to arrange the arguments of the returned function in the appropriate order. The first argument to `curry()` is the function that should be curried; all other arguments are the values to pass in. In order to get all arguments after the first one, the `slice()` method is called on the `arguments` object, and an argument of 1 is passed in, indicating that the returned array's first item should be the second argument. The `args` array then contains arguments from the outer function. For the inner function, the `innerArgs` array is created to contain all of the arguments that were passed in (once again using `slice()`). With the arguments from the

outer function and inner function now stored in arrays, you can use the `concat()` method to combine them into `finalArgs` and then pass the result into the function, using `apply()`. Note that this function doesn't take context into account, so the call to `apply()` passes in `null` as the first argument. The `curry()` function can be used as follows:



```
function add(num1, num2) {
    return num1 + num2;
}

var curriedAdd = curry(add, 5);
alert(curriedAdd(3)); //8
```

FunctionCurryingExample01.htm

In this example, a curried version of `add()` is created that has the first argument bound to 5. When `curriedAdd()` is called and 3 is passed in, the 3 becomes the second argument of `add()`, while the first is still 5, resulting in the sum of 8. You can also provide all function arguments, as shown in this example:

```
function add(num1, num2) {
    return num1 + num2;
}

var curriedAdd = curry(add, 5, 12);
alert(curriedAdd()); //17
```

FunctionCurryingExample01.htm

Here, the curried `add()` function provides both arguments, so there's no need to pass them in later. Function currying is often included as part of function binding, creating a more complex `bind()` function. For example:

```
function bind(fn, context){
    var args = Array.prototype.slice.call(arguments, 2);
    return function(){
        var innerArgs = Array.prototype.slice.call(arguments),
            finalArgs = args.concat(innerArgs);
        return fn.apply(context, finalArgs);
    };
}
```

FunctionCurryingExample02.htm

The major changes from the `curry()` function are the number of arguments passed into the function and how that affects the result of the code. Whereas `curry()` simply accepts a function to wrap, `bind()` accepts the function and a `context` object. That means the arguments for the bound function start at the third argument instead of the second, which changes the first call to `slice()`. The only other change is to pass in the `context` object to `apply()` on the third-to-last line. When

`bind()` is used, it returns a function that is bound to the given context and may have some number of its arguments set already. This can be useful when you want to pass arguments into an event handler in addition to the `event` object, such as this.



```
var handler = {
    message: "Event handled",

    handleClick: function(name, event) {
        alert(this.message + ":" + name + ":" + event.type);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", bind(handler.handleClick, handler,
    "my-btn"));
```

[FunctionCurryingExample02.htm](#)

In this updated example, the `handler.handleClick()` method accepts two arguments: the name of the element that you're working with and the `event` object. The name is passed into the `bind()` function as the third argument and then gets passed through to `handler.handleClick()`, which also receives the `event` object.

The ECMAScript 5 `bind()` method also implements function currying. Just pass in the additional arguments after the value for `this`:

```
var handler = {
    message: "Event handled",

    handleClick: function(name, event) {
        alert(this.message + ":" + name + ":" + event.type);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", handler.handleClick.bind(handler, "my-btn"));
```

[FunctionCurryingExample03.htm](#)

Curried and bound functions provide powerful dynamic function creation in JavaScript. The use of either `bind()` or `curry()` is determined by the requirement of a `context` object or the lack of one, respectively. They can both be used to create complex algorithms and functionality, although neither should be overused, because each function creates additional overhead.

TAMPER-PROOF OBJECTS

One of the long-lamented downsides of JavaScript is its shared nature: every object can be modified by any code running in the same context. This can lead to developers accidentally overwriting each other's code or, worse, overwriting a native object with incompatible changes. ECMAScript 5 sought to address this problem by allowing you to create *tamper-proof objects*.

Chapter 6 discussed the nature of properties on objects and how you can manually set each property's `[[Configurable]]`, `[[Writable]]`, `[[Enumerable]]`, `[[Value]]`, `[[Get]]`, and `[[Set]]` attributes to alter how the property behaves. In a similar manner, ECMAScript 5 adds several methods that allow you to specify how an entire object behaves.

One thing to keep in mind: once an object has been made tamper-proof, the operation cannot be undone.

Nonextensible Objects

By default, all objects in JavaScript are *extensible*, meaning that you can add additional properties and methods to the object at any time. For example, I can define an object and then later decide to add another property to it, such as:

```
var person = { name: "Nicholas" };
person.age = 29;
```

Even though the `person` object is fully defined on the first line, the second line is able to add an additional property. The `Object.preventExtensions()` method changes this behavior so that new properties and methods cannot be added to the object. For example:

```
var person = { name: "Nicholas" };
Object.preventExtensions(person);

person.age = 29;
alert(person.age); //undefined
```

NonExtensibleObjectsExample01.htm

After the call to `Object.preventExtensions()`, the `person` object can no longer have new properties or methods added. In nonstrict mode, an attempt to add a new object member is silently ignored, so the result of `person.age` is `undefined`. In strict mode, attempting to add an object member that doesn't allow extension causes an error to be thrown.

Even though the object cannot have new members added, all of the existing members remain unaffected. You can still modify and delete already-existing members. It's also possible to determine that an object can't have extensions by using the `Object.isExtensible()` method:



Available for download on Wrox.com

```
var person = { name: "Nicholas" };
alert(Object.isExtensible(person)); //true

Object.preventExtensions(person);
alert(Object.isExtensible(person)); //false
```

NonExtensibleObjectsExample02.htm

Sealed Objects

The next level of protection for objects in ECMAScript 5 is a *sealed object*. Sealed objects aren't extensible and existing object members have their `[[Configurable]]` attribute set to `false`. This means properties and methods can't be deleted as data properties cannot be changed to accessor properties or vice versa using `Object.defineProperty()`. Property values can still be changed.

You can seal an object by using the `Object.seal()` method:



Available for download on
Wrox.com

```
var person = { name: "Nicholas" };
Object.seal(person);

person.age = 29;
alert(person.age);      //undefined

delete person.name;
alert(person.name);    //"Nicholas"
```

[SealedObjectsExample01.htm](#)

In this example, the attempt to add an `age` property is ignored. The attempt to delete the `name` property is also ignored, and so the value remains intact. This is the behavior for nonstrict mode. In strict mode, attempting to add or delete an object member throws an error.

You can determine if an object is sealed by using `Object.isSealed()`. Since a sealed object is also not extensible, sealed objects also return `false` for `Object.isExtensible()`:

```
var person = { name: "Nicholas" };
alert(Object.isExtensible(person));    //true
alert(Object.isSealed(person));        //false

Object.seal(person);
alert(Object.isExtensible(person));    //false
alert(Object.isSealed(person));        //true
```

[SealedObjectsExample02.htm](#)

Frozen Objects

The strictest type of tamper-proof object is a *frozen object*. Frozen objects aren't extensible and are sealed, and also data properties have their `[[Writable]]` attribute set to `false`. Accessor properties may still be written to but only if a `[[Set]]` function has been defined. ECMAScript 5 defines `Object.freeze()` to allow freezing of objects:

```
var person = { name: "Nicholas" };
Object.freeze(person);

person.age = 29;
alert(person.age);      //undefined

delete person.name;
alert(person.name);    //"Nicholas"

person.name = "Greg";
alert(person.name);    //"Nicholas"
```

[FrozenObjectsExample01.htm](#)

As with preventing extensions and sealing, attempts to perform disallowed operations on a frozen object are ignored in nonstrict mode and throw an error in strict mode.

There is also an `Object.isFrozen()` method to detect frozen objects. Since frozen objects are both sealed and extensible, they also return `false` for `Object.isExtensible()` and `true` for `Object.isSealed()`:



```
var person = { name: "Nicholas" };
alert(Object.isExtensible(person)); //true
alert(Object.isSealed(person)); //false
alert(Object.isFrozen(person)); //false

Object.freeze(person);
alert(Object.isExtensible(person)); //false
alert(Object.isSealed(person)); //true
alert(Object.isFrozen(person)); //true
```

FrozenObjectsExample02.htm

Frozen objects are especially useful for library authors. A very common problem with JavaScript libraries is when people accidentally (or intentionally) change the main library object. Freezing the main library object (or sealing) can help to prevent some of these errors.

ADVANCED TIMERS

Timers created using `setTimeout()` or `setInterval()` can be used to achieve interesting and useful functionality. Despite the common misconception that timers in JavaScript are actually threads, JavaScript runs in a single-threaded environment. Timers, then, simply schedule code execution to happen at some point in the future. The timing of execution is not guaranteed, because other code may control the JavaScript process at different times during the page life cycle. Code running when the page is downloaded, event handlers, and Ajax callbacks all must use the same thread for execution. It's the browser's job to sort out which code has priority at what point in time.

It helps to think of JavaScript as running on a timeline. When a page is loading, the first code to be executed is any code included using a `<script>` element. This often is simply function and variable declarations to be used later during the page life cycle, but sometimes it can contain initial data processing. After that point, the JavaScript process waits for more code to execute. When the process isn't busy, the next code to be triggered is executed immediately. For instance, an `onclick` event handler is executed immediately when a button is clicked, as long as the JavaScript process isn't executing any other code. The timeline for such a page might look like Figure 22-1.

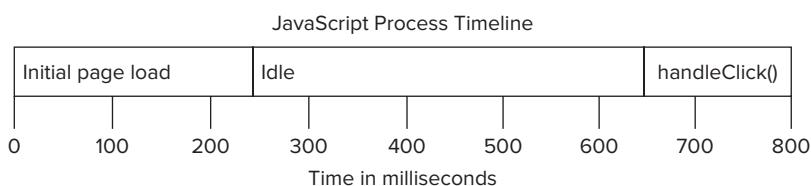


FIGURE 22-1

Alongside the main JavaScript execution process, there is a queue of code that should be executed the next time the process is idle. As the page goes through its life cycle, code is added to the queue in the order in which it should be executed. When a button is clicked, for example, its event handler code is added to the queue and executed at the next possible moment. When an Ajax response is received, the callback function code is added to the queue. No code is executed immediately in JavaScript; it is executed as soon as the process is idle.

Timers work with this queue by inserting code when a particular amount of time has passed. Note that adding code to the queue doesn't mean it's executed immediately; it simply means that it will be executed as soon as possible. Setting a timer for execution in 150 milliseconds doesn't mean that the code will be executed in 150 milliseconds; it means that the code will be added to the queue in 150 milliseconds. If nothing else is in the queue at that point in time, the timer code will be executed, giving the appearance that the code executed exactly when specified. At other times, the code may take significantly longer to execute.

Consider the following code:

```
var btn = document.getElementById("my-btn");
btn.onclick = function(){
    setTimeout(function() {
        document.getElementById("message").style.visibility = "visible";
    }, 250);

    //other code
};
```

Here, an event handler is set up for a button. The event handler sets a timer to be called in 250 milliseconds. When the button is clicked, the `onclick` event handler is first added to the queue. When it is executed, the timer is set, and 250 milliseconds later, the specified code is added to the queue for execution. In effect, the call to `setTimeout()` says that some code should be executed later.

The most important thing to remember about timers is that the specified interval indicates when the timer's code will be added to the queue, not when the code will actually be executed. If the `onclick` event handler in the previous example took 300 milliseconds to execute, then the timer's code would execute, at the earliest, 300 milliseconds after the timer was set. All code in the queue must wait until the JavaScript process is free before it can be executed, regardless of how it was added to the queue (see Figure 22-2).

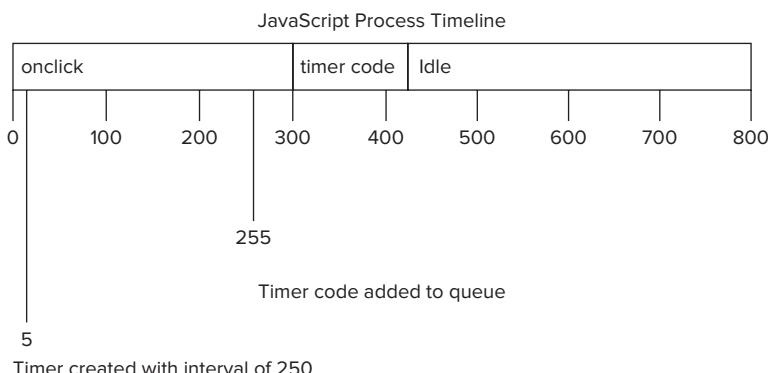


FIGURE 22-2

As you can see from Figure 22-2, even though the timer code was added at the 255-millisecond mark, it cannot be executed at that time because the `onclick` event handler is still running. The timer code's first opportunity to be executed is at the 300-millisecond mark, after the `onclick` event handler has finished.

Firefox's implementation of timers actually allows you to determine how far behind a timer has slipped. It does so by passing in the differential between the time that it was executed and the interval specified. Here is an example:

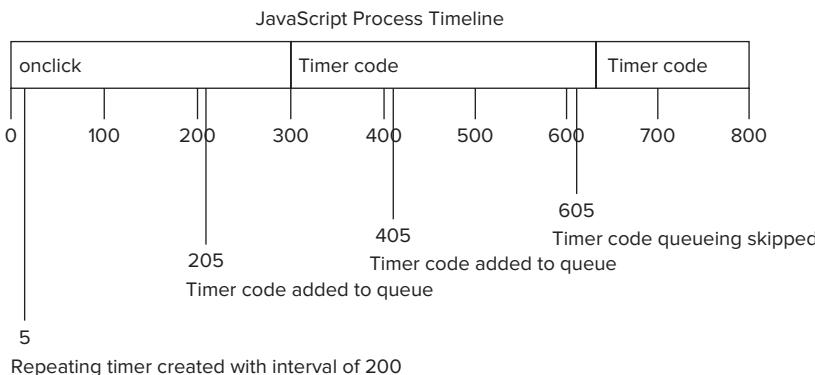
```
//works in Firefox only
setTimeout(function(diff){
    if (diff > 0) {
        //call is late
    } else if (diff < 0){
        //call is early
    } else {
        //call is on time
    }
}, 250);
```

When the execution of one set of code is complete, the JavaScript process yields for a short amount of time so that other processes on the page can be executed. Since the JavaScript process blocks other page processes, these small breaks are necessary to prevent the user interface from locking (which can still happen during long-running code). Setting a timer ensures that there will be at least one process break before the timer code is executed.

Repeating Timers

Timers created using `setInterval()` ensure regular injection of timer code into the queue. The problem with this approach is that the timer code may not finish execution before the code is added to the queue again. The result would be that the timer code is run multiple times in a row, with no amount of time between them. Fortunately, JavaScript engines are smart enough to avoid this issue. When using `setInterval()`, timer code is added to the queue only if there are no other instances of the timer code already in the queue. This ensures that the time between additions of the timer code to the queue is, at a minimum, the specified interval.

The downside to this regulation of repeating timers is twofold: (1) intervals may be skipped, and (2) intervals may be smaller than expected between multiple timer-code executions. Suppose you have a situation where an `onclick` event handler sets a repeating timer using `setInterval()` at any interval of 200 milliseconds. If the event handler takes a little over 300 milliseconds to complete, and the timer code takes about the same amount of time, you'll end up with both a skipped interval and timer code running back-to-back (see Figure 22-3).

**FIGURE 22-3**

The first timer in this example is added to the queue at 205 milliseconds but can't be executed until after the 300-millisecond mark. While the timer code is being executed, another copy is added to the queue at 405 milliseconds. At the next interval, 605 milliseconds, the first timer code is still being executed, and there is already one instance of the timer code in the queue. As a result, timer code is not added to the queue at that point. The timer code added at 405 milliseconds is then executed right after the timer code that was added at 5 milliseconds.

To avoid the two downfalls of repeating timers with `setInterval()`, you can use chained `setTimeout()` calls in the following pattern:

```
setTimeout(function() {
    //processing
    setTimeout(arguments.callee, interval);
}, interval);
```

This pattern chains calls to `setTimeout()`, creating a new timer each time the function is executed. The second call to `setTimeout()` uses `arguments.callee` to get a reference to the currently executing function and set another timer for it. The advantage is that new timer code isn't inserted into the queue until the previous timer code has been executed, ensuring that there won't be any dropped intervals. Furthermore, you are guaranteed that the next time the timer code is executed, it will be in at least the interval specified, avoiding back-to-back runs. This pattern is used most often for repeating timers, as in this example:



Available for
download on
Wrox.com

```
setTimeout(function() {
    var div = document.getElementById("myDiv"),
        left = parseInt(div.style.left) + 5;
    div.style.left = left + "px";

    if (left < 200) {
        setTimeout(arguments.callee, 50);
    }
});
```

```
    }  
  
}, 50);
```

[RepeatingTimersExample.htm](#)

This code moves a <div> element to the right every time the timer code executes, stopping when the left coordinate is at 200 pixels. It's quite common to use this pattern for JavaScript animation.



Each browser window, tab, or frame has its own code execution queue. This means that the timing of cross-frame or cross-window JavaScript calls may result in race conditions when code is executed synchronously. Whenever this type of communication is necessary, it's a good idea to create a timer on the receiving frame or window to execute the code.

Yielding Processes

JavaScript running in a browser has a finite amount of resources allocated to it. Unlike desktop applications, which often have free rein over the amount of memory and processor time they can command, JavaScript is severely restricted to ensure that malicious web programmers can't bring down a user's computer. One of these restrictions is the *long-running script* limit, which prevents code from running if it takes longer than a certain amount of time or a certain number of statements. If you reach that limit, the user is presented with a browser error dialog indicating that a script is taking too long to execute and asking whether the user would like to allow it to continue processing or stop. It's the goal of all JavaScript developers to ensure that the user never sees this confusing message from the browser. Timers are one way to work around this limitation.

Long-running script problems typically result from one of two issues: long, deeply nested function calls or loops that are doing a lot of processing. Of these two, the latter is an easier problem to solve. Long-running loops typically follow this pattern:

```
for (var i=0, len=data.length; i < len; i++){  
    process(data[i]);  
}
```

The problem with this pattern is that the number of items to process is unknown until runtime. If `process()` takes 100 milliseconds to complete, an array of two items may not be cause for worry, but an array of ten results in the script running for a second to complete. The amount of time it takes to completely execute this loop is directly related to the number of items in the array. And since JavaScript execution is a blocking operation, the longer a script takes to run, the longer users are left unable to interact with the page.

Before unrolling the loop, you need to answer these two important questions:

- 1.** Does the processing have to be done synchronously? If the processing of this data is blocking something else from finishing, then you may not want to touch it. However, if you can answer a definitive “no” to this question, then it’s a good candidate for deferring some processing until later.
- 2.** Does the data have to be processed sequentially? Oftentimes, an array of values is just a convenient way to group and iterate over items regardless of the order. If the order of the items has no significance, then it’s likely that you can postpone some processing until later.

When you find a loop is taking a significant amount of time to complete, and you can answer “no” to either of the previous two questions, you can split the loop using timers. This is a technique called *array chunking*, whereby processing of the array happens in small chunks, most often one at a time. The basic idea is to create a queue of items to process, use timers to pull the next item to process, process it, and then set another timer. The basic pattern looks like this:

```
setTimeout(function(){

    //get next item and process it
    var item = array.shift();
    process(item);

    //if there's more items, set another timeout
    if(array.length > 0){
        setTimeout(arguments.callee, 100);
    }
}, 100);
```

In the array chunking pattern, the `array` variable is essentially a “to do” list of items to process. Using the `shift()` method, you retrieve the next item in the queue to process and pass it in to a function. If there are still items in the queue, then another timer is set, calling the same anonymous function via `arguments.callee`. You can accomplish array chunking easily, using the following function:



Available for
download on
Wrox.com

```
function chunk(array, process, context){
    setTimeout(function(){
        var item = array.shift();
        process.call(context, item);

        if (array.length > 0){
            setTimeout(arguments.callee, 100);
        }
    }, 100);
}
```

[ArrayChunkingExample.htm](#)

The `chunk()` method accepts three arguments: the array of items to process, a function to use to process the items, and an optional context in which to run the function. Inside the function is a duplication of the basic pattern described previously, with the `process()` function being called

via `call()` so that a proper context can be set if necessary. The interval of the timers is set to 100 milliseconds, which gives the JavaScript process time to go idle between item processing events. This interval can be changed based on your needs, although 100 milliseconds works well in most cases. The function can be used as follows:



```
var data = [12,123,1234,453,436,23,23,5,4123,45,346,5634,2234,345,342];

function printValue(item){
    var div = document.getElementById("myDiv");
    div.innerHTML += item + "<br>";
}

chunk(data, printValue);
```

[ArrayChunkingExample.htm](#)

This example outputs each value in the `data` array to a `<div>` element by using the `printValue()` function. Since the function exists in the global scope, there's no need to pass in a `context` object to `chunk()`.

Something to be aware of is that the array passed into `chunk()` is used as a queue, so the items in the array change as the data is processed. If you want to keep the original array intact, you should pass a clone of the array into `chunk()`, such as in this example:

```
chunk(data.concat(), printValue);
```

When the `concat()` method is called on an array without any arguments, it returns an array with the same items as the original. In this way, you can be assured that the original array is not changed by the function.

The importance of array chunking is that it splits the processing of multiple items into separate code on the execution queue. Other browser processes are given a chance to run after each item is processed, and you'll avoid long-running script errors.



Whenever you have a function that takes over 50 milliseconds to complete, it's best to see if you can split up the job into a number of smaller ones that can be used with timers.

Function Throttling

Some calculations and processes are more expensive in the browser than others. For instance, DOM manipulations require more memory and CPU time than non-DOM interactions. Attempting to perform too many DOM-related operations in sequence can cause the browser to hang, and sometimes crash. This tends to happen frequently in Internet Explorer when using an `onresize` event handler, which fires repeatedly as the browser is being resized. Attempting DOM manipulations inside the `onresize` event handler can make the browser crash because of the frequency of the changes being calculated. To get around this problem, you can *throttle* the function call by using timers.

The basic idea behind function throttling is that some code should not be executed repeatedly without a break. The first time the function is called, a timer is created that will run the code after a specified interval. When the function is called a second time, it clears the previous timer and sets another. If the previous timer has already executed, then it is of no consequence. However, if the previous timer hasn't executed, it is essentially replaced by a newer timer. The goal is to execute the function only after the requests to execute it have subsided for some amount of time. The following is a basic representation of this pattern:

```
var processor = {
    timeoutId: null,

    //method that actually performs the processing
    performProcessing: function(){
        //actual processing code
    },

    //method that is called to initiate processing
    process: function(){
        clearTimeout(this.timeoutId);

        var that = this;
        this.timeoutId = setTimeout(function(){
            that.performProcessing();
        }, 100);
    }
};

//try to start processing
processor.process();
```

In this code, an object called `processor` is created. There are two methods on this object: `process()` and `performProcessing()`. The former is the one that should be called to initiate any processing, and the latter actually performs the processing that should be done. When `process()` is called, the first step is to clear the stored `timeoutId` to prevent any previous calls from being executed. Then, a new timer is created to call `performProcessing()`. Since the context of the function used in `setTimeout()` is always `window`, it's necessary to store a reference to `this` so that it can be used later.

The interval is set to 100 milliseconds, which means that `performProcessing()` will not be called until at least 100 milliseconds after the last call to `process()`. So if `process()` is called 20 times within 100 milliseconds, `performProcessing()` will still be called only once.

This pattern can be simplified by using a `throttle()` function that automatically sets up the timer setting/clearing functionality, as in the following example:



Available for
download on
Wrox.com

```
function throttle(method, context) {
    clearTimeout(method.tId);
    method.tId= setTimeout(function(){
        method.call(context);
    }, 100);
}
```

[ThrottlingExample.htm](#)

The `throttle()` function accepts two arguments: the function to execute and the scope in which to execute it. The function first clears any timer that was set previously. The timer ID is stored on the `tId` property of the function, which may not exist the first time the method is passed into `throttle()`. Next, a new timer is created, and its ID is stored in the method's `tId` property. If this is the first time that `throttle()` is being called with this method, then the code creates the property. The timer code uses `call()` to ensure that the method is executed in the appropriate context. If the second argument isn't supplied, then the method is executed in the global scope.

As mentioned previously, throttling is most often used during the `resize` event. If you are changing the layout of the page based on this event, it is best to throttle the processing to ensure that the browser doesn't do too many calculations in a short period of time. For example, consider having a `<div>` element that should have its height changed so that it's always equal to its width. The JavaScript to effect this change may look something like this:

```
window.onresize = function(){
    var div = document.getElementById("myDiv");
    div.style.height = div.offsetWidth + "px";
};
```

This very simple example shows a couple of things that may slow down the browser. First, the `offsetWidth` property is being calculated, which may be a complex calculation when there are enough CSS styles applied to the element and the rest of the page. Second, setting the height of an element requires a reflow of the page to take these changes into account. Once again, this can require multiple calculations if the page has many elements and a moderate amount of CSS applied. The `throttle()` function can help, as shown in this example:



```
function resizeDiv(){
    var div = document.getElementById("myDiv");
    div.style.height = div.offsetWidth + "px";
}

window.onresize = function(){
    throttle(resizeDiv);
};
```

[ThrottlingExample.htm](#)

Here, the resizing functionality has been moved into a separate function called `resizeDiv()`. The `onresize` event handler then calls `throttle()` and passes in the `resizeDiv()` function, instead of calling `resizeDiv()` directly. In many cases, there is no perceivable difference to the user, even though the calculation savings for the browser can be quite large.

Throttling should be used whenever there is code that should be executed only periodically, but you cannot control the rate at which the execution is requested. The `throttle()` function presented here uses an interval of 100 milliseconds, but that can be changed, depending on your needs.

CUSTOM EVENTS

Earlier in this book, you learned that events are the primary way in which JavaScript interacts with the browser. Events are a type of design pattern called an *observer*, which is a technique for creating loosely coupled code. The idea is that objects can publish events indicating when an interesting moment in the object's life cycle occurs. Other objects can then *observe* that object, waiting for these interesting moments to occur and responding by running code.

The observer pattern is made up of two types of objects: a *subject* and an *observer*. The subject is responsible for publishing events, and the observer simply observes the subject by subscribing to these events. A key concept for this pattern is that the subject doesn't know anything about the observer, meaning that it can exist and function appropriately even if the observer isn't present. The observer, on the other hand, knows about the subject and registers callbacks (event handlers) for the subject's events. When you're dealing with the DOM, a DOM element is the subject and your event-handling code is the observer.

Events are a very common way to interact with the DOM, but they can also be used in non-DOM code through implementing custom events. The idea behind custom events is to create an object that manages events, allowing others to listen to those events. A basic type that implements this functionality can be defined as follows:



Available for
download on
Wrox.com

```
function EventTarget(){
    this.handlers = {};
}

EventTarget.prototype = {
    constructor: EventTarget,

    addHandler: function(type, handler){
        if (typeof this.handlers[type] == "undefined"){
            this.handlers[type] = [];
        }

        this.handlers[type].push(handler);
    },

    fire: function(event){
        if (!event.target){
            event.target = this;
        }
        if (this.handlers[event.type] instanceof Array){
            var handlers = this.handlers[event.type];
            for (var i=0, len=handlers.length; i < len; i++){
                handlers[i](event);
            }
        }
    },

    removeHandler: function(type, handler){
        if (this.handlers[type] instanceof Array){
            var handlers = this.handlers[type];
            for (var i=0, len=handlers.length; i < len; i++){

```

```

        if (handlers[i] === handler){
            break;
        }
    }

    handlers.splice(i, 1);
}
}
};


```

[EventTarget.js](#)

The `EventTarget` type has a single property, `handlers`, which is used to store the event handlers. There are also three methods: `addHandler()`, which registers an event handler for a given type of event; `fire()`, which fires an event; and `removeHandler()`, which unregisters an event handler for an event type.

The `addHandler()` method accepts two arguments: the event type and a function used to handle the event. When this method is called, a check is made to see if an array for the event type already exists on the `handlers` property. If not, then one is created. The handler is then added to the end of the array, using `push()`.

When an event must be fired, the `fire()` method is called. This method accepts a single argument, which is an object containing at least a `type` property. The `fire()` method begins by setting a `target` property on the `event` object, if one isn't already specified. Then it simply looks for an array of handlers for the event type and calls each function, passing in the `event` object. Because these are custom events, it's up to you to determine what the additional information on the `event` object should be.

The `removeHandler()` method is a companion to `addHandler()` and accepts the same arguments: the type of event and the event handler. This method searches through the event handler array to find the location of the handler to remove. When it's found, the `break` operator is used to exit the `for` loop. The `splice()` method is then used to remove just that item from the array.

Custom events using the `EventTarget` type can then be used as follows:



```

function handleMessage(event){
    alert("Message received: " + event.message);
}

//create a new object
var target = new EventTarget();

//add an event handler
target.addHandler("message", handleMessage);

//fire the event
target.fire({ type: "message", message: "Hello world!"});

//remove the handler

```

```
target.removeHandler("message", handleMessage);

//try again - there should be no handler
target.fire({ type: "message", message: "Hello world!"});
```

[EventTargetExample01.htm](#)

In this code, the `handleMessage()` function is defined to handle a `message` event. It accepts the `event` object and outputs the `message` property. The `target` object's `addHandler()` method is called, passing in `"message"` and the `handleMessage()` function. On the next line, `fire()` is called with an object literal containing two properties: `type` and `message`. This calls the event handlers for the `message` event so an alert will be displayed (from `handleMessage()`). The event handler is then removed so that when the event is fired again, no alert will be displayed.

Because this functionality is encapsulated in a custom type, other objects can inherit this behavior by inheriting from `EventTarget`, as in this example:



```
function Person(name, age) {
    EventTarget.call(this);
    this.name = name;
    this.age = age;
}

inheritPrototype(Person, EventTarget);

Person.prototype.say = function(message) {
    this.fire({type: "message", message: message});
};
```

[EventTargetExample02.htm](#)

The `Person` type uses parasitic combination inheritance (see Chapter 6) to inherit from `EventTarget`. Whenever the `say()` method is called, an event is fired with the details of a message. It's common for the `fire()` method to be called during other methods of a type and quite uncommon for it to be called publicly. This code can then be used as follows:

```
function handleMessage(event) {
    alert(event.target.name + " says: " + event.message);
}

//create new person
var person = new Person("Nicholas", 29);

//add an event handler
person.addHandler("message", handleMessage);

//call a method on the object, which fires the message event
person.say("Hi there.");
```

[EventTargetExample02.htm](#)

The `handleMessage()` function in this example displays an alert with the person's name (retrieved via `event.target.name`) and the message text. When the `say()` method is called with a message, the `message` event is fired. That, in turn, calls the `handleMessage()` function and displays the alert.

Custom events are useful when there are multiple parts of your code that interact with each other at particular moments in time. If each object has references to all the others, the code becomes tightly coupled, and maintenance becomes difficult, because a change to one object affects others. Using custom events helps to decouple related objects, keeping functionality insulated. In many cases, the code that fires the events and the code that listens for the events are completely separate.

DRAG AND DROP

One of the most popular user interface patterns on computers is drag and drop. The idea is simple: click and hold a mouse button over an item, move the mouse to another area, and release the mouse button to “drop” the item there. The popularity of the drag-and-drop interface extends to the Web, where it has become a popular alternative to more traditional configuration interfaces.

The basic idea for drag and drop is simple: create an absolutely positioned element that can be moved with the mouse. This technique has its origins in a classic web trick called the *cursor trail*. A cursor trail was an image or multiple images that shadowed mouse pointer movements on the page. The basic code for a single-item cursor trail involves setting an `onmousemove` event handler on the document that always moves a given element to the cursor position, as in this example:



```
EventUtil.addHandler(document, "mousemove", function(event) {
    var myDiv = document.getElementById("myDiv");
    myDiv.style.left = event.clientX + "px";
    myDiv.style.top = event.clientY + "px";
});
```

[DragAndDropExample01.htm](#)

In this example, an element's left and top coordinates are set equal to the `event` object's `clientX` and `clientY` properties, which places the element at the cursor's position in the viewport. The effect is an element that follows the cursor around the page whenever it's moved. To implement drag and drop, you need only implement this functionality at the correct point in time (when the mouse button is pushed down) and remove it later (when the mouse button is released). A very simple drag-and-drop interface can be implemented using the following code:

```
var DragDrop = function() {
    var dragging = null;
    function handleEvent(event) {
        //get event and target
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        //determine the type of event
```

```

switch(event.type){
    case "mousedown":
        if (target.className.indexOf("draggable") > -1) {
            dragging = target;
        }
        break;

    case "mousemove":
        if (dragging !== null){

            //assign location
            dragging.style.left = event.clientX + "px";
            dragging.style.top = event.clientY + "px";
        }
        break;

    case "mouseup":
        dragging = null;
        break;
}
};

//public interface
return {
    enable: function(){
        EventUtil.addHandler(document, "mousedown", handleEvent);
        EventUtil.addHandler(document, "mousemove", handleEvent);
        EventUtil.addHandler(document, "mouseup", handleEvent);
    },
    disable: function(){
        EventUtil.removeHandler(document, "mousedown", handleEvent);
        EventUtil.removeHandler(document, "mousemove", handleEvent);
        EventUtil.removeHandler(document, "mouseup", handleEvent);
    }
}
}();

```

[DragAndDropExample02.htm](#)

The `DragDrop` object encapsulates all of the basic drag-and-drop functionality. It is a singleton object that uses the module pattern to hide some of its implementation details. The `dragging` variable starts out as `null` and will be filled with the element that is being dragged, so when this variable isn't `null`, you know that something is being dragged. The `handleEvent()` function handles all three mouse events for the drag-and-drop functionality. It starts by retrieving references to the `event` object and the `event.target`. After that, a `switch` statement determines which event type was fired. When a `mousedown` event occurs, the `class` of the `target` is checked to see if it contains a class of `"draggable"` and if so, the `target` is assigned to `dragging`. This technique allows draggable elements to easily be indicated through markup instead of JavaScript.

The `mousemove` case for `handleEvent()` is the same as the previous code, with the exception that a check is made to see if `dragging` is `null`. When it's not `null`, `dragging` is known to be the element

that's being dragged, so it is repositioned appropriately. The `mouseup` case simply resets dragging to `null`, which effectively negates the `mousemove` event.

There are two public methods on `DragDrop`: `enable()` and `disable()`, which simply attach and detach all event handlers, respectively. These methods provide an additional measure of control over the drag-and-drop functionality.

To use the `DragDrop` object, just include it on a page and call `enable()`. Drag and drop will automatically be enabled for all elements with a class containing "draggable", as in this example:

```
<div class="draggable" style="position:absolute; background:red"></div>
```

Note that for drag and drop to work with an element, it must be absolutely positioned.

Fixing Drag Functionality

When you try out this example, you'll notice that the upper-left corner of the element always lines up with the cursor. The result is a little jarring to users, because the element seems to jump when the mouse begins to move. Ideally, the action should look as if the element has been "picked up" by the cursor, meaning that the point where the user clicked should be where the cursor remains while the element is being dragged (see Figure 22-4).

Some additional calculations are necessary to achieve the desired effect. To do so, you need to calculate the difference between the upper-left corner of the element and the cursor location. That difference needs to be determined when the `mousedown` event occurs, and carried through until the `mouseup` event occurs. By comparing the `clientX` and `clientY` properties of `event` to the `offsetLeft` and `offsetTop` properties of the element, you can figure out how much more space is needed both horizontally and vertically (see Figure 22-5).

In order to store the differences in the x and y positions, you need a couple more variables. These variables, `diffX` and `diffY`, need to be used in the `onmousemove` event handler to properly position the element, as shown in the following example:



Available for download on
Wrox.com

```
var DragDrop = function() {
    var dragging = null,
        diffX = 0,
        diffY = 0;

    function handleEvent(event) {
        //get event and target
```

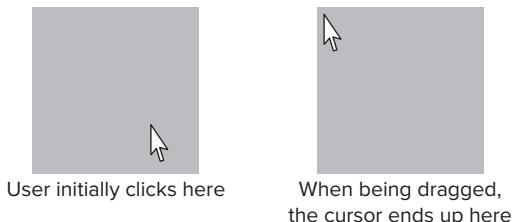


FIGURE 22-4

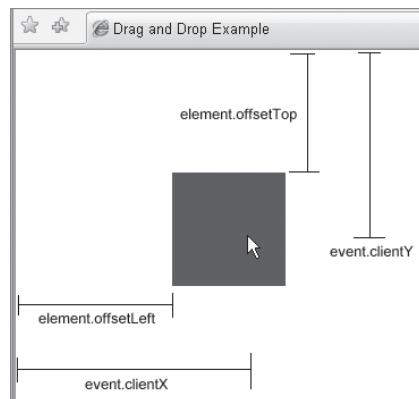


FIGURE 22-5

```

event = EventUtil.getEvent(event);
var target = EventUtil.getTarget(event);

//determine the type of event
switch(event.type){
    case "mousedown":
        if (target.className.indexOf("draggable") > -1){
            dragging = target;
            diffX = event.clientX - target.offsetLeft;
            diffY = event.clientY - target.offsetTop;
        }
        break;

    case "mousemove":
        if (dragging !== null){

            //assign location
            dragging.style.left = (event.clientX - diffX) + "px";
            dragging.style.top = (event.clientY - diffY) + "px";

        }
        break;

    case "mouseup":
        dragging = null;
        break;
}
};

//public interface
return {
    enable: function(){
        EventUtil.addHandler(document, "mousedown", handleEvent);
        EventUtil.addHandler(document, "mousemove", handleEvent);
        EventUtil.addHandler(document, "mouseup", handleEvent);
    },

    disable: function(){
        EventUtil.removeHandler(document, "mousedown", handleEvent);
        EventUtil.removeHandler(document, "mousemove", handleEvent);
        EventUtil.removeHandler(document, "mouseup", handleEvent);
    }
}
}();

```

[DragAndDropExample03.htm](#)

The `diffX` and `diffY` variables are private, because they are needed only by the `handleEvent()` function. When a `mousedown` event occurs, they are calculated by subtracting `offsetLeft` from the target's `clientX` and `offsetTop` from the target's `clientY`. These give you the amount that needs to be subtracted from each dimension when the `mousemove` event is fired. The result is a smoother dragging experience that behaves much more in the way that the user expects.

Adding Custom Events

The drag-and-drop functionality can't really be used in an application unless you know when the dragging occurs. To this point, the code provides no way to indicate that a drag has been started, is in progress, or has ended. Custom events can be used to indicate when each of these occurs, allowing other parts of the application to interact with the drag-and-drop functionality.

Since the `DragDrop` object is a singleton using the module pattern, some changes are necessary to use the `EventTarget` type. First, a new `EventTarget` object is created, then the `enable()` and `disable()` methods are added, and finally the object is returned. Consider the following:



Available for download on
Wrox.com

```
var DragDrop = function(){

    var dragdrop = new EventTarget(),
        dragging = null,
        diffX = 0,
        diffY = 0;

    function handleEvent(event){

        //get event and target
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        //determine the type of event
        switch(event.type){
            case "mousedown":
                if (target.className.indexOf("draggable") > -1){
                    dragging = target;
                    diffX = event.clientX - target.offsetLeft;
                    diffY = event.clientY - target.offsetTop;
                    dragdrop.fire({type:"dragstart", target: dragging,
                                  x: event.clientX, y: event.clientY});
                }
                break;

            case "mousemove":
                if (dragging !== null){

                    //assign location
                    dragging.style.left = (event.clientX - diffX) + "px";
                    dragging.style.top = (event.clientY - diffY) + "px";

                    //fire custom event
                    dragdrop.fire({type:"drag", target: dragging,
                                  x: event.clientX, y: event.clientY});
                }
                break;

            case "mouseup":
                dragdrop.fire({type:"dragend", target: dragging,
                              x: event.clientX, y: event.clientY});
                dragging = null;
                break;
        }
    }
}
```

```

        }
    };

    //public interface
    dragdrop.enable = function(){
        EventUtil.addHandler(document, "mousedown", handleEvent);
        EventUtil.addHandler(document, "mousemove", handleEvent);
        EventUtil.addHandler(document, "mouseup", handleEvent);
    };

    dragdrop.disable = function(){
        EventUtil.removeHandler(document, "mousedown", handleEvent);
        EventUtil.removeHandler(document, "mousemove", handleEvent);
        EventUtil.removeHandler(document, "mouseup", handleEvent);
    };

    return dragdrop;
}();

```

[DragAndDropExample04.htm](#)

This code defines three events: `dragstart`, `drag`, and `dragend`. Each of these events sets the dragged element as the `target` and provides `x` and `y` properties to indicate its current position. These are fired on the `dragdrop` object, which later is augmented with the `enable()` and `disable()` methods before being returned. This slight change in the module pattern allows the `DragDrop` object to support events such as the following:



Available for
download on
Wrox.com

```

DragDrop.addHandler("dragstart", function(event){
    var status = document.getElementById("status");
    status.innerHTML = "Started dragging " + event.target.id;
});

DragDrop.addHandler("drag", function(event){
    var status = document.getElementById("status");
    status.innerHTML += "<br>Dragged " + event.target.id + " to (" + event.x +
                      ", " + event.y + ")";
});

DragDrop.addHandler("dragend", function(event){
    var status = document.getElementById("status");
    status.innerHTML += "<br>Dropped " + event.target.id + " at (" + event.x +
                      ", " + event.y + ")";
});

```

[DragAndDropExample04.htm](#)

Here, event handlers are added for each event of the `DragDrop` object. An element is used to display the current state and location of the dragged element. Once the element is dropped, you have a listing of all the intermediate steps it took since it was initially dragged.

Adding custom events to `DragDrop` makes it a more robust object that can be used to manage complex drag-and-drop functionality in a web application.

SUMMARY

Functions in JavaScript are quite powerful, because they are first-class objects. Using closures and function context switching, you can use functions in a number of powerful ways. For example:

- It's possible to create scope-safe constructors, ensuring that a constructor called without the new operator will not change the wrong context object.
- You can use lazy loading functions by delaying any code forking until the first time that the function is called.
- Function binding allows you to create functions that are always run in a specific context, and function currying allows you to create functions that have some of their arguments already filled in.
- Combining binding and currying gives you a way to execute any function, in any context, and with any arguments.

ECMAScript 5 allows you to create tamper-proof objects in a number of different ways:

- Nonextensible objects don't allow new properties or methods to be added to the object.
- Sealed objects are nonextensible and also don't allow existing properties and methods to be deleted.
- Frozen objects are sealed and also don't allow the overwriting of object members.

Timers can be created in JavaScript using `setTimeout()` or `setInterval()` as follows:

- Timer code is placed into a holding area until the interval has been reached, at which point the code is added to the JavaScript process queue to be executed the next time the JavaScript process is idle.
- Every time a piece of code executes completely, there is a brief amount of idle time to allow other browser processes to complete.
- This behavior means that timers can be used to split up long-running scripts into smaller chunks that can be executed at a later time. Doing so helps the web application to be more responsive to user interaction.

The observer pattern is used quite often in JavaScript in the form of events. Although events are used frequently with the DOM, they can also be used in your own code by implementing custom events. Using custom events helps to decouple different parts of code from one another, allowing easier maintenance and reducing the chances of introducing an error by changing what seems to be isolated code.

Drag and drop is a popular user-interface paradigm for both desktop and web applications, allowing users to easily rearrange or configure things in an intuitive way. This type of functionality can be created in JavaScript using mouse events and some simple calculations. Combining drag-and-drop behavior with custom events creates a reusable framework that can be applied in many different ways.

23

Offline Applications and Client-Side Storage

WHAT'S IN THIS CHAPTER?

- Setting up offline detection
- Using the offline cache
- Storing data in the browser

One of HTML5's focus areas is enabling offline web applications. An *offline* web application still works even when there is no Internet connection available to the device. This focus is based on the desire of web application developers to better compete with traditional client applications that may be used so long as the device has power.

For web applications, creating an offline experience requires several steps. The first step is to ensure that the application knows whether an Internet connection is available or not in order to perform the correct operation. Then, the application still needs to have access to a subset of resources (images, JavaScript, CSS, and so on) in order to continue working properly. The last piece is a local data storage area that can be written to and read from regardless of Internet availability. HTML5 and other associated JavaScript APIs make offline applications a reality.

OFFLINE DETECTION

Since the first step for offline applications is to know whether or not the device is offline, HTML5 defines a `navigator.onLine` property that is `true` when an Internet connection is available or `false` when it's not. The idea is that the browser should be aware if the network is available or not and return an appropriate indicator. In practice, `navigator.onLine` is a bit quirky across browsers:

- Internet Explorer 6+ and Safari 5+ correctly detect that the network connection has been lost and switch `navigator.onLine` to `false`.

- Firefox 3+ and Opera 10.6+ support `navigator.onLine`, but you must manually set the browser to work in offline mode via the File ⇔ Work Offline menu item.
- Chrome through version 11 permanently has `navigator.onLine` set to true. There is an open bug to fix this.

Given these compatibility issues, `navigator.onLine` cannot be used as the sole determinant for network connectivity. Even so, it is useful in case errors do occur during requests. You can check the status as follows:



```
if (navigator.onLine) {
    //work as usual
} else {
    //perform offline behavior
}
```

[OnLineExample01.htm](#)

Along with `navigator.onLine`, HTML5 defines two events to better track when the network is available or not: `online` and `offline`. Each event is fired as the network status changes from online to offline or vice versa, respectively. These events fire on the `window` object:

```
EventUtil.addHandler(window, "online", function() {
    alert("Online");
});
EventUtil.addHandler(window, "offline", function() {
    alert("Offline");
});
```

[OnlineEventsExample01.htm](#)

For determining if an application is offline, it's best to start by looking at `navigator.onLine` to get the initial state when the page is loaded. After that, it's best to use the events to determine when network connectivity changes. The `navigator.onLine` property also changes as the events fire, but you would need to manually poll for changes to this property to detect a network change.

Offline detection is supported in Internet Explorer 6+ (`navigator.onLine` only), Firefox 3, Safari 4, Opera 10.6, Chrome, Safari for iOS 3.2, and WebKit for Android.

APPLICATION CACHE

The HTML5 *application cache*, or *appcache* for short, is designed specifically for use with offline web applications. The appcache is a cache area separate from the normal browser cache. You specify what should be stored in the page's appcache by providing a *manifest file* listing the resources to download and cache. Here's a simple manifest file:

```
CACHE MANIFEST
#Comment

file.js
file.css
```

In its simplest form, a manifest file lists out the resources to be downloaded so that they are available offline.



There are a lot of options for setting up this file, and these are beyond the scope of this book. Please see <http://html5doctor.com/go-offline-with-application-cache/> for a full description of the options.

The manifest file is associated with a page by specifying its path in the `manifest` attribute of `<html>`, for example:

```
<html manifest="/offline.manifest">
```

This code indicates that `/offline.manifest` contains the manifest file. The file must be served with a content type of `text/cache-manifest` to be used.

While the appcache is primarily a way for designated resources to be cached for offline use, it does have a JavaScript API that allows you to determine what the appcache is doing. The primary object for this is `applicationCache`. This object has one property, `status`, which indicates the current status of the appcache as one of the following constants:

- 0 for uncached, meaning that there is no appcache associated with the page.
- 1 for idle, meaning the appcache is not being updated.
- 2 for checking, meaning the appcache manifest file is being downloaded and checked for updates.
- 3 for downloading, meaning the appcache is downloading resources specified in the manifest file.
- 4 for update ready, meaning that the appcache was updated with new resources and all resources are downloaded and may be put into use via `swapCache()`.
- 5 for obsolete, meaning the appcache manifest file is no longer available and so the appcache is no longer valid for the page.

There are also a large number of events associated with the appcache to indicate when its status has changed. The events are:

- `checking` — Fires when the browser begins looking for an update to the appcache.
- `error` — Fires if there's an error at any point in the checking or downloading sequence.
- `noupdate` — Fires after checking if the appcache manifest hasn't changed.
- `downloading` — Fires when the appcache resources begin downloading.
- `progress` — Fires repeatedly as files are being downloaded into the appcache.
- `updateready` — Fires when a new version of the page's appcache has been downloaded and is ready for use with `swapCache()`.
- `cached` — Fires when the appcache is complete and ready for use.

These events fire generally in this order when the page is loaded. You can also trigger the appcache to go through this sequence of checking for updates again by calling `update()`:

```
applicationCache.update();
```

Once `update()` is called, the appcache goes to check if the manifest file is updated (fires `checking`) and then proceeds as if the page had just been loaded. If the `cached` event fires, it means the new appcache contents are ready for use and no further action is necessary. If the `updateready` event fires, that means a new version of the appcache is available and you need to enable it using `swapCache()`:

```
EventUtil.addHandler(applicationCache, "updateready", function(){
    applicationCache.swapCache();
});
```

The HTML5 appcache is supported in Firefox 3+, Safari 4+, Opera 10.6, Chrome, Safari for iOS 3.2+, and WebKit for Android. Firefox through version 4 throws an error when `swapCache()` is called.

DATA STORAGE

Along with the emergence of web applications came a call for the ability to store user information directly on the client. The idea is logical: information pertaining to a specific user should live on that user's machine. Whether that is login information, preferences, or other data, web application providers found themselves searching for ways to store data on the client. The first solution to this problem came in the form of cookies, a creation of the old Netscape Communications Corporation and described in a specification titled *Persistent Client State: HTTP Cookies* (still available at http://curl.haxx.se/rfc/cookie_spec.html). Today, cookies are just one option available for storing data on the client.

Cookies

HTTP cookies, commonly just called *cookies*, were originally intended to store session information on the client. The specification called for the server to send a `Set-Cookie` HTTP header containing session information as part of any response to an HTTP request. For instance, the headers of a server response may look like this:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value
Other-header: other-header-value
```

This HTTP response sets a cookie with the name of "name" and a value of "value". Both the name and the value are URL-encoded when sent. Browsers store such session information and send it back to the server via the `Cookie` HTTP header for every request after that point, such as the following:

```
GET /index.html HTTP/1.1
Cookie: name=value
Other-header: other-header-value
```

This extra information being sent back to the server can be used to uniquely identify the client from which the request was sent.

Restrictions

Cookies are, by nature, tied to a specific domain. When a cookie is set, it is sent along with requests to the same domain from which it was created. This restriction ensures that information stored in cookies is available only to approved recipients and cannot be accessed by other domains.

Since cookies are stored on the client computer, restrictions have been put in place to ensure that cookies can't be used maliciously and that they won't take up too much disk space. The total number of cookies per domain is limited, although it varies from browser to browser. For example:

- Internet Explorer 6 and lower enforced a limit of 20 cookies per domain.
- Internet Explorer 7 and later have a limit of 50 cookies per domain. Internet Explorer 7 initially shipped with support for a maximum of 20 cookies per domain, but that was later updated with a patch from Microsoft.
- Firefox limits cookies to 50 per domain.
- Opera limits cookies to 30 per domain.
- Safari and Chrome have no hard limit on the number of cookies per domain.

When cookies are set above the per-domain limit, the browser starts to eliminate previously set cookies. Internet Explorer and Opera begin by removing the least recently used (LRU) cookie to allow space for the newly set cookie. Firefox seemingly randomly decides which cookies to eliminate, so it's very important to mind the cookie limit to avoid unintended consequences.

There are also limitations as to the size of cookies in browsers. Most browsers have a byte-count limit of around 4096 bytes, give or take a byte. For best cross-browser compatibility, it's best to keep the total cookie size to 4095 bytes or less. The size limit applies to all cookies for a domain, not per cookie.

If you attempt to create a cookie that exceeds the maximum cookie size, the cookie is silently dropped. Note that one character typically takes one byte, unless you're using multibyte characters.

Cookie Parts

Cookies are made up of the following pieces of information stored by the browser:

- **Name** — A unique name to identify the cookie. Cookie names are case-insensitive, so `myCookie` and `MyCookie` are considered to be the same. In practice, however, it's always best to treat the cookie names as case-sensitive because some server software may treat them as such. The cookie name must be URL-encoded.
- **Value** — The string value stored in the cookie. This value must also be URL-encoded.
- **Domain** — The domain for which the cookie is valid. All requests sent from a resource at this domain will include the cookie information. This value can include a subdomain (such as `www.wrox.com`) or exclude it (such as `.wrox.com`, which is valid for all subdomains of `wrox.com`). If not explicitly set, the domain is assumed to be the one from which the cookie was set.

- **Path** — The path within the specified domain for which the cookie should be sent to the server. For example, you can specify that the cookie be accessible only from `http://www.wrox.com/books/` so pages at `http://www.wrox.com` won't send the cookie information, even though the request comes from the same domain.
- **Expiration** — A time stamp indicating when the cookie should be deleted (that is, when it should stop being sent to the server). By default, all cookies are deleted when the browser session ends; however, it is possible to set another time for the deletion. This value is set as a date in GMT format (Wdy, DD-Mon-YYYY HH:MM:SS GMT) and specifies an exact time when the cookie should be deleted. Because of this, a cookie can remain on a user's machine even after the browser is closed. Cookies can be deleted immediately by setting an expiration date that has already occurred.
- **Secure flag** — When specified, the cookie information is sent to the server only if an SSL connection is used. For instance, requests to `https://www.wrox.com` should send cookie information, whereas requests to `http://www.wrox.com` should not.

Each piece of information is specified as part of the `Set-Cookie` header using a semicolon-space combination to separate each section, as shown in the following example:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value; expires=Mon, 22-Jan-07 07:10:24 GMT; domain=.wrox.com
Other-header: other-header-value
```

This header specifies a cookie called "name" that expires on Monday, January 22, 2007, at 7:10:24 GMT and is valid for `www.wrox.com` and any other subdomains of `wrox.com` such as `p2p.wrox.com`.

The `secure` flag is the only part of a cookie that is not a name-value pair; the word "`secure`" is simply included. Consider the following example:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value; domain=.wrox.com; path=/; secure
Other-header: other-header-value
```

Here, a cookie is created that is valid for all subdomains of `wrox.com` and all pages on that domain (as specified by the `path` argument). This cookie can be transmitted only over an SSL connection because the `secure` flag is included.

It's important to note that the domain, path, expiration date, and `secure` flag are indications to the browser as to when the cookie should be sent with a request. These arguments are not actually sent as part of the cookie information to the server; only the name-value pairs are sent.

Cookies in JavaScript

Dealing with cookies in JavaScript is a little complicated because of a notoriously poor interface, the BOM's `document.cookie` property. This property is unique in that it behaves very differently depending on how it is used. When used to retrieve the property value, `document.cookie` returns

a string of all cookies available to the page (based on the domain, path, expiration, and security settings of the cookies) as a series of name-value pairs separated by semicolons, as in the following example:

```
name1=value1;name2=value2;name3=value3
```

All of the names and values are URL-encoded and so must be decoded via `decodeURIComponent()`.

When used to set a value, the `document.cookie` property can be set to a new cookie string. That cookie string is interpreted and added to the existing set of cookies. Setting `document.cookie` does not overwrite any cookies unless the name of the cookie being set is already in use. The format to set a cookie is as follows, which is the same format used by the `Set-Cookie` header:

```
name=value; expires=expiration_time; path=domain_path; domain=domain_name; secure
```

Of these parameters, only the cookie's name and value are required. Here's a simple example:

```
document.cookie = "name=Nicholas";
```

This code creates a session cookie called "name" that has a value of "Nicholas". This cookie will be sent every time the client makes a request to the server; it will be deleted when the browser is closed. Although this will work, as there are no characters that need to be encoded in either the name or the value, it's a best practice to always use `encodeURIComponent()` when setting a cookie, as shown in the following example:

```
document.cookie = encodeURIComponent("name") + "=" +
    encodeURIComponent("Nicholas");
```

To specify additional information about the created cookie, just append it to the string in the same format as the `Set-Cookie` header, like this:

```
document.cookie = encodeURIComponent("name") + "=" +
    encodeURIComponent("Nicholas") + "; domain=.wrox.com;
path=/;
```

Since the reading and writing of cookies in JavaScript isn't very straightforward, functions are often used to simplify cookie functionality. There are three basic cookie operations: reading, writing, and deleting. These are all represented in the `CookieUtil` object as follows:



Available for
download on
Wrox.com

```
var CookieUtil = {

    get: function (name) {
        var cookieName = encodeURIComponent(name) + "=",
            cookieStart = document.cookie.indexOf(cookieName),
            cookieValue = null;

        if (cookieStart > -1) {
            var cookieEnd = document.cookie.indexOf(";", cookieStart);
            if (cookieEnd == -1) {
                cookieEnd = document.cookie.length;
            }
            cookieValue = decodeURIComponent(document.cookie.substring(cookieStart
                , cookieEnd));
        }
        return cookieValue;
    }

};
```

```
        + cookieName.length, cookieEnd));
    }

    return cookieValue;
},
set: function (name, value, expires, path, domain, secure) {
    var cookieText = encodeURIComponent(name) + "=" +
        encodeURIComponent(value);

    if (expires instanceof Date) {
        cookieText += "; expires=" + expires.toGMTString();
    }

    if (path) {
        cookieText += "; path=" + path;
    }

    if (domain) {
        cookieText += "; domain=" + domain;
    }

    if (secure) {
        cookieText += "; secure";
    }

    document.cookie = cookieText;
},
unset: function (name, path, domain, secure){
    this.set(name, "", new Date(0), path, domain, secure);
}
};
```

[CookieUtil.js](#)

The `CookieUtil.get()` method retrieves the value of a cookie with the given name. To do so, it looks for the occurrence of the cookie name followed by an equal sign in `document.cookie`. If that pattern is found, then `indexOf()` is used to find the next semicolon after that location (which indicates the end of the cookie). If the semicolon isn't found, this means that the cookie is the last one in the string, so the entire rest of the string should be considered the cookie value. This value is decoded using `decodeURIComponent()` and returned. In the case where the cookie isn't found, `null` is returned.

The `CookieUtil.set()` method sets a cookie on the page and accepts several arguments: the name of the cookie, the value of the cookie, an optional `Date` object indicating when the cookie should be deleted, an optional URL path for the cookie, an optional domain for the cookie, and an optional Boolean value indicating if the `secure` flag should be added. The arguments are in the order in which they are most frequently used, and only the first two are required. Inside the method, the name and value are URL-encoded using `encodeURIComponent()`, and then the other options are checked. If the `expires` argument is a `Date` object, then an `expires` option is added using the `Date` object's `toGMTString()` method to format the date correctly. The rest of the method simply builds up the cookie string and sets it to `document.cookie`.

There is no direct way to remove existing cookies. Instead, you need to set the cookie again — with the same path, domain, and secure options — and set its expiration date to some time in the past. The `CookieUtil.unset()` method handles this case. It accepts four arguments: the name of the cookie to remove, an optional path argument, an optional domain argument, and an optional secure argument.

These arguments are passed through to `CookieUtil.set()` with the value set to a blank string and the expiration date set to January 1, 1970 (the value of a `Date` object initialized to 0 milliseconds). Doing so ensures that the cookie is removed.

These methods can be used as follows:



Available for
download on
[Wrox.com](#)

```
//set cookies
CookieUtil.set("name", "Nicholas");
CookieUtil.set("book", "Professional JavaScript");

//read the values
alert(CookieUtil.get("name")); //Nicholas
alert(CookieUtil.get("book")); //Professional JavaScript

//remove the cookies
CookieUtil.unset("name");
CookieUtil.unset("book");

//set a cookie with path, domain, and expiration date
CookieUtil.set("name", "Nicholas", "/books/projs/", "www.wrox.com",
               new Date("January 1, 2010"));

//delete that same cookie
CookieUtil.unset("name", "/books/projs/", "www.wrox.com");

//set a secure cookie
CookieUtil.set("name", "Nicholas", null, null, null, true);
```

[CookieExample01.htm](#)

These methods make using cookies to store data on the client easier by handling the parsing and cookie string construction tasks.

Subcookies

To get around the per-domain cookie limit imposed by browsers, some developers use a concept called *subcookies*. Subcookies are smaller pieces of data stored within a single cookie. The idea is to use the cookie's value to store multiple name-value pairs within a single cookie. The most common format for subcookies is as follows:

```
name=name1=value1&name2=value2&name3=value3&name4=value4&name5=value5
```

Subcookies tend to be formatted in query string format. These values can then be stored and accessed using a single cookie, rather than using a different cookie for each name-value pair. The result is that more structured data can be stored by a website or web application without reaching the per-domain cookie limit.

To work with subcookies, you need a new set of methods. The parsing and serialization of subcookies are slightly different and a bit more complicated because of the expected subcookie usage. To get a subcookie, for example, you need to follow the same basic steps to get a cookie, but before decoding the value, you need to find the subcookie information as follows:



Available for
download on
Wrox.com

```
var SubCookieUtil = {

    get: function (name, subName) {
        var subCookies = this.getAll(name);
        if (subCookies) {
            return subCookies[subName];
        } else {
            return null;
        }
    },

    getAll: function(name) {
        var cookieName = encodeURIComponent(name) + "=",
            cookieStart = document.cookie.indexOf(cookieName),
            cookieValue = null,
            cookieEnd,
            subCookies,
            i,
            parts,
            result = {};

        if (cookieStart > -1) {
            cookieEnd = document.cookie.indexOf(";", cookieStart);
            if (cookieEnd == -1) {
                cookieEnd = document.cookie.length;
            }
            cookieValue = document.cookie.substring(cookieStart +
                cookieName.length, cookieEnd);

            if (cookieValue.length > 0) {
                subCookies = cookieValue.split("&");

                for (i=0, len=subCookies.length; i < len; i++) {
                    parts = subCookies[i].split("=");
                    result[decodeURIComponent(parts[0])] =
                        decodeURIComponent(parts[1]);
                }
            }
            return result;
        }
        return null;
    },
    //more code here
};


```

[SubCookieUtil.js](#)

There are two methods for retrieving subcookies: `get()` and `getAll()`. Whereas `get()` retrieves a single subcookie value, `getAll()` retrieves all subcookies and returns them in an object whose properties are equal to the subcookie names and the values are equal to the subcookie values. The `get()` method accepts two arguments: the name of the cookie and the name of the subcookie. It simply calls `getAll()` to retrieve all of the subcookies and then returns just the one of interest (or `null` if the cookie doesn't exist).

The `SubCookieUtil.getAll()` method is very similar to `CookieUtil.get()` in the way it parses a cookie value. The difference is that the cookie value isn't immediately decoded. Instead, it is split on the ampersand character to get all subcookies into an array. Then, each subcookie is split on the equal sign so that the first item in the parts array is the subcookie name, and the second is the subcookie value. Both items are decoded using `decodeURIComponent()` and assigned on the result object, which is returned as the method value. If the cookie doesn't exist, then `null` is returned.

These methods can be used as follows:



Available for download on Wrox.com

```
//assume document.cookie=data=name=Nicholas&book=Professional%20JavaScript

//get all subcookies
var data = SubCookieUtil.getAll("data");
alert(data.name); // "Nicholas"
alert(data.book); // "Professional JavaScript"

//get subcookies individually
alert(SubCookieUtil.get("data", "name")); // "Nicholas"
alert(SubCookieUtil.get("data", "book")); // "Professional JavaScript"
```

SubCookiesExample01.htm

To write subcookies, you can use two methods: `set()` and `setAll()`. The following code shows their constructs:

```
var SubCookieUtil = {

    set: function (name, subName, value, expires, path, domain, secure) {
        var subcookies = this.getAll(name) || {};
        subcookies[subName] = value;
        this.setAll(name, subcookies, expires, path, domain, secure);
    },

    setAll: function(name, subcookies, expires, path, domain, secure){

        var cookieText = encodeURIComponent(name) + "=",
            subcookieParts = new Array(),
            subName;

        for (subName in subcookies){
            if (subName.length > 0 && subcookies.hasOwnProperty(subName)){
                subcookieParts.push(encodeURIComponent(subName) + "=" +
                    encodeURIComponent(subcookies[subName]));
            }
        }
    }
}
```

```

        }

        if (cookieParts.length > 0){
            cookieText += subcookieParts.join("&");

            if (expires instanceof Date) {
                cookieText += "; expires=" + expires.toGMTString();
            }

            if (path) {
                cookieText += "; path=" + path;
            }

            if (domain) {
                cookieText += "; domain=" + domain;
            }

            if (secure) {
                cookieText += "; secure";
            }
        } else {
            cookieText += "; expires=" + (new Date(0)).toGMTString();
        }

        document.cookie = cookieText;

    },
    //more code here
};

```

[SubCookieUtil.js](#)

The `set()` method accepts seven arguments: the cookie name, the subcookie name, the subcookie value, an optional `Date` object for the cookie expiration day/time, an optional cookie path, an optional cookie domain, and an optional Boolean `secure` flag. All of the optional arguments refer to the cookie itself and not to the subcookie. In order to store multiple subcookies in the same cookie, the path, domain, and `secure` flag must be the same; the expiration date refers to the entire cookie and can be set whenever an individual subcookie is written. Inside the method, the first step is to retrieve all of the subcookies for the given cookie name. The logical OR operator is used to set `subcookies` to a new object if `getAll()` returns `null`. After that, the subcookie value is set on the `subcookies` object and then passed into `setAll()`.

The `setAll()` method accepts six arguments: the cookie name, an object containing all of the subcookies, and then the rest of the optional arguments used in `set()`. This method iterates over the properties of the second argument using a `for-in` loop. To ensure that the appropriate data is saved, use the `hasOwnProperty()` method to ensure that only the instance properties are serialized into subcookies. Since it's possible to have a property name equal to the empty string, the length of the property name is also checked before being added to the result. Each subcookie name-value pair is added to the `subcookieParts` array so that they can later be

easily joined with an ampersand using the `join()` method. The rest of the method is the same as `CookieUtil.set()`.

These methods can be used as follows:



Available for
download on
Wrox.com

```
//assume document.cookie=data=name=Nicholas&book=Professional%20JavaScript

//set two subcookies
SubCookieUtil.set("data", "name", "Nicholas");
SubCookieUtil.set("data", "book", "Professional JavaScript");

//set all subcookies with expiration date
SubCookieUtil.setAll("data", { name: "Nicholas", book: "Professional JavaScript" },
    new Date("January 1, 2010"));

//change the value of name and change expiration date for cookie
SubCookieUtil.set("data", "name", "Michael", new Date("February 1, 2010"));
```

SubCookiesExample01.htm

The last group of subcookie methods has to do with removing subcookies. Regular cookies are removed by setting the expiration date to some time in the past, but subcookies cannot be removed as easily. In order to remove a subcookie, you need to retrieve all subcookies contained within the cookie, eliminate just the one that is meant to be removed, and then set the value of the cookie back with the remaining subcookie values. Consider the following:

```
var SubCookieUtil = {

    //more code here

    unset: function (name, subName, path, domain, secure){
        var subcookies = this.getAll(name);
        if (subcookies){
            delete subcookies[subName];
            this.setAll(name, subcookies, null, path, domain, secure);
        }
    },

    unsetAll: function(name, path, domain, secure){
        this.setAll(name, null, new Date(0), path, domain, secure);
    }

};
```

SubCookieUtil.js

The two methods defined here serve two different purposes. The `unset()` method is used to remove a single subcookie from a cookie while leaving the rest intact; whereas the `unsetAll()` method is the equivalent of `CookieUtil.unset()`, which removes the entire cookie. As with `set()` and `setAll()`,

the path, domain, and secure flag must match the options with which a cookie was created. These methods can be used as follows:

```
//just remove the "name" subcookie
SubCookieUtil.unset("data", "name");

//remove the entire cookie
SubCookieUtil.unsetAll("data");
```

If you are concerned about reaching the per-domain cookie limit in your work, subcookies are an attractive alternative. You will have to more closely monitor the size of your cookies to stay within the individual cookie size limit.

Cookie Considerations

There is also a type of cookie called *HTTP-only*. HTTP-only cookies can be set either from the browser or from the server but can be read only from the server, because JavaScript cannot get the value of HTTP-only cookies.

Since all cookies are sent as request headers from the browser, storing a large amount of information in cookies can affect the overall performance of browser requests to a particular domain. The larger the cookie information, the longer it will take to complete the request to the server. Even though the browser places size limits on cookies, it's a good idea to store as little information as possible in cookies, to avoid performance implications.

The restrictions on and nature of cookies make them less than ideal for storing large amounts of information, which is why other approaches have emerged.



It is strongly recommended to avoid storing important or sensitive data in cookies. Cookie data is not stored in a secure environment, so any data contained within may be accessible by others. You should avoid storing data such as credit card numbers or personal addresses in cookies.

Internet Explorer User Data

In Internet Explorer 5, Microsoft introduced the concept of persistent user data via a custom behavior. User data allows you to store up to 128KB of data per document and up to 1MB of data per domain. To use persistent user data, you first must specify the `userData` behavior as shown here on an element using CSS:

```
<div style="behavior:url(#default#userData)" id="dataStore"></div>
```

Once an element is using the `userData` behavior, you can save data onto it using the `setAttribute()` method. In order to commit the data into the browser cache, you must then call `save()` and pass in the name of the data store to save to. The data store name is completely arbitrary and is used to differentiate between different sets of data. Consider the following example:



```
var dataStore = document.getElementById("dataStore");
dataStore.setAttribute("name", "Nicholas");
dataStore.setAttribute("book", "Professional JavaScript");
dataStore.save("BookInfo");
```

[UserDataExample01.htm](#)

In this code, two pieces of information are saved on the `<div>` element. After `setAttribute()` is used to store that data, the `save()` method is called with a data store name of "BookInfo". The next time the page is loaded, you can use the `load()` method with the data store name to retrieve the data as follows:

```
dataStore.load("BookInfo");
alert(dataStore.getAttribute("name"));      // "Nicholas"
alert(dataStore.getAttribute("book"));      // "Professional JavaScript"
```

[UserDataExample01.htm](#)

The call to `load()` retrieves all of the information from the "BookInfo" data store and makes it available on the element; the information is not available until explicitly loaded. If `getAttribute()` is called for a name that either doesn't exist or hasn't been loaded, then `null` is returned.

You can explicitly remove data from the element by using the `removeAttribute()` method and passing in the attribute name. Once removed, `save()` must be called again, to commit the changes as shown here:

```
dataStore.removeAttribute("name");
dataStore.removeAttribute("book");
dataStore.save("BookInfo");
```

[UserDataExample01.htm](#)

This code removes two data attributes and then saves those changes to the cache.

The accessibility restrictions on Internet Explorer user data are similar to the restrictions on cookies. In order to access a data store, the page on which the script is running must be from the same domain, on the same directory path, and using the same protocol as the script that saved data to the store. Unlike with cookies, you cannot change accessibility restrictions on user data to a wider range of consumers. Also unlike cookies, user data persists across sessions by default and doesn't expire; data needs to be specifically removed using `removeAttribute()` in order to free up space.



As with cookies, Internet Explorer user data is not secure and should not be used to store sensitive information.

Web Storage

Web Storage was first described in the Web Applications 1.0 specification of the Web Hypertext Application Technical Working Group (WHAT-WG). The initial work from this specification eventually became part of HTML5 before being split into its own specification. Its intent is to overcome some of the limitations imposed by cookies when data is needed strictly on the client side, with no need to continuously send data back to the server. The two primary goals of Web Storage are:

- To provide a way to store session data outside of cookies.
- To provide a mechanism for storing large amounts of data that persists across sessions.

The original Web Storage specification includes definitions for two objects: `localStorage` and `globalStorage`. These objects are available as a property of `window` in Internet Explorer 8+, Firefox 3.5+, Safari 3.1+, Chrome 4+, and Opera 10.5+.



Firefox 2 and 3 had partial implementations of Web Storage that were based on earlier work where an object called `globalStorage` was implemented instead of `localStorage`.

The Storage Type

The `Storage` type is designed to hold name-value pairs up to a maximum size (determined by the browser). An instance of `Storage` acts like any other object and has the following additional methods:

- `clear()` — Removes all values; not implemented in Firefox.
- `getItem(name)` — Retrieves the value for the given `name`.
- `key(index)` — Retrieves the name of the value in the given numeric position.
- `removeItem(name)` — Removes the name-value pair identified by `name`.
- `setItem(name, value)` — Sets the value for the given `name`.

The `getItem()`, `removeItem()`, and `setItem()` methods can be called directly or indirectly by manipulating the `Storage` object. Since each item is stored on the object as a property, you can simply read values by accessing the property with dot or bracket notation, set the value by doing the same, or remove it by using the `delete` operator. Even so, it's generally recommended to use the methods instead of property access to ensure you don't end up overwriting one of the already available object members with a key.

You can determine how many name-value pairs are in a `Storage` object by using the `length` property. It's not possible to determine the size of all data in the object, although Internet Explorer 8 provides a `remainingSpace` property that retrieves the amount of space, in bytes, that is still available for storage.



*The **Storage** type is capable of storing only strings. Nonstring data is converted into a string before being stored.*

The sessionStorage Object

The `sessionStorage` object stores data only for a session, meaning that the data is stored until the browser is closed. This is the equivalent of a session cookie that disappears when the browser is closed. Data stored on `sessionStorage` persists across page refreshes and may also be available if the browser crashes and is restarted, depending on the browser vendor. (Firefox and WebKit support this, but Internet Explorer does not.)

Because the `sessionStorage` object is tied to a server session, it isn't available when a file is run locally. Data stored on `sessionStorage` is accessible only from the page that initially placed the data onto the object, making it of limited use for multipage applications.

Since the `sessionStorage` object is an instance of `Storage`, you can assign data onto it either by using `setItem()` or by assigning a new property directly. Here's an example of each of these methods:



Available for download on
Wrox.com

```
//store data using method
sessionStorage.setItem("name", "Nicholas");

//store data using property
sessionStorage.book = "Professional JavaScript";
```

[SessionStorageExample01.htm](#)

Writing to storage has slight differences from browser to browser. Firefox and WebKit implement storage writing synchronously, so data added to storage is committed right away. The Internet Explorer implementation writes data asynchronously, so there may be a lag between the time when data is assigned and the time that the data is written to disk. For small amounts of data, the difference is negligible. For large amounts of data, you'll notice that JavaScript in Internet Explorer resumes execution faster than in other browsers, because it offloads the actual disk write process.

You can force disk writing to occur in Internet Explorer 8 by using the `begin()` method before assigning any new data, and the `commit()` method after all assignments have been made. Consider the following example:

```
//IE8 only
sessionStorage.begin();
sessionStorage.name = "Nicholas";
sessionStorage.book = "Professional JavaScript";
sessionStorage.commit();
```

This code ensures that the values for `"name"` and `"book"` are written as soon as `commit()` is called. The call to `begin()` ensures that no disk writes will occur while the code is executed. For small amounts of data, this process isn't necessary; however, you may wish to consider this transactional approach for larger amounts of data such as documents.

When data exists on `sessionStorage`, it can be retrieved either by using `getItem()` or by accessing the property name directly. Here's an example of each of these methods:



```
//get data using method
var name = sessionStorage.getItem("name");

//get data using property
var book = sessionStorage.book;
```

[SessionStorageExample01.htm](#)

You can iterate over the values in `sessionStorage` using a combination of the `length` property and `key()` method, as shown here:

```
for (var i=0, len = sessionStorage.length; i < len; i++){
    var key = sessionStorage.key(i);
    var value = sessionStorage.getItem(key);
    alert(key + " = " + value);
}
```

[SessionStorageExample01.htm](#)

The name-value pairs in `sessionStorage` can be accessed sequentially by first retrieving the name of the data in the given position via `key()` and then using that name to retrieve the value via `getItem()`.

It's also possible to iterate over the values in `sessionStorage` using a `for-in` loop:

```
for (var key in sessionStorage){
    var value = sessionStorage.getItem(key);
    alert(key + " = " + value);
}
```

Each time through the loop, `key` is filled with another name in `sessionStorage`; none of the built-in methods or the `length` property will be returned.

To remove data from `sessionStorage`, you can use either the `delete` operator on the object property or the `removeItem()` method. Here's an example of each of these methods:

```
//use delete to remove a value - won't work in WebKit
delete sessionStorage.name;

//use method to remove a value
sessionStorage.removeItem("book");
```

[SessionStorageExample01.htm](#)

It's worth noting that as of the time of this writing, the `delete` operator doesn't remove data in WebKit, whereas `removeItem()` works correctly across all supporting browsers.

The `sessionStorage` object should be used primarily for small pieces of data that are valid only for a session. If you need to persist data across sessions, then either `globalStorage` or `localStorage` is more appropriate.

The `globalStorage` Object

The `globalStorage` object is implemented in Firefox 2. As part of the original Web Storage specification, its purpose is to persist data across sessions and with specific access restrictions. In order to use `globalStorage`, you need to specify the domains for which the data should be available. This is done using a property via bracket notation, as shown in the following example:



```
//save value
globalStorage["wrox.com"].name = "Nicholas";

//get value
var name = globalStorage["wrox.com"].name;
```

[GlobalStorageExample01.htm](#)

Here, a storage area for the domain `wrox.com` is accessed. Whereas the `globalStorage` object itself is not an instance of `Storage`, the `globalStorage["wrox.com"]` specification is and can be used accordingly. This storage area is accessible from `wrox.com` and all subdomains. You can limit the subdomain by specifying it as follows:

```
//save value
globalStorage["www.wrox.com"].name = "Nicholas";

//get value
var name = globalStorage["www.wrox.com"].name;
```

[GlobalStorageExample01.htm](#)

The storage area specified here is accessible only from a page on `www.wrox.com`, excluding other subdomains.

Some browsers allow more general access restrictions, such as those limited only by top-level domains (TLDs) or by allowing global access, such as in the following example:

```
//store data that is accessible to everyone - AVOID!
globalStorage[""].name = "Nicholas";

//store data available only to domains ending with .net - AVOID!
globalStorage[".net"].name = "Nicholas";
```

Even though these are supported, it is recommended to avoid using generally accessible data stores, to prevent possible security issues. It's also possible that because of security concerns, this ability will be either removed or severely limited in the future, so applications should not rely on this type of functionality. Always specify a domain name when using `globalStorage`.

Access to `globalStorage` areas is limited by the domain, protocol, and port of the page making the request. For instance, if data is stored for `wrox.com` while using the HTTPS protocol, a page on `wrox.com` accessed via HTTP cannot access that information. Likewise, a page accessed via port 80 cannot share data with a page on the same domain and use the same protocol that is accessed on port 8080. This is similar to the same-origin policy for Ajax requests.

Each property of `globalStorage` is an instance of `Storage`. Therefore, it can be used as in the following example:



```
globalStorage["www.wrox.com"].name = "Nicholas";
globalStorage["www.wrox.com"].book = "Professional JavaScript";
globalStorage["www.wrox.com"].removeItem("name");

var book = globalStorage["www.wrox.com"].getItem("book");
```

[GlobalStorageExample01.htm](#)

If you aren't certain of the domain name to use ahead of time, it may be safer to use `location.host` as the property name. For example:

```
globalStorage[location.host].name = "Nicholas";
var book = globalStorage[location.host].getItem("book");
```

[GlobalStorageExample01.htm](#)

The data stored in a `globalStorage` property remains on disk until it's removed via either `removeItem()` or `delete`, or until the user clears the browser's cache. This makes `globalStorage` ideal for storing documents on the client or persisting user settings.

The `localStorage` Object

The `localStorage` object superceded `globalStorage` in the revised HTML5 specification as a way to store persistent client-side data. Unlike with `globalStorage`, you cannot specify any accessibility rules on `localStorage`; the rules are already set. In order to access the same `localStorage` object, pages must be served from the same domain (subdomains aren't valid), using the same protocol, and on the same port. This is effectively the same as `globalStorage[location.host]`.

Since `localStorage` is an instance of `Storage`, it can be used in the same manner as `sessionStorage`. Here are some examples:

```
//store data using method
localStorage.setItem("name", "Nicholas");

//store data using property
localStorage.book = "Professional JavaScript";

//get data using method
```

```
var name = localStorage.getItem("name");
//get data using property
var book = localStorage.book;
```

[LocalStorageExample01.htm](#)

Data that is stored in `localStorage` follows the same rules as data stored in `globalStorage`, because the data is persisted until it is specifically removed via JavaScript or the user clears the browser's cache.

To equalize for browsers that support only `globalStorage`, you can use the following function:



Available for download on
Wrox.com

```
function getLocalStorage(){
    if (typeof localStorage == "object"){
        return localStorage;
    } else if (typeof globalStorage == "object"){
        return globalStorage[location.host];
    } else {
        throw new Error("Local storage not available.");
    }
}
```

[GlobalAndLocalStorageExample01.htm](#)

Then, the following initial call to the function is all that is necessary to identify the correct location for data:

```
var storage = getLocalStorage();
```

[GlobalAndLocalStorageExample01.htm](#)

After determining which `Storage` object to use, you can easily continue storing and retrieving data with the same access rules across all browsers that support Web Storage.

The storage Event

Whenever a change is made to a `Storage` object, the `storage` event is fired on the document. This occurs for every value set using either properties or `setItem()`, every value removal using either `delete` or `removeItem()`, and every call to `clear()`. The event object has the following four properties:

- `domain` — The domain for which the storage changed.
- `key` — The key that was set or removed.
- `newValue` — The value that the key was set to, or null if the key was removed.
- `oldValue` — The value prior to the key being changed.

Of these four properties, Internet Explorer 8 and Firefox have implemented only the `domain` property. WebKit doesn't support the `storage` event as of the date of this writing.



```
EventUtil.addHandler(document, "storage", function(event) {
    alert("Storage changed for " + event.domain);
});
```

StorageEventExample01.htm

The `storage` event is fired for all changes to `sessionStorage` and `localStorage` but doesn't distinguish between them.

Limits and Restrictions

As with other client-side data storage solutions, Web Storage also has limitations. These limitations are browser-specific. Generally speaking, the size limit for client-side data is set on a per-origin (protocol, domain, and port) basis, so each origin has a fixed amount of space in which to store its data. Analyzing the origin of the page that is storing the data enforces this restriction.

Most desktop browsers have a 5MB per-origin limit for `localStorage`. Chrome and Safari have a per-origin limit of 2.5MB. Safari for iOS and WebKit for Android also have a limit of 2.5MB.

The limits for `sessionStorage` vary across browsers. Many browsers have no limit on `sessionStorage` data, while Chrome, Safari, Safari for iOS, and WebKit for Android have a limit of 2.5MB. Internet Explorer 8+ and Opera have a limit of 5MB for `sessionStorage`.

For more information about Web Storage limits, please see the Web Storage Support Test at <http://dev-test.nemikor.com/web-storage/support-test/>.

IndexedDB

The Indexed Database API, *IndexedDB* for short, is a structured data store in the browser. IndexedDB came about as an alternative to the now-deprecated Web SQL Database API (not covered in this book because of its deprecated state). The idea behind IndexedDB was to create an API that easily allowed the storing and retrieval of JavaScript objects while still allowing querying and searching.

IndexedDB is designed to be almost completely asynchronous. As a result, most operations are performed as requests that will execute later and produce either a successful result or an error. Nearly every IndexedDB operation requires you to attach `onerror` and `onsuccess` event handlers to determine the outcome.

Once fully supported, there will be a global `indexedDB` object that serves as the API host. While the API is still in flux, browsers are using vendor prefixes, so the object in Internet Explorer 10 is called `msIndexedDB`, in Firefox 4 it's called `mozIndexedDB`, and in Chrome it's called `webkitIndexedDB`. This section uses `indexedDB` in the examples for clarity, so you may need to include the following code before each example:

```
var indexedDB = window.indexedDB || window.msIndexedDB || window.mozIndexedDB ||
window.webkitIndexedDB;
```

IndexedDBExample01.htm

Databases

IndexedDB is a database similar to databases you've probably used before such as MySQL or Web SQL Database. The big difference is that IndexedDB uses object stores instead of tables to keep track of data. An IndexedDB database is simply a collection of object stores grouped under a common name.

The first step to using a database is to open it using `indexedDB.open()` and passing in the name of the database to open. If a database with the given name already exists, then a request is made to open it; if the database doesn't exist, then a request is made to create and open it. The call to `indexDB.open()` returns an instance of `IDBRequest` onto which you can attach `onerror` and `onsuccess` event handlers. Here's an example:



Available for
download on
Wrox.com

```
var request, database;
request = indexedDB.open("admin");
request.onerror = function(event){
    alert("Something bad happened while trying to open: " +
        event.target.errorCode);
};
request.onsuccess = function(event){
    database = event.target.result;
};
```

IndexedDBExample01.htm

In both event handlers, `event.target` points to `request`, so these may be used interchangeably. If the `onsuccess` event handler is called, then the database instance object (`IDBDatabase`) is available in `event.target.result` and stored in the `database` variable. From this point on, all requests to work with the database are made through the `database` object itself. If an error occurs, an error code stored in `event.target.errorCode` indicates the nature of the problem as one of the following (these error codes apply to all operations):

- `IDBDatabaseException.UNKNOWN_ERR` (1) — The error is unexpected and doesn't fall into an available category.
- `IDBDatabaseException.NON_TRANSIENT_ERR` (2) — The operation is not allowed.
- `IDBDatabaseException.NOT_FOUND_ERR` (3) — The database on which to perform the operation is not found.
- `IDBDatabaseException.CONSTRAINT_ERR` (4) — A database constraint was violated.
- `IDBDatabaseException.DATA_ERR` (5) — Data provided for the transaction doesn't fulfill the requirements.
- `IDBDatabaseException.NOT_ALLOWED_ERR` (6) — The operation is not allowed.
- `IDBDatabaseException.TRANSACTION_INACTIVE_ERR` (7) — An attempt was made to reuse an already completed transaction.
- `IDBDatabaseException.ABORT_ERR` (8) — The request was aborted and so did not succeed.
- `IDBDatabaseException.READ_ONLY_ERR` (9) — Attempt to write or otherwise change data while in read-only mode.

- `IDBDatabaseException.TIMEOUT_ERR` (10) — The operation could not be completed in the amount of time available.
- `IDBDatabaseException.QUOTA_ERR` (11) — Not enough remaining disk space.

By default, a database has no version associated with it, so it's a good idea to set the initial version when starting out. To do so, call the `setVersion()` method and pass in the version as a string. Once again, this creates a request object on which you'll need to assign event handlers:



```
if (database.version != "1.0"){
    request = database.setVersion("1.0");
    request.onerror = function(event){
        alert("Something bad happened while trying to set version: " +
            event.target.errorCode);
    };
    request.onsuccess = function(event){
        alert("Database initialization complete. Database name: " + database.name +
            ", Version: " + database.version);
    };
} else {
    alert("Database already initialized. Database name: " + database.name +
        ", Version: " + database.version);
}
```

IndexedDBExample01.htm

This example tries to set the version of the database to “1.0”. The first line checks the `version` property to see if the database version has already been set. If not, then `setVersion()` is called to create the version change request. If that request is successful, then a message is displayed indicating that the version change is complete. (In a real implementation, this is where you would set up your object stores. See the next section for details.)

If the database version is already “1.0”, then a message is displayed stating that the database has already been initialized. This basic pattern is how you can tell if the database you want to use has already been set up with appropriate object stores or not. Over the course of a web application, you may have many different versions of the database as you update and modify the data structures.

Object Stores

Once you have established a connection to the database, the next step is to interact with object stores. If the database version doesn't match the one you expect then you likely will need to create an object store. Before creating an object store, however, it's important to think about the type of data you want to store.

Suppose that you'd like to store user records containing `username`, `password`, and so on. The object to hold a single record may look like this:

```
var user = {
    username: "007",
    firstName: "James",
    lastName: "Bond",
    password: "foo"
};
```

Looking at this object, you can easily see that an appropriate key for this object store is the `username` property. A `username` must be globally unique, and it's probably the way you'll be accessing data most of the time. This is important because you must specify a key when creating an object store. Here's how you would create an object store for these users:

```
var store = db.createObjectStore("users", { keyPath: "username" });
```

[IndexedDBExample02.htm](#)

The `keyPath` property of the second argument indicates the property name of the stored objects that should be used as a key.

Since you now have a reference to the object store, it's possible to populate it with data using either `add()` or `put()`. Both of these methods accept a single argument, the object to store, and save the object into the object store. The difference between these two occurs only when an object with the same key already exists in the object store. In that case, `add()` will cause an error while `put()` will simply overwrite the object. More simply, think of `add()` as being used for inserting new values while `put()` is used for updating values. So to initialize an object store for the first time, you may want to do something like this:



Available for download on
Wrox.com

```
//where users is an array of new users
var i=0,
    len = users.length;

while(i < len){
    store.add(users[i++]);
}
```

[IndexedDBExample02.htm](#)

Each call to `add()` or `put()` creates a new update request for the object store. If you want verification that the request completed successfully, you can store the request object in a variable and assign `onerror` and `onsuccess` event handlers:

```
//where users is an array of new users
var i=0,
    request,
    requests = [],
    len = users.length;

while(i < len){
    request = store.add(users[i++]);
    request.onerror = function(){
        //handle error
    };
    request.onsuccess = function(){
        //handle success
    };
    requests.push(request);
}
```

Once the object store is created and filled with data, it's time to start querying.

Transactions

Past the creation step of an object store, all further operations are done through *transactions*. A transaction is created using the `transaction()` method on the database object. Any time you want to read or change data, a transaction is used to group all changes together. In its simplest form, you create a new transaction as follows:

```
var transaction = db.transaction();
```

With no arguments specified, you have read-only access to all object stores in the database. To be more optimal, you can specify one or more object store names that you want to access:

```
var transaction = db.transaction("users");
```

This ensures that only information about the `users` object store is loaded and available during the transaction. If you want access to more than one object store, the first argument can also be an array of strings:

```
var transaction = db.transaction(["users", "anotherStore"]);
```

As mentioned previously, each of these transactions accesses data in a read-only manner. To change that, you must pass in a second argument indicating the access mode. These constants are accessible on `IDBTransaction` as `READ_ONLY` (0), `READ_WRITE` (1), and `VERSION_CHANGE` (2). While Internet Explorer 10+ and Firefox 4+ implement `IDBTransaction`, Chrome supports it via `webkitIDBTransaction`, so the following code is necessary to normalize the interface:



Available for
download on
Wrox.com

```
var IDBTransaction = window.IDBTransaction || window.webkitIDBTransaction;
```

[IndexedDBExample03.htm](#)

With that setup, you can specify the second argument to `transaction()`:

```
var transaction = db.transaction("users", IDBTransaction.READ_WRITE);
```

[IndexedDBExample03.htm](#)

This transaction is capable of both reading and writing into the `users` object store.

Once you have a reference to the transaction, you can access a particular object store using the `objectStore()` method and passing in the store name you want to work with. You can then use `add()` and `put()` as before, as well as `get()` to retrieve values, `delete()` to remove an object, and `clear()` to remove all objects. The `get()` and `delete()` methods each accept an object key as their argument, and all five of these methods create a new request object. For example:



```
var request = db.transaction("users").objectStore("users").get("007");
request.onerror = function(event){
    alert("Did not get the object!");
};
request.onsuccess = function(event){
    var result = event.target.result;
    alert(result.firstName);      // "James"
};
```

[IndexedDBExample02.htm](#)

Because any number of requests can be completed as part of a single transaction, the transaction object itself also has event handlers: `onerror` and `oncomplete`. These are used to provide transaction-level state information:

```
transaction.onerror = function(event){
    //entire transaction was cancelled
};

transaction.oncomplete = function(event){
    //entire transaction completed successfully
};
```

Keep in mind that the `event` object for `oncomplete` doesn't give you access to any data returned by `get()` requests, so you still need an `onsuccess` event handler for those types of requests.

Querying with Cursors

Transactions can be used directly to retrieve a single item with a known key. When you want to retrieve multiple items, you need to create a *cursor* within the transaction. A cursor is a pointer into a result set. Unlike traditional database queries, a cursor doesn't gather all of the result set up front. Instead, a cursor points to the first result and doesn't try to find the next until instructed to do so.

Cursors are created using the `openCursor()` method on an object store. As with other operations with IndexedDB, the return value of `openCursor()` is a request, so you must assign `onsuccess` and `onerror` event handlers. For example:

```
var store = db.transaction("users").objectStore("users"),
    request = store.openCursor();

request.onsuccess = function(event){
    //handle success
};

request.onerror = function(event){
    //handle failure
};
```

[IndexedDBExample04.htm](#)

When the `onsuccess` event handler is called, the next item in the object store is accessible via `event.target.result`, which holds an instance of `IDBCursor` when there is a next item or `null` when there are no further items. The `IDBCursor` instance has several properties:

- `direction` — A numeric value indicating the direction the cursor should travel in. The default is `IDBCursor.NEXT` (0) for next. Other values include `IDBCursor.NEXT_NO_DUPLICATE` (1) for next without duplicates, `IDBCursor.PREV` (2) for previous, and `IDBCursor.PREV_NO_DUPLICATE` (3) for previous without duplicates.
- `key` — The key for the object.
- `value` — The actual object.
- `primaryKey` — The key being used by the cursor. Could be the object key or an index key (discussed later).

You can retrieve information about a single result using the following:

```
request.onsuccess = function(event){  
    var cursor = event.target.result;  
    if (cursor){ //always check  
        console.log("Key: " + cursor.key + ", Value: " +  
            JSON.stringify(cursor.value));  
    }  
};
```

Keep in mind that `cursor.value` in this example is an object, which is why it is JSON encoded before being displayed.

A cursor can be used to update an individual record. The `update()` method updates the current cursor value with the specified object. As with other such operations, the call to `update()` creates a new request, so you need to assign `onsuccess` and `onerror` if you want to know the result:

```
request.onsuccess = function(event){  
    var cursor = event.target.result,  
        value,  
        updateRequest;  
  
    if (cursor){ //always check  
        if (cursor.key == "foo"){  
            value = cursor.value; //get current value  
            value.password = "magic!"; //update the password  
  
            updateRequest = cursor.update(value); //request the update be saved  
            updateRequest.onsuccess = function(){  
                //handle success;  
            };  
            updateRequest.onerror = function(){  
                //handle failure  
            };  
        }  
    };  
};
```

You can also delete the item at that position by calling `delete()`. As with `update()`, this also creates a request:

```
request.onsuccess = function(event){
    var cursor = event.target.result,
        value,
        deleteRequest;

    if (cursor){ //always check
        if (cursor.key == "foo"){
            deleteRequest = cursor.delete(); //request the value be deleted
            deleteRequest.onerror = function(){
                //handle success;
            };
            deleteRequest.onfailure = function(){
                //handle failure
            };
        }
    }
};
```

Both `update()` and `delete()` will throw errors if the transaction doesn't have permission to modify the object store.

Each cursor makes only one request by default. To make another request, you must call one of the following methods:

- `continue(key)` — Moves to the next item in the result set. The argument `key` is optional. When not specified, the cursor just moves to the next item; when provided, the cursor will move to the specified key.
- `advance(count)` — Moves the cursor ahead by `count` number of items.

Each of these methods causes the cursor to reuse the same request, so the same `onsuccess` and `onfailure` event handlers are reused until no longer needed. For example, the following iterates over all items in an object store:

```
request.onsuccess = function(event){
    var cursor = event.target.result;
    if (cursor){ //always check
        console.log("Key: " + cursor.key + ", Value: " +
                    JSON.stringify(cursor.value));
        cursor.continue(); //go to the next one
    } else {
        console.log("Done!");
    }
};
```

The call to `continue()` triggers another request and `onsuccess` is called again. When there are no more items to iterate over, `onsuccess` is called one last time with `event.target.result` equal to `null`.

Key Ranges

Working with cursors may seem suboptimal since you're limited in the ways data can be retrieved. *Key ranges* are used to make working with cursors a little more manageable. A key range is represented by an instance of `IDBKeyRange`. The standard version is `IDBKeyRange`, which is supported in Internet Explorer 10+ and Firefox 4+, while Chrome supports this type with `webkitIDBKeyRange`. As with other types related to IndexedDB, you'll first need to create a local copy, taking these differences into account:

```
var IDBKeyRange = window.IDBKeyRange || window.webkitIDBKeyRange;
```

There are four different ways to specify key ranges. The first is to use the `only()` method and pass in the key you want to retrieve:

```
var onlyRange = IDBKeyRange.only("007");
```

This range ensures that only the value with a key of "007" will be retrieved. A cursor created using this range is similar to directly accessing an object store and calling `get("007")`.

The second type of range defines a lower bound for the result set. The lower bound indicates the item at which the cursor should start. For example, the following key range ensures the cursor starts at the key "007" and continues until the end:

```
//start at item "007", go to the end
var lowerRange = IDBKeyRange.lowerBound("007");
```

If you want to start at the item immediately following the value at "007", then you can pass in a second argument of `true`:

```
//start at item after "007", go to the end
var lowerRange = IDBKeyRange.lowerBound("007", true);
```

The third type of range is an upper bound, indicating the key you don't want to go past by using the `upperBound()` method. The following key ensures that the cursor starts at the beginning and stops when it gets to the value with key "ace":

```
//start at beginning, go to "ace"
var upperRange = IDBKeyRange.upperBound("ace");
```

If you don't want to include the given key, then pass in `true` as the second argument:

```
//start at beginning, go to the item just before "ace"
var upperRange = IDBKeyRange.upperBound("ace", true);
```

To specify both a lower and an upper bound, use the `bound()` method. This method accepts four arguments, the lower bound key, the upper bound key, an optional Boolean indicating to skip the lower bound, and an optional Boolean indicating to skip the upper bound. Here are some examples:

```
//start at "007", go to "ace"
var boundRange = IDBKeyRange.bound("007", "ace");

//start at item after "007", go to "ace"
var boundRange = IDBKeyRange.bound("007", "ace", true);

//start at item after "007", go to item before "ace"
var boundRange = IDBKeyRange.bound("007", "ace", true, true);

//start at "007", go to item before "ace"
var boundRange = IDBKeyRange.bound("007", "ace", false, true);
```

Once you have defined a range, pass it into the `openCursor()` method and you'll create a cursor that stays within the constraints:

```
var store = db.transaction("users").objectStore("users"),
    range = IDBKeyRange.bound("007", "ace");
    request = store.openCursor(range);

request.onsuccess = function(event){
    var cursor = event.target.result;
    if (cursor){ //always check
        console.log("Key: " + cursor.key + ", Value: " +
                    JSON.stringify(cursor.value));
        cursor.continue(); //go to the next one
    } else {
        console.log("Done!");
    }
};
```

This example outputs only the values between keys "007" and "ace", which are fewer than the previous section's example.

Setting Cursor Direction

There are actually two arguments to `openCursor()`. The first is an instance of `IDBKeyRange` and the second is a numeric value indicating the direction. These constants are specified as constants on `IDBCursor` as discussed in the querying section. Firefox 4+ and Chrome once again have different implementations, so the first step is to normalize the differences locally:

```
var IDBCursor = window.IDBCursor || window.webkitIDBCursor;
```

Normally cursors start at the first item in the object store and progress toward the last with each call to `continue()` or `advance()`. These cursors have the default direction value of `IDBCursor.NEXT`. If there are duplicates in the object store, you may want to have a cursor that skips over the duplicates. You can do so by passing `IDBCursor.NEXT_NO_DUPLICATE` into `openCursor()` as the second argument:

```
var store = db.transaction("users").objectStore("users"),
    request = store.openCursor(null, IDBCursor.NEXT_NO_DUPLICATE);
```

Note that the first argument to `openCursor()` is null, which indicates that the default key range of all values should be used. This cursor will iterate through the items in the object store starting from the first item and moving toward the last while skipping any duplicates.

You can also create a cursor that moves backward through the object store, starting at the last item and moving toward the first by passing in either `IDBCursor.PREV` or `IDBCursor.PREV_NO_DUPLICATE` (the latter, of course, to avoid duplicates). For example:



```
var store = db.transaction("users").objectStore("users"),
    request = store.openCursor(null, IDBCursor.PREV);
```

IndexedDBExample05.htm

When you open a cursor using `IDBCursor.PREV` or `IDBCursor.PREV_NO_DUPLICATE`, each call to `continue()` or `advance()` moves the cursor backward through the object store instead of forward.

Indexes

For some data sets, you may want to specify more than one key for an object store. For example, if you're tracking users by both a user ID and a username, you may want to access records using either piece of data. To do so, you would likely consider the user ID as the primary key and create an index on the username.

To create a new index, first retrieve a reference to the object store and then call `createIndex()`, as in this example:

```
var store = db.transaction("users").objectStore("users"),
    index = store.createIndex("username", "username", { unique: true });
```

The first argument to `createIndex()` is the name of the index, the second is the name of the property to index, and third is an `options` object containing the key `unique`. This option should always be specified so as to indicate whether or not the key is unique across all records. Since `username` may not be duplicated, this index is not unique.

The returned value from `createIndex()` is an instance of `IDBIndex`. You can also retrieve the same instance via the `index()` method on an object store. For example, to use an already existing index named "username", the code would be:

```
var store = db.transaction("users").objectStore("users"),
    index = store.index("username");
```

An index acts a lot like an object store. You can create a new cursor on the index using the `openCursor()` method, which works exactly the same as `openCursor()` on an object store except that the `result.key` property is filled in with the index key instead of the primary key. Here's an example:

```
var store = db.transaction("users").objectStore("users"),
    index = store.index("username"),
    request = index.openCursor();
```

```
request.onsuccess = function(event) {
```

```

        //handle success
    };

```

An index can also create a special cursor that returns just the primary key for each record using the `openKeyCursor()` method, which accepts the same arguments as `openCursor()`. The big difference is that `event.result.key` is the index key and `event.result.value` is the primary key instead of the entire record.

```

var store = db.transaction("users").objectStore("users"),
    index = store.index("username"),
    request = index.openKeyCursor();

request.onsuccess = function(event){
    //handle success
    //event.result.key is the index key, event.result.value is the primary key
};

```

You can also retrieve a single value from an index by using `get()` and passing in the index key, which creates a new request:

```

var store = db.transaction("users").objectStore("users"),
    index = store.index("username"),
    request = index.get("007");

request.onsuccess = function(event){
    //handle success
};

request.onerror = function(event){
    //handle failure
};

```

To retrieve just the primary key for a given index key, use the `getKey()` method. This also creates a new request but `result.value` is equal to the primary key value rather than the entire record:

```

var store = db.transaction("users").objectStore("users"),
    index = store.index("username"),
    request = index.getKey("007");

request.onsuccess = function(event){
    //handle success
    //event.result.key is the index key, event.result.value is the primary key
};

```

In the `onsuccess` event handler in this example, `event.result.value` would be the user ID.

At any point in time, you can retrieve information about the index by using properties on the `IDBIndex` object:

- `name` — The name of the index.
- `keyPath` — The property path that was passed into `createIndex()`.

- `objectStore` — The object store that this index works on.
- `unique` — A Boolean indicating if the index key is unique.

The object store itself also tracks the indexes by name in the `indexNames` property. This makes it easy to figure out which indexes already exist on an object using the following code:

```
var store = db.transaction("users").objectStore("users"),
    indexNames = store.indexNames,
    index,
    i = 0,
    len = indexNames.length;

while(i < len){
    index = store.index(indexNames[i++]);
    console.log("Index name: " + index.name + ", KeyPath: " + index.keyPath +
        ", Unique: " + index.unique);
}
```

This code iterates over each index and outputs its information to the console.

An index can be deleted by calling the `deleteIndex()` method on an object store and passing in the name of the index:

```
var store = db.transaction("users").objectStore("users");
store.deleteIndex("username");
```

Since deleting an index doesn't touch the data in the object store, the operation happens without any callbacks.

Concurrency Issues

While IndexedDB is an asynchronous API inside of a web page, there are still concurrency issues. If the same web page is open in two different browser tabs at the same time, it's possible that one may attempt to upgrade the database before the other is ready. The problematic operation is in setting the database to a new version, and so calls to `setVersion()` can be completed only when there is just one tab in the browser using the database.

When you first open a database, it's important to assign an `onversionchange` event handler. This callback is executed when another tab from the same origin calls `setVersion()`. The best response to this event is to immediately close the database so that the version upgrade can be completed. For example:

```
var request, database;

request = indexedDB.open("admin");
request.onerror = function(event){
    database = event.target.result;

    database.onversionchange = function(){
        database.close();
    };
};
```

You should assign `onversionchange` after every successful opening of a database.

When you are calling `setVersion()`, it's also important to assign an `onblocked` event handler to the request. This event handler executes when another tab has the database open while you're trying to update the version. In that case, you may want to inform the user to close all other tabs before attempting to retry `setVersion()`. For example:

```
var request = database.setVersion("2.0");
request.onblocked = function(){
    alert("Please close all other tabs and try again.");
};

request.onerror = function(){
    //handle success, continue on
};
```

Remember, `onversionchange` will have been called in the other tab(s) as well.

By always assigning these event handlers, you will ensure your web application will be able to better handle concurrency issues related to IndexedDB.

Limits and Restrictions

Many of the restrictions on IndexedDB are exactly the same as those for Web Storage. First, IndexedDB databases are tied to the origin (protocol, domain, and port) of the page, so the information cannot be shared across domains. This means there is a completely separate data store for `www.wrox.com` as for `p2p.wrox.com`.

Second, there is a limit to the amount of data that can be stored per origin. The current limit in Firefox 4+ is 50MB per origin while Chrome has a limit of 5MB. Firefox for mobile has a limit of 5MB and will ask the user for permission to store more than that if the quota is exceeded.

Firefox imposes an extra limitation that local files cannot access IndexedDB databases at all. Chrome doesn't have this restriction. When running the examples from this book locally, be sure to use Chrome.

SUMMARY

Offline web applications and client-side data storage are a big part of the Web's future. Browsers now have the ability to detect when a user has gone offline and fire events in JavaScript so that your application can respond. The application cache allows you to specify which files should be made available offline. A JavaScript API is available for determining what the application cache state is and how it may be changing.

This chapter also covered the following aspects of client-side storage:

- Traditionally, such storage was limited to using cookies, small pieces of information that could be set from the client or server and transmitted along with every request.
- JavaScript provides access to cookies through `document.cookie`.

- The limitations placed on cookies make them okay for storing small amounts of data but inefficient for storing large amounts.

Internet Explorer provides a behavior called user data that can be applied to an element on the page as follows:

- Once applied, the element can load data from a named data store and make the information accessible via the `getAttribute()`, `setAttribute()`, and `removeAttribute()` methods.
- The data must be explicitly saved to a named data store using the `save()` method for it to persist between sessions.

Web Storage defines two objects to save data: `sessionStorage` and `localStorage`. The former is used strictly to save data within a browser session, because the data is removed once the browser is closed. The latter is used to persist data across sessions and based on cross-domain security policies.

IndexedDB is a structured data storage mechanism similar to an SQL database. Instead of storing data in tables, data is stored in object stores. Object stores are created by defining a key and then adding data. Cursors are used to query object stores for particular pieces of data, and indexes may be created for faster lookups on particular properties.

With all of these options available, it's possible to store a significant amount of data on the client machine using JavaScript. You should use care not to store sensitive information, because the data cache isn't encrypted.

24

Best Practices

WHAT'S IN THIS CHAPTER?

- Writing maintainable code
- Ensuring code performance
- Deploying code to production

The discipline of web development has grown at an extraordinary rate since 2000. What used to be a virtual Wild West, where just about anything was acceptable, has evolved into a complete discipline with research and established best practices. As simple websites grew into more complex web applications, and web hobbyists became paid professionals, the world of web development was filled with information about the latest techniques and development approaches. JavaScript, in particular, was the beneficiary of a lot of research and conjecture. Best practices for JavaScript fall into several categories and are handled at different points in the development process.

MAINTAINABILITY

In early websites, JavaScript was used primarily for small effects or form validation. Today's web applications are filled with thousands of lines of JavaScript executing all types of complicated processes. This evolution requires that developers take maintainability into account. As with software engineers in more traditional disciplines, JavaScript developers are hired to create value for their company, and they do that not just by delivering products on time but also by developing intellectual property that continues to add value long after.

Writing maintainable code is important, because most developers spend a large amount of their time maintaining other people's code. It's a truly rare occurrence to be able to develop new code from scratch; it's often the case that you must build on work that someone else has done. Making sure that your code is maintainable ensures that other developers can perform their jobs as well as possible.



Note that the concept of maintainable code is not unique to JavaScript. Some of these concepts apply broadly to any programming language, although there are some JavaScript-specific concepts as well.

What Is Maintainable Code?

Maintainable code has several characteristics. In general, code is said to be maintainable when it is all of the following:

- **Understandable** — Someone else can pick up the code and figure out its purpose and general approach without a walk-through by the original developer.
- **Intuitive** — Things in the code just seem to make sense, no matter how complex the operation.
- **Adaptable** — The code is written in such a way that variances in data don't require a complete rewrite.
- **Extendable** — Care has been given in the code architecture to allow extension of the core functionality in the future.
- **Debuggable** — When something goes wrong, the code gives you enough information to identify the issue as directly as possible.

Being able to write maintainable JavaScript code is an important skill for professionals. This is the difference between hobbyists who hack together a site over the weekend and professional developers who really know their craft.

Code Conventions

One of the simplest ways to start writing maintainable code is to come up with code conventions for the JavaScript that you write. Code conventions have been developed for most programming languages, and a quick Internet search is likely to turn up thousands of documents. Professional organizations have long instituted code conventions for developers in an attempt to make code more maintainable for everyone. The best-run open-source projects have strict code convention requirements that allow everyone in the community to easily understand how code is organized.

Code conventions are important for JavaScript because of the language's adaptability. Unlike most object-oriented languages, JavaScript doesn't force developers into defining everything as objects. The language can support any number of programming styles, from traditional object-oriented approaches to declarative approaches to functional approaches. A quick review of several open-source JavaScript libraries can easily yield multiple approaches to creating objects, defining methods, and managing the environment.

The following sections discuss the generalities of how to develop code conventions. These topics are important to address, although the way in which they are addressed may differ, depending on your individual needs.

Readability

For code to be maintainable, it must first be readable. Readability has to do with the way the code is formatted as a text file. A large part of readability has to do with the indentation of the code. When everyone is using the same indentation scheme, code across an entire project becomes much easier to read. Indentation is usually done by using a number of spaces instead of by using the tab character, which is typically displayed differently by different text editors. A good general indentation size is four spaces, although you may decide to use less or more.

Another part of readability is comments. In most programming languages, it's an accepted practice to comment each method. Because of JavaScript's ability to create functions at any point in the code, this is often overlooked. Because of this, it is perhaps even more important to document each function in JavaScript. Generally speaking, the places that should be commented in your code are as follows:

- **Functions and methods** — Each function or method should include a comment that describes its purpose and possibly the algorithm being used to accomplish the task. It's also important to state assumptions that are being made, what the arguments represent, and whether or not the function returns a value (since this is not discernible from a function definition).
- **Large sections of code** — Multiple lines of code that are all used to accomplish a single task should be preceded with a comment describing the task.
- **Complex algorithms** — If you're using a unique approach to solve a problem, explain how you are doing it as a comment. This will not only help others who are looking at your code but also help you the next time you look at it.
- **Hacks** — Because of browser differences, JavaScript code typically contains some hacks. Don't assume that someone else who is looking at the code will understand the browser issue that such a hack is working around. If you need to do something differently because one of the browsers can't use the normal way, put that in a comment. It reduces the likelihood that someone will come along, see your hack, and "fix" it, inadvertently introducing the bug that you had already worked around.

Indentation and comments create more readable code that is easier to maintain in the future.

Variable and Function Naming

The proper naming of variables and functions in code is vital to making it understandable and maintainable. Since many JavaScript developers began as hobbyists, there's a tendency to use nonsensical names such as "foo" and "bar" for variables and names such as "doSomething" for functions. A professional JavaScript developer must overcome these old habits to create maintainable code. General rules for naming are as follows:

- Variable names should be nouns, such as "car" or "person".
- Function names should begin with a verb, such as `getName()`. Functions that return Boolean values typically begin with `is`, as in `isEnabled()`.
- Use logical names for both variables and functions, without worrying about the length. Length can be mitigated through postprocessing and compression (discussed later in this chapter).

It's imperative to avoid useless variable names that don't indicate the type of data they contain. With proper naming, code reads like a narrative of what is happening, making it easier to understand.

Variable Type Transparency

Since variables are loosely typed in JavaScript, it is easy to lose track of the type of data that a variable should contain. Proper naming mitigates this to some point, but it may not be enough in all cases. There are three ways to indicate the data type of a variable.

The first way is through initialization. When a variable is defined, it should be initialized to a value that indicates how it will be used in the future. For example, a variable that will hold a Boolean should be initialized to either `true` or `false`, and a variable to hold numbers should be initialized to a number, as in the following example:

```
//variable type indicated by initialization
var found = false;           //Boolean
var count = -1;              //number
var name = "";                //string
var person = null;            //object
```

Initialization to a particular data type is a good indication of a variable's type. The downside of initialization is that it cannot be used with function arguments in the function declaration.

The second way to indicate a variable's type is to use Hungarian notation. Hungarian notation prepends one or more characters to the beginning of a variable to indicate the data type. This notation is popular among scripted languages and was, for quite some time, the preferred format for JavaScript as well. The most traditional Hungarian notation format for JavaScript prepends a single character for the basic data types: "`o`" for objects, "`s`" for strings, "`i`" for integers, "`f`" for floats, and "`b`" for Booleans. Here's an example:

```
//Hungarian notation used to indicate data type
var bFound;      //Boolean
var iCount;      //integer
var sName;        //string
var oPerson;      //object
```

Hungarian notation for JavaScript is advantageous in that it can be used equally well for function arguments. The downside of Hungarian notation is that it makes code somewhat less readable, interrupting the intuitive, sentence-like nature of code that is accomplished without it. For this reason, Hungarian notation has started to fall out of favor among some developers.

The last way to indicate variable type is to use type comments. Type comments are placed right after the variable name but before any initialization. The idea is to place a comment indicating the data type right by the variable, as in this example:

```
//type comments used to indicate type
var found /*:Boolean*/ = false;
var count /*:int*/ = 10;
var name /*:String*/ = "Nicholas";
var person /*:Object*/ = null;
```

Type comments maintain the overall readability of code while injecting type information at the same time. The downside of type comments is that you cannot comment out large blocks of code using multiline comments, because the type comments are also multiline comments that will interfere, as this example demonstrates:

```
//The following won't work correctly
/*
var found /*:Boolean*/ = false;
var count /*:int*/ = 10;
var name /*:String*/ = "Nicholas";
var person /*:Object*/ = null;
*/
```

Here, the intent was to comment out all of the variables using a multiline comment. The type comments interfere with this because the first instance of /* (second line) is matched with the first instance of */ (third line), which will cause a syntax error. If you want to comment out lines of code using type comments, it's best to use single-line comments on each line (many editors will do this for you).

These are the three most common ways to indicate the data type of variables. Each has advantages and disadvantages for you to evaluate before deciding on one. The important thing is to decide which works best for your project and use it consistently.

Loose Coupling

Whenever parts of an application depend too closely on one another, the code becomes too tightly coupled and hard to maintain. The typical problem arises when objects refer directly to one another in such a way that a change to one always requires a change to the other. Tightly coupled software is difficult to maintain and invariably has to be rewritten frequently.

Because of the technologies involved, there are several ways in which web applications can become too tightly coupled. It's important to be aware of this and to try to maintain loosely coupled code whenever possible.

Decouple HTML/JavaScript

One of the most common types of coupling is HTML/JavaScript coupling. On the Web, HTML and JavaScript each represent a different layer of the solution: HTML is the data, and JavaScript is the behavior. Because they are intended to interact, there are a number of different ways to tie these two technologies together. Unfortunately, there are some ways that too tightly couple HTML and JavaScript.

JavaScript that appears inline in HTML, either using a `<script>` element with inline code or using HTML attributes to assign event handlers, is too tightly coupled. Consider the following code examples:

```
<!-- tightly coupled HTML/JavaScript using <script> -->
<script type="text/javascript">
  document.write("Hello world!");
</script>

<!-- tightly coupled HTML/JavaScript using event handler attribute -->
<input type="button" value="Click Me" onclick="doSomething()" />
```

Although these are both technically correct, in practice they tightly couple the HTML representing the data with the JavaScript that defines the behavior. Ideally, HTML and JavaScript should be completely separate, with the JavaScript being included via external files and attaching behavior using the DOM.

When HTML and JavaScript are too tightly coupled, a JavaScript error means first determining whether the error occurred in the HTML portion of the solution or in a JavaScript file. It also introduces new types of errors related to the availability of code. In this example, the button may be clicked before the `doSomething()` function is available, causing a JavaScript error. Maintainability is affected because any change to the button's behavior requires touching both the HTML and the JavaScript, when it should require only the latter.

HTML and JavaScript can also be too tightly coupled when the reverse is true: HTML is contained within JavaScript. This usually occurs when using `innerHTML` to insert a chunk of HTML text into the page, as in this example:

```
//tight coupling of HTML to JavaScript
function insertMessage(msg) {
    var container = document.getElementById("container");
    container.innerHTML = "<div class=\"msg\"><p class=\"post\">" + msg + "</p>" +
        "<p><em>Latest message above.</em></p></div>";
}
```

Generally speaking, you should avoid creating large amounts of HTML in JavaScript. This, once again, has to do with keeping the layers separate and being able to easily identify the source of errors. When using this example code, a problem with page layout may be related to dynamically created HTML that is improperly formatted. However, locating the error may be difficult, because you would typically first view the source of the page to look for the offending HTML but wouldn't find it there because it's dynamically generated. Changes to the data or layout would also require changes to the JavaScript, which indicates that the two layers are too tightly coupled.

HTML rendering should be kept separate from JavaScript as much as possible. When JavaScript is used to insert data, it should do so without inserting markup whenever possible. Markup can typically be included and hidden when the entire page is rendered such that JavaScript can be used to display the markup later, instead of generating it. Another approach is to make an Ajax request to retrieve additional HTML to be displayed; this approach allows the same rendering layer (PHP, JSP, Ruby, and so on) to output the markup, instead of embedding it in JavaScript.

Decoupling HTML and JavaScript can save time during debugging, by making it easier to identify the source of errors, and it also eases maintainability: changes to behavior occur only in JavaScript files, whereas changes to markup occur only in rendering files.

Decouple CSS/JavaScript

Another layer of the web tier is CSS, which is primarily responsible for the display of a page. JavaScript and CSS are closely related: they are both layers on top of HTML and as such are often used together. As with HTML and JavaScript, however, it's possible for CSS and JavaScript to be too tightly coupled. The most common example of tight coupling is using JavaScript to change individual styles, as shown here:

```
//tight coupling of CSS to JavaScript
element.style.color = "red";
element.style.backgroundColor = "blue";
```

Since CSS is responsible for the display of a page, any trouble with the display should be addressable by looking just at CSS files. However, when JavaScript is used to change individual styles, such as color, it adds a second location that must be checked and possibly changed. The result is that JavaScript is somewhat responsible for the display of the page and a tight coupling with CSS. If the styles need to change in the future, both the CSS and the JavaScript files may require changes. This creates a maintenance nightmare for developers. A cleaner separation between the layers is needed.

Modern web applications use JavaScript to change styles frequently, so although it's not possible to completely decouple CSS and JavaScript, the coupling can be made looser. This is done by dynamically changing classes instead of individual styles, as in the following example:

```
//loose coupling of CSS to JavaScript
element.className = "edit";
```

By changing only the CSS class of an element, you allow most of the style information to remain strictly in the CSS. JavaScript can be used to change the class, but it's not directly affecting the style of the element. As long as the correct class is applied, then any display issues can be tracked directly to CSS and not to JavaScript.

The second type of tight coupling is valid only in Internet Explorer (but not in Internet Explorer 8+ running in standards mode), where it's possible to embed JavaScript in CSS via expressions, as in this example:

```
/* tight coupling of JavaScript to CSS */
div {
    width: expression(document.body.offsetWidth - 10 + "px");
}
```

Expressions are typically avoided because they're not cross-browser-compatible. They should also be avoided because of the tight coupling between JavaScript and CSS that they introduce. When you use expressions, it's possible that a JavaScript error can occur in CSS. Developers who have tried to track down a JavaScript error due to CSS expressions can tell you how long it took before they even considered looking at the CSS for the source of the error.

Once again, the importance of keeping a good separation of layers is paramount. The only source for display issues should be CSS, and the only source for behavior issues should be JavaScript. Keeping a loose coupling between these layers makes your entire application more maintainable.

Decouple Application Logic/Event Handlers

Every web application is typically filled with lots of event handlers listening for numerous different events. Few of them, however, take care to separate application logic from event handlers. Consider the following example:

```
function handleKeyPress(event) {
    event = EventUtil.getEvent(event);
    if (event.keyCode == 13) {
```

```
var target = EventUtil.getTarget(event);
var value = 5 * parseInt(target.value);
if (value > 10){
    document.getElementById("error-msg").style.display = "block";
}
}
}
```

This event handler contains application logic in addition to handling the event. The problem with this approach is twofold. First, there is no way to cause the application logic to occur other than through the event, which makes it difficult to debug. What if the anticipated result didn't occur? Does that mean that the event handler wasn't called or that the application logic failed? Second, if a subsequent event causes the same application logic to occur, you'll need to duplicate the functionality or else extract it into a separate function. Either way, it requires more changes to be made than are really necessary.

A better approach is to separate the application logic from event handlers, so that each handles just what it's supposed to. An event handler should interrogate the `event` object for relevant information and then pass that information to some method that handles the application logic. For example, the previous code can be rewritten like this:

```
function validateValue(value){
    value = 5 * parseInt(value);
    if (value > 10){
        document.getElementById("error-msg").style.display = "block";
    }
}

function handleKeyPress(event){
    event = EventUtil.getEvent(event);
    if (event.keyCode == 13){
        var target = EventUtil.getTarget(event);
        validateValue(target.value);
    }
}
```

This updated code properly separates the application logic from the event handler. The `handleKeyPress()` function checks to be sure that the Enter key was pressed (`event.keyCode` is 13), and then gets the target of the event and passes the `value` property into the `validateValue()` function, which contains the application logic. Note that there is nothing in `validateValue()` that depends on any event handler logic whatsoever; it just receives a value and can do everything else based on that value.

Separating application logic from event handlers has several benefits. First, it allows you to easily change the events that trigger certain processes with a minimal amount of effort. If a mouse click initially caused the processing to occur, but now a key press should do the same, it's quite easy to make that change. Second, you can test code without attaching events, making it easier to create unit tests or to automate application flow.

Here are a few rules to keep in mind for loose coupling of application and business logic:

- Don't pass the `event` object into other methods; pass only the data from the `event` object that you need.
- Every action that is possible in the application should be possible without executing an event handler.
- Event handlers should process the event and then hand off processing to application logic.

Keeping this approach in mind is a huge maintainability win in any code base, opening up numerous possibilities for testing and further development.

Programming Practices

Writing maintainable JavaScript isn't just about how the code is formatted; it's also about what the code does. Web applications created in an enterprise environment are often worked on by numerous people at the same time. The goal in these situations is to ensure that the browser environment in which everyone is working has constant and unchanging rules. To achieve this, developers should adhere to certain programming practices.

Respect Object Ownership

The dynamic nature of JavaScript means that almost anything can be modified at any point in time. It's been said that nothing in JavaScript is sacred, as you're unable to mark something as final or constant. This changed somewhat with ECMAScript 5's introduction of tamper-proof objects (discussed in Chapter 22), but by default, all objects can be modified. In other languages, objects and classes are immutable when you don't have the actual source code. JavaScript allows you to modify any object at any time, making it possible to override default behaviors in unanticipated ways. Because the language doesn't impose limits, it's important and necessary for developers to do so.

Perhaps the most important programming practice in an enterprise environment is to respect object ownership, which means that you don't modify objects that don't belong to you. Put simply: if you're not responsible for the creation or maintenance of an object, its constructor, or its methods, you shouldn't be making changes to it. More specifically:

- Don't add properties to instances or prototypes.
- Don't add methods to instances or prototypes.
- Don't redefine existing methods.

The problem is that developers assume that the browser environment works in a certain way. Changes to objects that are used by multiple people mean that errors will occur. If someone expects a function called `stopEvent()` to cancel the default behavior for an event, and you change it so it does that and also attaches other event handlers, it is certain that problems will follow. Other developers are assuming that the function just does what it did originally, so their usage will be incorrect and possibly harmful, because they don't know the side effects.

These rules apply not only to custom types and objects but also to native types and objects such as `Object`, `String`, `document`, `window`, and so on. The potential issues here are even more perilous because browser vendors may change these objects in unannounced and unanticipated ways.

An example of this occurred in the popular Prototype JavaScript library, which implemented the `getElementsByClassName()` method on the `document` object, returning an instance of `Array` that had also been augmented to include a method called `each()`. John Resig outlined on his blog the sequence of events that caused the issue. In his post (<http://ejohn.org/blog/getelementsbyclassname-pre-prototype-16/>), he noted that the problem occurred when browsers began to natively implement `getElementsByClassName()`, which returns not an `Array` but rather a `NodeList` that doesn't have an `each()` method. Developers using the Prototype library had gotten used to writing code such as this:

```
document.getElementsByClassName("selected").each(Element.hide);
```

Although this code worked fine in browsers that didn't implement `getElementsByClassName()` natively, it caused an error in the ones that did, as a result of the return value differences. You cannot anticipate how browser vendors will change native objects in the future, so modifying them in any way can lead to issues down the road when your implementation clashes with theirs.

The best approach, therefore, is to never modify objects you don't own. You own an object only when you created it yourself, such as a custom type or object literal. You don't own `Array`, `document`, or so on, because they were there before your code executed. You can still create new functionality for objects by doing the following:

- Create a new object with the functionality you need, and let it interact with the object of interest.
- Create a custom type that inherits from the type you want to modify. You can then modify the custom type with the additional functionality.

Many JavaScript libraries now subscribe to this theory of development, allowing them to grow and adapt even as browsers continually change.

Avoid Globals

Closely related to respecting object ownership is avoiding global variables and functions whenever possible. Once again, this has to do with creating a consistent and maintainable environment in which scripts will be executed. At most, a single global variable should be created on which other objects and functions exist. Consider the following:

```
//two globals - AVOID!!!
var name = "Nicholas";
function sayName(){
    alert(name);
}
```

This code contains two globals: the variable `name` and the function `sayName()`. These can easily be created on an object that contains both, as in this example:

```
//one global - preferred
var MyApplication = {
    name: "Nicholas",
    sayName: function(){
        alert(this.name);
    }
};
```

This rewritten version of the code introduces a single global object, `MyApplication`, onto which both `name` and `sayName()` are attached. Doing so clears up a couple of issues that existed in the previous code. First, the variable `name` overwrites the `window.name` property, which possibly interferes with other functionality. Second, it helps to clear up confusion over where the functionality lives. Calling `MyApplication.sayName()` is a logical hint that any issues with the code can be identified by looking at the code in which `MyApplication` is defined.

An extension of the single global approach is the concept of *namespacing*, popularized by the Yahoo! User Interface (YUI) library. Namespacing involves creating an object to hold functionality. In the 2.x version of YUI, there were several namespaces onto which functionality was attached. Here are some examples:

- `YAHOO.util.Dom` — Methods for manipulating the DOM.
- `YAHOO.util.Event` — Methods for interacting with events.
- `YAHOO.lang` — Methods for helping with low-level language features.

For YUI, the single global object `YAHOO` serves as a container onto which other objects are defined. Whenever objects are used simply to group together functionality in this manner, they are called *namespaces*. The entire YUI library is built on this concept, allowing it to coexist on the same page with any other JavaScript library.

The important part of namespacing is to decide on a global object name that everyone agrees to use and that is unique enough that others aren't likely to use it as well. In most cases, this can be the name of the company for which you're developing the code, such as `YAHOO` or `Wrox`. You can then start creating namespaces to group your functionality, as in this example:

```
//create global object
var Wrox = {};  
  
//create namespace for Professional JavaScript
Wrox.ProJS = {};  
  
//attach other objects used in the book
Wrox.ProJS.EventUtil = { ... };
Wrox.ProJS.CookieUtil = { ... };
```

In this example, `Wrox` is the global on which namespaces are created. If all code for this book is placed under the `Wrox.ProJS` namespace, it leaves other authors to add their code onto the `wrox` object as well. As long as everyone follows this pattern, there's no reason to be worried that someone else will also write an object called `EventUtil` or `CookieUtil`, because it will exist on a different namespace. Consider this example:

```
//create namespace for Professional Ajax
Wrox.ProAjax = {};  
  
//attach other objects used in the book
Wrox.ProAjax.EventUtil = { ... };
Wrox.ProAjax.CookieUtil = { ... };
```

```
//you can still access the ProJS one
Wrox.ProJS.EventUtil.addHandler( ... );

//and the ProAjax one separately
Wrox.ProAjax.EventUtil.addHandler( ... );
```

Although namespacing requires a little more code, it is worth the trade-off for maintainability purposes. Namespacing helps ensure that your code can work on a page with other code in a nonharmful way.

Avoid Null Comparisons

Since JavaScript doesn't do any automatic type checking, it becomes the developer's responsibility. As a result, very little type checking actually gets done in JavaScript code. The most common type check is to see if a value is `null`. Unfortunately, checking a value against `null` is overused and frequently leads to errors due to insufficient type checking. Consider the following:

```
function sortArray(values){
    if (values != null){           //AVOID!!
        values.sort(comparator);
    }
}
```

The purpose of this function is to sort an array with a given comparator. The `values` argument must be an array for the function to execute correctly, but the `if` statement simply checks to see that `values` isn't `null`. There are several values that can make it past the `if` statement, including any string or any number, which would then cause the function to throw an error.

Realistically, `null` comparisons are rarely good enough to be used. Values should be checked for what they are expected to be, not for what they aren't expected to be. For example, in the previous code, the `values` argument is expected to be an array, so you should be checking to see if it is an array, rather than checking to see if it's not `null`. The function can be rewritten more appropriately as follows:

```
function sortArray(values){
    if (values instanceof Array){ //preferred
        values.sort(comparator);
    }
}
```

This version of the function protects against all invalid values and doesn't need to use `null` at all.



This technique for identifying an array doesn't work properly in a multiframe web page, because each frame has its own global object and, therefore, its own Array constructor. If you are passing arrays from one frame to another, you may want to test for the existence of the sort() method instead.

If you see a `null` comparison in code, try replacing it using one of the following techniques:

- If the value should be a reference type, use the `instanceof` operator to check its constructor.
- If the value should be a primitive type, use the `typeof` operator to check its type.
- If you're expecting an object with a specific method name, use the `typeof` operator to ensure that a method with the given name exists on the object.

The fewer `null` comparisons in code, the easier it is to determine the purpose of the code and to eliminate unnecessary errors.

Use Constants

Even though JavaScript doesn't have a formal concept of constants, they are still useful. The idea is to isolate data from application logic in such a way that it can be changed without risking the introduction of errors. Consider the following:

```
function validate(value){
    if (!value){
        alert("Invalid value!");
        location.href = "/errors/invalid.php";
    }
}
```

There are two pieces of data in this function: the message displayed to the user and the URL. Strings that are displayed in the user interface should always be extracted in such a way as to allow for internationalization. URLs should also be extracted, because they have a tendency to change as an application grows. Basically, each of these has a possibility of changing for one reason or another, and a change would mean going into the function and changing code there. Any time you're changing application logic code, you open up the possibility of creating errors. You can insulate application logic from data changes by extracting data into constants that are defined separately. Consider the following example:

```
var Constants = {
    INVALID_VALUE_MSG: "Invalid value!",
    INVALID_VALUE_URL: "/errors/invalid.php"
};

function validate(value){
    if (!value){
        alert(Constants.INVALID_VALUE_MSG);
        location.href = Constants.INVALID_VALUE_URL;
    }
}
```

In this rewritten version of the code, both the message and the URL have been defined on a `Constants` object; the function then references these values. This setup allows the data to change without your ever needing to touch the function that uses it. The `Constants` object could even be defined in a completely separate file, and that file could be generated by some process that includes the correct values based on internationalization settings.

The key is to separate data from the logic that uses it. The types of values to look for are as follows:

- **Repeated values** — Any values that are used in more than one place should be extracted into a constant. This limits the chance of errors when one value is changed but others are not. This includes CSS class names.
- **User interface strings** — Any strings that are to be displayed to the user should be extracted for easier internationalization.
- **URLs** — Resource locations tend to change frequently in web applications, so having a common place to store all URLs is recommended.
- **Any value that may change** — Any time you're using a literal value in code, ask yourself if this value might change in the future. If the answer is yes, then the value should be extracted into a constant.

Using constants is an important technique for enterprise JavaScript development, because it makes code more maintainable and keeps it safe from data changes.

PERFORMANCE

The amount of JavaScript developers now write per web page has grown dramatically since the language was first introduced. With that increase came concerns over the runtime execution of JavaScript code. JavaScript was originally an interpreted language, so the speed of execution was significantly slower than it was for compiled languages. Chrome was the first browser to introduce an optimizing engine that compiles JavaScript into native code. Since then, all other major browsers have followed suit and have implemented JavaScript compilation.

Even with the move to compiled JavaScript, it's still possible to write slow code. However, there are some basic patterns that, when followed, ensure the fastest possible execution of code.

Be Scope-Aware

Chapter 4 discussed the concept of scopes in JavaScript and how the scope chain works. As the number of scopes in the scope chain increases, so does the amount of time it takes to access variables outside of the current scope. It is always slower to access a global variable than it is to access a local variable, because the scope chain must be traversed. Anything you can do to decrease the amount of time spent traversing the scope chain will increase overall script performance.

Avoid Global Lookups

Perhaps the most important thing you can do to improve the performance of your scripts is to be wary of global lookups. Global variables and functions are always more expensive to use than local ones because they involve a scope chain lookup. Consider the following function:

```
function updateUI() {
    var imgs = document.getElementsByTagName("img");
    for (var i=0, len=imgs.length; i < len; i++) {
```

```

        imgs[i].title = document.title + " image " + i;
    }

    var msg = document.getElementById("msg");
    msg.innerHTML = "Update complete.";
}

```

This function may look perfectly fine, but it has three references to the global `document` object. If there are multiple images on the page, the `document` reference in the `for` loop could get executed dozens or hundreds of times, each time requiring a scope chain lookup. By creating a local variable that points to the `document` object, you can increase the performance of this function by limiting the number of global lookups to just one:

```

function updateUI(){
    var doc = document;
    var imgs = doc.getElementsByTagName("img");
    for (var i=0, len=imgs.length; i < len; i++){
        imgs[i].title = doc.title + " image " + i;
    }

    var msg = doc.getElementById("msg");
    msg.innerHTML = "Update complete.";
}

```

Here, the `document` object is first stored in the local `doc` variable. The `doc` variable is then used in place of `document` throughout the rest of the code. There's only one global lookup in this function, compared to the previous version, ensuring that it will run faster.

A good rule of thumb is to store any global object that is used more than once in a function as a local variable.

Avoid the `with` Statement

The `with` statement should be avoided where performance is important. Similar to functions, the `with` statement creates its own scope and therefore increases the length of the scope chain for code executed within it. Code executed within a `with` statement is guaranteed to run slower than code executing outside, because of the extra steps in the scope chain lookup.

It is rare that the `with` statement is required, because it is mostly used to eliminate extra characters. In most cases, a local variable can be used to accomplish the same thing without introducing a new scope. Here is an example:

```

function updateBody(){
    with(document.body){
        alert(tagName);
        innerHTML = "Hello world!";
    }
}

```

The `with` statement in this code enables you to use `document.body` more easily. The same effect can be achieved by using a local variable, as follows:

```
function updateBody() {
    var body = document.body;
    alert(body.tagName);
    body.innerHTML = "Hello world!";
}
```

Although this code is slightly longer, it reads better than the `with` statement, ensuring that you know the object to which `tagName` and `innerHTML` belong. This code also saves global lookups by storing `document.body` in a local variable.

Choose the Right Approach

As with other languages, part of the performance equation has to do with the algorithm or approach used to solve the problem. Skilled developers know from experience which approaches are likely to achieve better performance results. Many of the techniques and approaches that are typically used in other programming languages can also be used in JavaScript.

Avoid Unnecessary Property Lookup

In computer science, the complexity of algorithms is represented using O notation. The simplest, and fastest, algorithm is a constant value or $O(1)$. After that, the algorithms just get more complex and take longer to execute. The following table lists the common types of algorithms found in JavaScript.

NOTATION	NAME	DESCRIPTION
$O(1)$	Constant	Amount of time to execute remains constant no matter the number of values. Represents simple values and values stored in variables.
$O(\log n)$	Logarithmic	Amount of time to execute is related to the number of values, but each value need not be retrieved for the algorithm to complete. Example: binary search.
$O(n)$	Linear	Amount of time to execute is directly related to the number of values. Example: iterating over all items in an array.
$O(n^2)$	Quadratic	Amount of time to execute is related to the number of values such that each value must be retrieved at least n times. Example: insertion sort.

Constant values, or $O(1)$, refer to both literals and values that are stored in variables. The notation $O(1)$ indicates that the amount of time necessary to retrieve a constant value remains the same regardless of the number of values. Retrieving a constant value is an extremely efficient process and so is quite fast. Consider the following:

```
var value = 5;
var sum = 10 + value;
alert(sum);
```

This code performs four constant value lookups: the number 5, the variable `value`, the number 10, and the variable `sum`. The overall complexity of this code is then considered to be O(1).

Accessing array items is also an O(1) operation in JavaScript, performing just as well as a simple variable lookup. So the following code is just as efficient as the previous example:

```
var values = [5, 10];
var sum = values[0] + values[1];
alert(sum);
```

Using variables and arrays is more efficient than accessing properties on objects, which is an O(n) operation. Every property lookup on an object takes longer than accessing a variable or array, because a search must be done for a property of that name up the prototype chain. Put simply, the more property lookups there are, the slower the execution time. Consider the following:

```
var values = { first: 5, second: 10};
var sum = values.first + values.second;
alert(sum);
```

This code uses two property lookups to calculate the value of `sum`. Doing one or two property lookups may not result in significant performance issues, but doing hundreds or thousands will definitely slow down execution.

Be wary of multiple property lookups to retrieve a single value. For example, consider the following:

```
var query = window.location.href.substring(window.location.href.indexOf("?"));
```

In this code, there are six property lookups: three for `window.location.href.substring()` and three for `window.location.href.indexOf()`. You can easily identify property lookups by counting the number of dots in the code. This code is especially inefficient because the `window.location.href` value is being used twice, so the same lookup is done twice.

Whenever an object property is being used more than once, store it in a local variable. You'll still take the initial O(n) hit to access the value the first time, but every subsequent access will be O(1), which more than makes up for it. For example, the previous code can be rewritten as follows:

```
var url = window.location.href;
var query = url.substring(url.indexOf("?"));
```

This version of the code has only four property lookups, a savings of 33 percent over the original. Making this kind of optimization in a large script is likely to lead to larger gains.

Generally speaking, any time you can decrease the complexity of an algorithm, you should. Replace as many property lookups as possible by using local variables to store the values. Furthermore, if you have an option to access something as a numeric array position or a named property (such as with `NodeList` objects), use the numeric position.

Optimize Loops

Loops are one of the most common constructs in programming and, as such, are found frequently in JavaScript. Optimizing these loops is an important part of the performance optimization process,

since they run the same code repeatedly, automatically increasing execution time. There's been a great deal of research done into loop optimization for other languages, and these techniques also apply to JavaScript. The basic optimization steps for a loop are as follows:

- 1. Decrement iterators** — Most loops are created with an iterator that starts at 0 and is incremented up to a certain value. In many cases, it's more efficient to start the iterator at the maximum number and decrement each time through the loop.
- 2. Simplify the terminal condition** — Since the terminal condition is evaluated each time through the loop, it should be as fast as possible. This means avoiding property lookups or other $O(n)$ operations.
- 3. Simplify the loop body** — The body of the loop is executed the most, so make sure it's as optimized as possible. Make sure there's no intensive computation being performed that could easily be moved to outside the loop.
- 4. Use posttest loops** — The most commonly used loops are `for` and `while`, both of which are pretest loops. Posttest loops, such as `do-while`, avoid the initial evaluation of the terminal condition and tend to run faster.

These changes are best illustrated with an example. The following is a basic `for` loop:

```
for (var i=0; i < values.length; i++) {
    process(values[i]);
}
```

This code increments the variable `i` from 0 up to the total number of items in the `values` array. Assuming that the order in which the values are processed is irrelevant, the loop can be changed to decrement `i` instead, as follows:

```
for (var i=values.length-1; i >= 0; i--) {
    process(values[i]);
}
```

Here, the variable `i` is decremented each time through the loop. In the process, the terminal condition is simplified by removing the $O(n)$ call to `values.length` and replacing it with the $O(1)$ call of 0. Since the loop body has only a single statement, it can't be optimized further. However, the loop itself can be changed into a posttest loop like this:

```
var i=values.length-1;
if (i > -1){
    do {
        process(values[i]);
    }while(--i >= 0);
}
```

The primary optimization here is combining the terminal condition and the decrement operator into a single statement. At this point, any further optimization would have to be done to the `process()` function itself because the loop is fully optimized.

Keep in mind that using a posttest loop works only when you're certain that there will always be at least one value to process. An empty array causes an unnecessary trip through the loop that a pre-test loop would otherwise avoid.

Unrolling Loops

When the number of times through a loop is finite, it is often faster to eliminate the loop altogether and replace it with multiple function calls. Consider the loop from the previous example. If the length of the array will always be the same, it may be more optimal to simply call `process()` on each item, as in the following code:

```
//eliminated the loop
process(values[0]);
process(values[1]);
process(values[2]);
```

This example assumes that there are only three items in the `values` array and simply calls `process()` directly on each item. Unrolling loops in this way eliminates the overhead of setting up a loop and processing a terminal condition, making the code run faster.

If the number of iterations through the loop can't be determined ahead of time, you may want to consider using a technique called *Duff's device*. The technique is named after its creator, Tom Duff, who first proposed using it in the C programming language. Jeff Greenberg is credited with implementing Duff's device in JavaScript. The basic idea of Duff's device is to unroll a loop into a series of statements by calculating the number of iterations as a multiple of 8. Consider the following code example:

```
//credit: Jeff Greenberg for JS implementation of Duff's Device
//assumes values.length > 0
var iterations = Math.ceil(values.length / 8);
var startAt = values.length % 8;
var i = 0;

do {
    switch(startAt) {
        case 0: process(values[i++]);
        case 7: process(values[i++]);
        case 6: process(values[i++]);
        case 5: process(values[i++]);
        case 4: process(values[i++]);
        case 3: process(values[i++]);
        case 2: process(values[i++]);
        case 1: process(values[i++]);
    }
    startAt = 0;
} while (--iterations > 0);
```

This implementation of Duff's device starts by calculating how many iterations through the loop need to take place by dividing the total number of items in the `values` array by 8. The ceiling function is then used to ensure that the result is a whole number. The `startAt` variable holds the number of items that wouldn't be processed if the iterations were based solely on dividing by 8. When the loop executes for the first time, the `startAt` variable is checked to see how many extra

calls should be made. For instance, if there are 10 values in the array, `startAt` would be equal to 2, so `process()` would be called only twice the first time through the loop. At the bottom of the loop, `startAt` is reset to 0 so that each subsequent time through the loop results in eight calls to `process()`. This unrolling speeds up processing of large data sets.

The book *Speed Up Your Site* by Andrew B. King (New Riders, 2003) proposed an even faster Duff's device technique that separated the `do-while` loop into two separate loops. Here's an example:

```
//credit: Speed Up Your Site (New Riders, 2003)
var iterations = Math.floor(values.length / 8);
var leftover = values.length % 8;
var i = 0;

if (leftover > 0){
    do {
        process(values[i++]);
    } while (--leftover > 0);
}

do {
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
} while (--iterations > 0);
```

In this implementation, the leftover count that wouldn't have been handled in the loop when simply dividing by 8 is handled in an initial loop. Once those extra items are processed, execution continues in the main loop that calls `process()` eight times. This approach is almost 40 percent faster than the original Duff's device implementation.

Unrolling loops can yield big savings for large data sets but may not be worth the extra effort for small data sets. The trade-off is that it takes more code to accomplish the same task, which is typically not worth it when large data sets aren't being processed.

Avoid Double Interpretation

Double interpretation penalties exist when JavaScript code tries to interpret JavaScript code. This situation arises when using the `eval()` function or the `Function` constructor or when using `setTimeout()` with a string argument. Here are some examples:

```
//evaluate some code - AVOID!!
eval("alert('Hello world!')");

//create a new function - AVOID!!
var sayHi = new Function("alert('Hello world!')");

//set a timeout - AVOID!!
setTimeout("alert('Hello world!')", 500);
```

In each of these instances, a string containing JavaScript code has to be interpreted. This can't be done during the initial parsing phase because the code is contained in a string, which means a new parser has to be started while the JavaScript code is running to parse the new code. Instantiating a new parser has considerable overhead, so the code runs slower than if it were included natively.

There are workarounds for all of these instances. It's rare that `eval()` is absolutely necessary, so try to avoid it whenever possible. In this case, the code could just be included inline. For the `Function` constructor, the code can be rewritten as a regular function quite easily, and the `setTimeout()` call can pass in a function as the first argument. Here are some examples:

```
//fixed
alert('Hello world!');

//create a new function - fixed
var sayHi = function(){
    alert('Hello world!');
};

//set a timeout - fixed
setTimeout(function(){
    alert('Hello world!');
}, 500);
```

To increase the performance of your code, avoid using strings that need to be interpreted as JavaScript whenever possible.

Other Performance Considerations

There are a few other things to consider when evaluating the performance of your script. The following aren't major issues, but they can make a difference when used frequently:

- **Native methods are fast** — Whenever possible, use a native method instead of one written in JavaScript. Native methods are written in compiled languages such as C or C++ and thus run much faster than those in JavaScript. The most often forgotten methods in JavaScript are the complex mathematical operations available on the `Math` object; these methods always run faster than any JavaScript equivalent for calculating sine, cosine, and so on.
- **Switch statements are fast** — If you have a complex series of `if-else` statements, converting it to a single switch statement can result in faster code. You can further improve the performance of switch statements by organizing the cases in the order of most likely to least likely.
- **Bitwise operators are fast** — When performing mathematical operations, bitwise operations are always faster than any Boolean or numeric arithmetic. Selectively replacing arithmetic operations with bitwise operations can greatly improve the performance of complex calculations. Operations such as modulus, logical AND, and logical OR are good candidates to be replaced with bitwise operations.

Minimize Statement Count

The number of statements in JavaScript code affects the speed with which the operations are performed. A single statement can complete multiple operations faster than multiple statements each

performing a single operation. The task, then, is to seek out statements that can be combined in order to decrease the execution time of the overall script. To do so, you can look for several patterns.

Multiple Variable Declarations

One area in which developers tend to create too many statements is in the declaration of multiple variables. It's quite common to see code declaring multiple variables using multiple `var` statements, such as the following:

```
//four statements - wasteful
var count = 5;
var color = "blue";
var values = [1,2,3];
var now = new Date();
```

In strongly typed languages, variables of different data types must be declared in separate statements. In JavaScript, however, all variables can be declared using a single `var` statement. The preceding code can be rewritten as follows:

```
//one statement
var count = 5,
    color = "blue",
    values = [1,2,3],
    now = new Date();
```

Here, the variable declarations use a single `var` statement and are separated by commas. This is an optimization that is easy to make in most cases and performs much faster than declaring each variable separately.

Insert Iterative Values

Any time you are using an iterative value (that is, a value that is being incremented or decremented at various locations), combine statements whenever possible. Consider the following code snippet:

```
var name = values[i];
i++;
```

Each of the two preceding statements has a single purpose: the first retrieves a value from `values` and stores it in `name`; the second increments the variable `i`. These can be combined into a single statement by inserting the iterative value into the first statement, as shown here:

```
var name = values[i++];
```

This single statement accomplishes the same thing as the previous two statements. Because the increment operator is postfix, the value of `i` isn't incremented until after the rest of the statement executes. Whenever you have a similar situation, try to insert the iterative value into the last statement that uses it.

Use Array and Object Literals

Throughout this book, you've seen two ways of creating arrays and objects: using a constructor or using a literal. Using constructors always leads to more statements than are necessary to insert items or define properties, whereas literals complete all operations in a single statement. Consider the following example:

```
//four statements to create and initialize array - wasteful
var values = new Array();
values[0] = 123;
values[1] = 456;
values[2] = 789;

//four statements to create and initialize object - wasteful
var person = new Object();
person.name = "Nicholas";
person.age = 29;
person.sayName = function(){
    alert(this.name);
};
```

In this code, an array and an object are created and initialized. Each requires four statements: one to call the constructor and three to assign data. These can easily be converted to use literals as follows:

```
//one statement to create and initialize array
var values = [123, 456, 789];

//one statement to create and initialize object
var person = {
    name : "Nicholas",
    age : 29,
    sayName : function(){
        alert(this.name);
    }
};
```

This rewritten code contains only two statements: one to create and initialize the array, and one to create and initialize the object. What previously took eight statements now takes only two, reducing the statement count by 75 percent. The value of these optimizations is even greater in codebases that contain thousands of lines of JavaScript.

Whenever possible, replace your array and object declarations with their literal representation to eliminate unnecessary statements.



There is a slight performance penalty for using literals in Internet Explorer 6 and earlier. These issues are resolved in Internet Explorer 7.

Optimize DOM Interactions

Of all the parts of JavaScript, the DOM is without a doubt the slowest part. DOM manipulations and interactions take a large amount of time because they often require rerendering all or part of the page. Furthermore, seemingly trivial operations can take longer to execute because the DOM manages so much information. Understanding how to optimize interactions with the DOM can greatly increase the speed with which scripts complete.

Minimize Live Updates

Whenever you access part of the DOM that is part of the displayed page, you are performing a *live update*. Live updates are so called because they involve immediate (live) updates of the page's display to the user. Every change, whether it be inserting a single character or removing an entire section, incurs a performance penalty as the browser recalculates thousands of measurements to perform the update. The more live updates you perform, the longer it will take for the code to completely execute. The fewer live updates necessary to complete an operation, the faster the code will be. Consider the following example:

```
var list = document.getElementById("myList"),
    item,
    i;

for (i=0; i < 10; i++) {
    item = document.createElement("li");
    list.appendChild(item);
    item.appendChild(document.createTextNode("Item " + i));
}
```

This code adds 10 items to a list. For each item that is added, there are two live updates: one to add the `` element and another to add the text node to it. Since 10 items are being added, that's a total of 20 live updates to complete this operation.

To fix this performance bottleneck, you need to reduce the number of live updates. There are generally two approaches to this. The first is to remove the list from the page, perform the updates, and then reinsert the list into the same position. This approach is not ideal because it can cause unnecessary flickering as the page updates each time. The second approach is to use a document fragment to build up the DOM structure and then add it to the `list` element. This approach avoids live updates and page flickering. Consider the following:

```
var list = document.getElementById("myList"),
    fragment = document.createDocumentFragment(),
    item,
    i

for (i=0; i < 10; i++) {
    item = document.createElement("li");
    fragment.appendChild(item);
    item.appendChild(document.createTextNode("Item " + i));
}

list.appendChild(fragment);
```

There is only one live update in this example, and it occurs after all items have been created. The document fragment is used as a temporary placeholder for the newly created items. All items are then added to the list, using `appendChild()`. Remember, when a document fragment is passed in to `appendChild()`, all of the children of the fragment are appended to the parent, but the fragment itself is never added.

Whenever updates to the DOM are necessary, consider using a document fragment to build up the DOM structure before adding it to the live document.

Use `innerHTML`

There are two ways to create new DOM nodes on the page: using DOM methods such as `createElement()` and `appendChild()`, and using `innerHTML`. For small DOM changes, the two techniques perform roughly the same. For large DOM changes, however, using `innerHTML` is much faster than creating the same DOM structure using standard DOM methods.

When `innerHTML` is set to a value, an HTML parser is created behind the scenes, and the DOM structure is created using the native DOM calls rather than JavaScript-based DOM calls. The native methods execute much faster, since they are compiled rather than interpreted. The previous example can be rewritten to use `innerHTML` like this:

```
var list = document.getElementById("myList"),
    html = "",
    i;

for (i=0; i < 10; i++) {
    html += "<li>Item " + i + "</li>";
}

list.innerHTML = html;
```

This code constructs an HTML string and then assigns it to `list.innerHTML`, which creates the appropriate DOM structure. Although there is always a small performance hit for string concatenation, this technique still performs faster than performing multiple DOM manipulations.

The key to using `innerHTML`, as with other DOM operations, is to minimize the number of times it is called. For instance, the following code uses `innerHTML` too much for this operation:

```
var list = document.getElementById("myList"),
    i;

for (i=0; i < 10; i++) {
    list.innerHTML += "<li>Item " + i + "</li>";      //AVOID!!!
}
```

The problem with this code is that `innerHTML` is called each time through the loop, which is incredibly inefficient. A call to `innerHTML` is, in fact, a live update and should be treated as such. It's far faster to build up a string and call `innerHTML` once than it is to call `innerHTML` multiple times.

Use Event Delegation

Most web applications make extensive use of event handlers for user interaction. There is a direct relationship between the number of event handlers on a page and the speed with which the page responds to user interaction. To mitigate these penalties, you should use event delegation whenever possible.

Event delegation, as discussed in Chapter 13, takes advantage of events that bubble. Any event that bubbles can be handled not just at the event target but also at any of the target's ancestors. Using this knowledge, you can attach event handlers at a high level that are responsible for handling events for multiple targets. Whenever possible, attach an event handler at the document level that can handle events for the entire page.

Beware of `HTMLCollections`

The pitfalls of `HTMLCollection` objects have been discussed throughout this book, because they are a big performance sink for web applications. Keep in mind that any time you access an `HTMLCollection`, whether it be a property or a method, you are performing a query on the document, and that querying is quite expensive. Minimizing the number of times you access an `HTMLCollection` can greatly improve the performance of a script.

Perhaps the most important area in which to optimize `HTMLCollection` access is loops. Moving the length calculation into the initialization portion of a `for` loop was discussed previously. Now consider this example:

```
var images = document.getElementsByTagName("img"),
    i, len;

for (i=0, len=images.length; i < len; i++) {
    //process
}
```

The key here is that the `length` is stored in the `len` variable instead of constantly accessing the `length` property of the `HTMLCollection`. When using an `HTMLCollection` in a loop, you should make your next step a retrieval of a reference to the item you'll be using, as shown here, so as to avoid calling the `HTMLCollection` multiple times in the loop body:

```
var images = document.getElementsByTagName("img"),
    image,
    i, len;

for (i=0, len=images.length; i < len; i++) {
    image = images[i];
    //process
}
```

This code adds the `image` variable, which stores the current image. Once this is complete, there should be no further reason to access the `images` `HTMLCollection` inside the loop.

When writing JavaScript, it's important to realize when `HTMLCollection` objects are being returned, so you can minimize accessing them. An `HTMLCollection` object is returned when any of the following occurs:

- A call to `getElementsByName()` is made.
- The `childNodes` property of an element is retrieved.
- The `attributes` property of an element is retrieved.
- A special collection is accessed, such as `document.forms`, `document.images`, and so forth.

Understanding when you're using `HTMLCollection` objects and making sure you're using them appropriately can greatly speed up code execution.

DEPLOYMENT

Perhaps the most important part of any JavaScript solution is the final deployment to the website or web application in production. You've done a lot of work before this point, architecting and optimizing a solution for general consumption. It's time to move out of the development environment and into the Web, where real users can interact with it. Before you do so, however, there are a number of issues that need to be addressed.

Build Process

One of the most important things you can do to ready JavaScript code for deployment is to develop some type of build process around it. The typical pattern for developing software is write-compile-test, in that you write the code, compile it, and then run it to ensure that it works. Since JavaScript is not a compiled language, the pattern has become write-test, where the code you write is the same code you test in the browser. The problem with this approach is that it's not optimal; the code you write should not be passed, untouched, to the browser, for the following reasons:

- **Intellectual property issues** — If you put the fully commented source code online, it's easier for others to figure out what you're doing, reuse it, and potentially figure out security holes.
- **File size** — You write code in a way that makes it easy to read, which is good for maintainability but bad for performance. The browser doesn't benefit from the extra white space, indentation, or verbose function and variable names.
- **Code organization** — The way you organize code for maintainability isn't necessarily the best way to deliver it to the browser.

For these reasons, it's best to define a build process for your JavaScript files.

A build process starts by defining a logical structure for storing your files in source control. It's best to avoid having a single file that contains all of your JavaScript. Instead, follow the pattern that is typically taken in object-oriented languages: separate each object or custom type into its own file. Doing so ensures that each file contains just the minimum amount of code, making it easier to make changes without introducing errors. Additionally, in environments that use concurrent source control systems such as CVS or Subversion, this reduces the risk of conflicts during merge operations.

Keep in mind that separating your code into multiple files is for maintainability and not for deployment. For deployment, you'll want to combine the source files into one or more rollup files. It's recommended that web applications use the smallest number of JavaScript files possible, because HTTP requests are some of the main performance bottlenecks on the Web. Keep in mind that including a JavaScript file via the `<script>` tag is a blocking operation that stops all other downloads while the code is downloaded and executed. Therefore, try to logically group JavaScript code into deployment files.

Once you've organized your file and directory structure, and determined what should be in your deployment files, you'll want to create a build system. The *Ant build tool* (<http://ant.apache.org>) was created to automate Java build processes but has gained popularity with web application developers because of its ease of use and coverage by software engineers, such as Julien Lecomte, who have written tutorials explaining how to use Ant for JavaScript and CSS build automation (Lecomte's article can be found at www.julienlecomte.net/blog/2007/09/16/).

Ant is ideal for a JavaScript build system because of its simple file-manipulation capabilities. For example, you can easily get a list of all files in a directory and then combine them into a single file, as shown here:



Available for download on Wrox.com

```
<project name="JavaScript Project" default="js.concatenate">

    <!-- the directory to output to -->
    <property name="build.dir" value=".//js" />

    <!-- the directory containing the source files -->
    <property name="src.dir" value=".//dev/src" />

    <!-- Target to concatenate all JS files -->
    <!-- Credit: Julien Lecomte, http://www.julienlecomte.net/blog/2007/09/16/ -->
    <target name="js.concatenate">
        <concat destfile="${build.dir}/output.js">
            <filelist dir="${src.dir}/js" file="a.js, b.js"/>
            <fileset dir="${src.dir}/js" includes="*.js" excludes="a.js, b.js"/>
        </concat>
    </target>

</project>
```

SampleAntDir/build.xml

This `build.xml` file defines two properties: a build directory into which the final file should be output and a source directory where the JavaScript source files exist. The target `js.concatenate` uses the `<concat>` element to specify a list of files that should be concatenated and the location where the resulting file should be output. The `<filelist>` element is used to indicate that the files `a.js` and `b.js` should be first in the concatenated file, and the `<fileset>` element indicates that all of the other files in the directory, with the exception of `a.js` and `b.js`, should be added afterwards. The resulting file will be output to `/js/output.js`.

With Ant installed, you can go to the directory in which this `build.xml` file exists, and run this code snippet:

```
ant
```

The build process is then kicked off, and the concatenated file is produced. If there are other targets in the file, you can execute just the `js.concatenate` target, using the following code:

```
ant js.concatenate
```

Depending on your needs, the build process can be changed to include more or less steps. Introducing the build step to your development cycle gives you a location where you can add more processing for JavaScript files prior to deployment.

Validation

Even though IDEs that understand and support JavaScript are starting to appear, most developers still check their syntax by running code in a browser. There are a couple of problems with this approach. First, this validation can't be easily automated or ported from system to system. Second, aside from syntax errors, problems are encountered only when code is executed, leaving it possible for errors to occur. There are several tools available to help identify potential issues with JavaScript code, the most popular being *Douglas Crockford's JSLint* (www.jslint.com).

JSLint looks for syntax errors and common coding errors in JavaScript code. Some of the potential issues it surfaces are as follows:

- Use of `eval()`
- Use of undeclared variables
- Omission of semicolons
- Improper line breaks
- Incorrect comma usage
- Omission of braces around statements
- Omission of break in switch cases
- Variables being declared twice
- Use of `with`
- Incorrect use of equals (instead of double- or triple-equals)
- Unreachable code

The online version is available for easy access, but it can also be run on the command line using the Java-based Rhino JavaScript engine (www.mozilla.org/rhino/). To run JSLint on the command line, you first must download Rhino and then download the Rhino version of JSLint from www.jslint.com/rhino/. Once it is installed, you can run JSLint on the command line using the following syntax:

```
java -jar rhino-1.6R7.jar jslint.js [input files]
```

Here is an example:

```
java -jar rhino-1.6R7.jar jslint.js a.js b.js c.js
```

If there are any syntax issues or potential errors in the given files, a report is output with the errors and warnings. If there are no issues, then the code completes without displaying any messages.

You can run JSLint as part of your build process using Ant with a target such as this:



```
<target name="js.verify">
    <apply executable="java" parallel="false">
        <fileset dir="${build.dir}" includes="output.js"/>
        <arg line="-jar />
        <arg path="${rhino.jar}"/>
        <arg path="${jslint.js}"/>
        <srcfile/>
    </apply>
</target>
```

SampleAntDir/build.xml

This target assumes that the location of the Rhino jar file is specified in a property called `rhino.jar`, and the location of the JSLint Rhino file is specified as a property called `jslint.js`. The `output.js` file is passed into JSLint to be verified and will output any issues that it finds.

Adding code validation to your development cycle helps to avoid errors down the road. It's recommended that developers add some type of code validation to the build process as a way of identifying potential issues before they become errors.



A list of JavaScript code validators can be found in Appendix D.

Compression

When talking about JavaScript file compression, you're really talking about two things: *code size* and *wire weight*. Code size refers to the number of bytes that need to be parsed by the browser, and wire weight refers to the number of bytes that are actually transmitted from the server to the browser. In the early days of web development, these two numbers were almost always identical, because source files were transmitted, unchanged, from server to client. In today's Web, however, the two are rarely equal and realistically should never be.

File Compression

Because JavaScript isn't compiled into byte code and is actually transmitted as source code, the code files often contain additional information and formatting that isn't necessary for browser execution. Comments, extra white space, and long variable or function names improve readability for developers

but are unnecessary extra bytes when sent to the browser. You can, however, decrease the file size using a compressor tool.

Compressors typically perform some or all of the following steps:

- Remove extra white space (including line breaks)
- Remove all comments
- Shorten variable names

There are many compressors available for JavaScript (a full list is included in Appendix D), but the best is arguably the YUI Compressor, available at <http://developer.yahoo.com/yui/compressor/>. The YUI Compressor uses the Rhino JavaScript parser to tokenize JavaScript code. This token stream can then be used to create an optimal version of the code without white space or comments. Unlike regular expression-based compressors, the YUI Compressor is guaranteed to not introduce syntax errors and can therefore safely shorten local variable names.

The YUI Compressor comes as a Java jar file named `yuicompressor-x.y.z.jar`, where `x.y.z` is the version number. At the time of this writing, 2.4.6 is the most recent version. You can execute the YUI Compressor using the following command-line format:

```
java -jar yuicompressor-x.y.z.jar [options] [input file]
```

Options for the YUI Compressor are listed in the following table.

OPTION	DESCRIPTION
<code>-h</code>	Displays help information.
<code>-o outputFile</code>	Specifies the name of the output file. If not included, the output file name is the input file name appended with “-min”. For example, an input file of <code>input.js</code> would produce <code>input-min.js</code> .
<code>--line-break column</code>	Indicates to include a line break after the <code>column</code> number of characters. By default, the compressed file is output on one line, which may cause issues in some source control systems.
<code>-v, --verbose</code>	Verbose mode. Outputs hints for better compression and warnings.
<code>--charset charset</code>	Indicates the character set that the input file is in. The output file will use the same character set.
<code>--nomunge</code>	Turns off local variable name replacement.
<code>--disable-optimizations</code>	Turns off YUI Compressor’s micro-optimizations.
<code>--preserve-semi</code>	Preserves unnecessary semicolons that would otherwise have been removed.

For example, the following can be used to compress the `CookieUtil.js` file into a file named simply `cookie.js`:

```
java -jar yuicompressor-2.3.5.jar -o cookie.js CookieUtil.js
```

The YUI Compressor can also be used from Ant by calling the `java` executable directly, as in this example:



Available for
download on
Wrox.com

```
<!-- Credit: Julien Lecomte, http://www.julienlecomte.net/blog/2007/09/16/ -->
<target name="js.compress">
    <apply executable="java" parallel="false">
        <fileset dir="${build.dir}" includes="output.js"/>
        <arg line="-jar"/>
        <arg path="${yuicompressor.jar}"/>
        <arg line="-o ${build.dir}/output-min.js"/>
        <srcfile/>
    </apply>
</target>
```

SampleAntDir/build.xml

This target includes a single file, the `output.js` file created as part of the build process, and passes it to the YUI Compressor. The output file is specified as `output-min.js` in the same directory. This assumes that the property `yuicompressor.jar` contains the location of the YUI Compressor jar file. You can run this target using the following command:

```
ant js.compress
```

All JavaScript files should be compressed using the YUI Compressor or a similar tool before being deployed to a production environment. Adding a step in your build process to compress JavaScript files is an easy way to ensure that this always happens.

HTTP Compression

Wire weight refers to the actual number of bytes sent from the server to the browser. The number of bytes doesn't necessarily have to be the same as the code size, because of the compression capabilities of both the server and the browser. All of the five major web browsers — Internet Explorer, Firefox, Safari, Chrome, and Opera — support client-side decompression of resources that they receive. The server is therefore able to compress JavaScript files using server-dependent capabilities. As part of the server response, a header is included indicating that the file has been compressed using a given format. The browser then looks at the header to determine that the file is compressed, and then decompresses it using the appropriate format. The result is that the amount of bytes transferred over the network is significantly less than the original code size.

For the Apache web server, there are two modules that make HTTP compression easy: `mod_gzip` (for Apache 1.3.x) and `mod_deflate` (for Apache 2.0.x). For `mod_gzip`, you can enable automatic compression of JavaScript files by adding the following line to either your `httpd.conf` file or a `.htaccess` file:

```
#Tell mod_gzip to include any file ending with .js
mod_gzip_item_include file \.js$
```

This line tells `mod_gzip` to compress any file ending with `.js` that is requested from the browser. Assuming that all of your JavaScript files end with `.js`, this will compress every request and apply the appropriate headers to indicate that the contents have been compressed. For more information about `mod_gzip`, visit the project site at www.sourceforge.net/projects/mod-gzip/.

For `mod_deflate`, you can similarly include a single line to ensure that the JavaScript files are compressed before being sent. Place the following line in either your `httpd.conf` file or a `.htaccess` file:

```
#Tell mod_deflate to include all JavaScript files
AddOutputFilterByType DEFLATE application/x-javascript
```

Note that this line uses the MIME type of the response to determine whether or not to compress it. Remember that even though `text/javascript` is used for the type attribute of `<script>`, JavaScript files are typically served with a MIME type of `application/x-javascript`. For more information on `mod_deflate`, visit http://httpd.apache.org/docs/2.0/mod/mod_deflate.html.

Both `mod_gzip` and `mod_deflate` result in savings of around 70 percent of the original file size of JavaScript files. This is largely due to the fact that JavaScript files are plain text and can therefore be compressed very efficiently. Decreasing the wire weight of your files decreases the amount of time it takes to transmit to the browser. Keep in mind that there is a slight trade-off, because the server must spend time compressing the files on each request, and the browser must take some time to decompress the files once they arrive. Generally speaking, however, the trade-off is well worth it.



Most web servers, both open source and commercial, have some HTTP compression capabilities. Please consult the documentation for your server to determine how to configure compression properly.

SUMMARY

As JavaScript development has matured, best practices have emerged. What once was considered a hobby is now a legitimate profession and, as such, has experienced the type of research into maintainability, performance, and deployment traditionally done for other programming languages.

Maintainability in JavaScript has to do partially with the following code conventions:

- Code conventions from other languages may be used to determine when to comment and how to indent, but JavaScript requires some special conventions to make up for the loosely typed nature of the language.
- Since JavaScript must coexist with HTML and CSS, it's also important to let each wholly define its purpose: JavaScript should define behavior, HTML should define content, and CSS should define appearance.

- Any mixing of these responsibilities can lead to difficult-to-debug errors and maintenance issues.

As the amount of JavaScript has increased in web applications, performance has become more important. Therefore, you should keep these things in mind:

- The amount of time it takes JavaScript to execute directly affects the overall performance of a web page, so its importance cannot be dismissed.
- A lot of the performance recommendations for C-based languages also apply to JavaScript relating to loop performance and using switch statements instead of `if`.
- Another important thing to remember is that DOM interactions are expensive, so you should limit the number of DOM operations.

The last step in the process is deployment. Here are some key points discussed in this chapter:

- To aid in deployment, you should set up a build process that combines JavaScript files into a small number of files (ideally just one).
- Having a build process also gives you the opportunity to automatically run additional processes and filters on the source code. You can, for example, run a JavaScript verifier to ensure that there are no syntax errors or potential issues with the code.
- It's also recommended to use a compressor to get the file as small as possible before deployment.
- Coupling that with HTTP compression ensures that the JavaScript files are as small as possible and will have the least possible impact on overall page performance.

25

Emerging APIs

WHAT'S IN THIS CHAPTER?

- Creating smooth animations
- Working with files
- Background JavaScript with Web Workers

The introduction of HTML5 also spurred a tremendous growth in JavaScript APIs aimed at the future of web applications. These APIs are not part of the HTML5 specification but rather exist in their own specifications that are often bundled “HTML5 Related APIs”. All of the APIs in this chapter are still undergoing a fair amount of work and are not fully stable.

Despite that, browsers have already begun implementing the various APIs, and web application developers have begun using them. You’ll note that many of these APIs have browser-specific prefixes in front of them, such as “`ms`” for Microsoft or “`webkit`” for Chrome and Safari. These prefixes allow browsers to experiment with new APIs while they are still in development, knowing that the final, nonprefixed version will be consistent with other browsers.

requestAnimationFrame()

For a long time, timers and intervals have been the state of the art for JavaScript-based animations. While CSS transitions and animations make some animations easy for web developers, little has changed in the world of JavaScript-based animation over the years. Firefox 4 was the first browser to include a new API for JavaScript animations called `mozRequestAnimationFrame()`. This method indicates to the browser that an animation is taking place so that the browser may, in turn, determine the best way to schedule a redraw.

Early Animation Loops

The typical way to create animations in JavaScript is to use `setInterval()` to manage all animations. A basic animation loop using `setInterval()` looks like this:

```
(function() {
    function updateAnimations() {
        doAnimation1();
        doAnimation2();
        //etc.
    }

    setInterval(updateAnimations, 100);
})();
```

To build out a small animation library, the `updateAnimations()` method would cycle through the running animations and make the appropriate changes to each one (for example, both a news ticker and a progress bar running together). If there are no animations to update, the method can exit without doing anything and perhaps even stop the animation loop until more animations are ready for updating.

The tricky part about this animation loop is knowing what the delay should be. The interval has to be short enough to handle a variety of different animation types smoothly but long enough so as to produce changes the browser can actually render. Most computer monitors refresh at a rate of 60 Hz, which basically means there's a repaint 60 times per second. Most browsers cap their repaints so they do not attempt to repaint any more frequently than that, knowing that the end user gets no improvement in experience.

Therefore, the best interval for the smoothest animation is $1000\text{ms} / 60$, or about 17ms. You'll see the smoothest animation at this rate, because you're more closely mirroring what the browser is capable of doing. Multiple animations may need to be throttled so as not to complete too quickly when using an animation loop with a 17ms interval.

Even though `setInterval()`-based animation loops are more efficient than having multiple sets of `setTimeout()`-based loops, there are still problems. Neither `setInterval()` nor `setTimeout()` are intended to be precise. The delay you specify as the second argument is only an indication of when the code is added in the browser's UI thread queue for possible execution. If there are other jobs in the queue ahead of it, then that code waits to be executed. In short: the millisecond delay is not an indication of when the code will be executed, only an indication of when the job will be queued. If the UI thread is busy, perhaps dealing with user actions, then that code will not execute immediately.

Problems with Intervals

Understanding when the next frame will be drawn is key to smooth animations, and until recently, there was no way to guarantee when the next frame would be drawn in a browser. As `<canvas>` became popular and new browser-based games emerged, developers became increasingly frustrated with the inaccuracy of `setInterval()` and `setTimeout()`.

Exacerbating these problems is the timer resolution of the browser. Timers are not accurate to the millisecond. Here are some common timer resolutions:

- Internet Explorer 8 and earlier have a timer resolution of 15.625ms.
- Internet Explorer 9 and later have a timer resolution of 4ms.
- Firefox and Safari have a timer resolution of ~10ms.
- Chrome has a timer resolution of 4ms.

Internet Explorer prior to version 9 has a timer resolution of 15.625ms, so any value between 0 and 15 could be either 0 or 15 but nothing else. Internet Explorer 9 improved timer resolution to 4ms, but that's still not very specific when it comes to animations. Chrome's timer resolution is 4ms, while Firefox's and Safari's is 10ms. Complicating matters more, browsers have started to throttle timers for tabs that are in the background or inactive. So even if you set your interval for optimum display, you're still only getting close to the timing you want.

mozRequestAnimationFrame

Robert O'Callahan of Mozilla was thinking about this problem and came up with a unique solution. He pointed out that CSS transitions and animations benefit from the browser knowing that some animation should be happening, and so figured out the correct interval at which to refresh the UI. With JavaScript animations, the browser has no idea that an animation is taking place. His solution was to create a new method, called `mozRequestAnimationFrame()` that indicates to the browser that some JavaScript code is performing an animation. This allows the browser to optimize appropriately after running some code.

The `mozRequestAnimationFrame()` method accepts a single argument, which is a function to call prior to repainting the screen. This function is where you make appropriate changes to DOM styles that will be reflected with the next repaint. In order to create an animation loop, you can chain multiple calls to `mozRequestAnimationFrame()` together in the same way previously done with `setTimeout()`. For example:

```
function updateProgress(){
    var div = document.getElementById("status");
    div.style.width = (parseInt(div.style.width, 10) + 5) + "%";

    if (div.style.left != "100%"){
        mozRequestAnimationFrame(updateProgress);
    }
}

mozRequestAnimationFrame(updateProgress);
```

Since `mozRequestAnimationFrame()` runs the given function only once, you need to call it again manually the next time you want to make a UI change for the animation. You also need to manage when to stop the animation in the same way. The result is a very smooth animation.

So far, `mozRequestAnimationFrame()` has solved the problem of browsers not knowing when a JavaScript animation is happening and the problem of not knowing the best interval, but what about

the problem of not knowing when your code will actually execute? That's also covered with the same solution.

The function you pass into `mozRequestAnimationFrame()` actually receives an argument, which is a time code (in milliseconds since January 1, 1970) for when the next repaint will actually occur. This is a very important point: `mozRequestAnimationFrame()` actually schedules a repaint for some known point in the future and can tell you when that is. You're then able to determine how best to adjust your animation.

In order to determine how much time has passed since the last repaint, you can query `mozAnimationStartTime`, which contains the time code for the last repaint. Subtracting this value from the time passed into the callback allows you to figure out exactly how much time will have passed before your next set of changes are drawn to the screen. The typical pattern for using these values is as follows:

```
function draw(timestamp) {  
  
    //calculate difference since last repaint  
    var diff = timestamp - startTime;  
  
    //use diff to determine correct next step  
  
    //reset startTime to this repaint  
    startTime = timestamp;  
  
    //draw again  
    mozRequestAnimationFrame(draw);  
}  
  
var startTime = mozAnimationStartTime;  
mozRequestAnimationFrame(draw);
```

The key is to make the first call to `mozAnimationStartTime` outside of the callback that is passed to `mozRequestAnimationFrame()`. If you call `mozAnimationStartTime` inside of the callback, it will be equal to the time code that is passed in as an argument.

webkitRequestAnimationFrame and msRequestAnimationFrame

Chrome and Internet Explorer 10+ have also created their own implementations of `mozRequestAnimationFrame()` called `webkitRequestAnimationFrame()` and `msRequestAnimationFrame()`, respectively. These versions are slightly different from the Firefox version in two ways. First, there isn't a time code passed into the callback function, so you don't know when the next repaint will occur. Second, Chrome adds a second, optional argument that is the DOM element where the changes will occur. So if you know the repaint will occur only inside of one particular element on the page, you can limit the repaint to just that area.

It should come as no surprise that there is no equivalent to `mozAnimationStartTime`, since that information without the time of the next paint is not very useful. There is, however, a Chrome-specific method called `webkitCancelAnimationFrame()`, which cancels the previously scheduled repaint.

If you don't need precise time differences, you can create an animation loop for Firefox 4+, Internet Explorer 10+, and Chrome with the following pattern:

```
(function(){

    function draw(timestamp){

        //calculate difference since last repaint
        var drawStart = (timestamp || Date.now()),
            diff = drawStart - startTime;

        //use diff to determine correct next step

        //reset startTime to this repaint
        startTime = drawStart;

        //draw again
        requestAnimationFrame(draw);
    }

    var requestAnimationFrame = window.requestAnimationFrame ||
                               window.mozRequestAnimationFrame ||
                               window.webkitRequestAnimationFrame ||
                               window.msRequestAnimationFrame,
        startTime = window.mozAnimationStartTime || Date.now();
    requestAnimationFrame(draw);
})();
```

This pattern uses the available features to create an animation loop with some idea of how much time has passed. In Firefox, this uses the time code information that is available, while Chrome and Internet Explorer default to the less-accurate `Date` object. When using this pattern, you will get a general idea from the time difference of how much time has passed, but it certainly isn't going to tell you the next time a repaint will occur in Chrome or Internet Explorer. Still, it's better to have some idea of how much time has passed rather than no idea.

By checking for the standard function name first and then the browser-specific ones, this animation loop will continue to work in the future.

The `requestAnimationFrame()` API is now being drafted as a new recommendation by the W3C and is being worked on jointly by Mozilla and Google as part of the Web Performance Group.

PAGE VISIBILITY API

A major pain point for web developers is knowing when users are actually interacting with the page. If a page is minimized or hidden behind another tab, it may not make sense to continue functionality such as polling the server for updates or performing animations. The Page Visibility API aims to give developers information about whether or not the page is visible to the user.

The API itself is very simple, consisting of three parts:

- `document.hidden` — A Boolean value indicating if the page is hidden from view. This may mean the page is in a background tab or that the browser is minimized.

- `document.visibilityState` — A value indicating one of four states:
 - The page is in a background tab or the browser is minimized.
 - The page is in the foreground tab.
 - The actual page is hidden, but a preview of the page is visible (such as in Windows 7 when moving the mouse over an icon in the taskbar shows a preview).
 - The page is being prerendered off screen.
- `visibilitychange` event — This event fires when a document changes from hidden to visible or vice versa.

As of the time of this writing, only Internet Explorer 10 and Chrome have implemented the Page Visibility API. Internet Explorer has prefixed everything with “`ms`” while Chrome has prefixed everything with “`webkit`”. So `document.hidden` is implemented as `document.msHidden` in Internet Explorer and `document.webkitHidden` in Chrome. The best way to check for support is with this code:



```
function isHiddenSupported() {
    return typeof (document.hidden || document.msHidden ||
        document.webkitHidden) != "undefined";
}
```

[PageVisibilityAPIExample01.htm](#)

Similarly, you can check to see if the page is hidden by using the same construct:

```
if (document.hidden || document.msHidden || document.webkitHidden) {
    //page is hidden
} else {
    //page is not hidden
}
```

[PageVisibilityAPIExample01.htm](#)

Note that this code will indicate that the page is not hidden in unsupported browsers, which is the intentional behavior of the API for backwards compatibility.

To be notified when the page changes from visible to hidden or hidden to visible, you can listen for the `visibilitychange` event. In Internet Explorer, this event is called `msvisibilitychange` and in Chrome it’s called `webkitvisibilitychange`. In order for this event to work in both browsers, you need to assign the same event handler to each event, as in this example:

```
function handleVisibilityChange() {
    var output = document.getElementById("output"),
        msg;

    if (document.hidden || document.msHidden || document.webkitHidden) {
        msg = "Page is now hidden." + (new Date()) + "<br>";
    } else {
        msg = "Page is now visible." + (new Date()) + "<br>";
    }
}
```

```

        }

        output.innerHTML += msg;

    }

//need to add to both
EventUtil.addHandler(document, "msvisibilitychange", handleVisibilityChange);
EventUtil.addHandler(document, "webkitvisibilitychange", handleVisibilityChange);

```

[PageVisibilityAPIExample01.htm](#)

This code works well in both Internet Explorer and Chrome. Furthermore, this part of the API is relatively stable, so it's safe to use the code in real web applications.

The biggest difference between the implementations is with `document.visibilityState`. Internet Explorer 10 PR 2's `document.msVisibilityState` is a numeric value representing one of four constants:

1. `document.MS_PAGE_HIDDEN (0)`
2. `document.MS_PAGE_VISIBLE (1)`
3. `document.MS_PAGE_PREVIEW (2)`
4. `document.MS_PAGE_PRERENDER (3)`

In Chrome, `document.webkitVisibilityState` is one of three possible string values:

1. “hidden”
2. “visible”
3. “prerender”

Chrome does not feature constants for each state, though the final implementation will likely contain them.

Because of these differences, it's recommended to not rely on the vendor-prefixed version of `document.visibilityState` and instead stick to using `document.hidden`.

GEOLOCATION API

One of the most interesting, and well-supported, new APIs is *geolocation*. Geolocation allows JavaScript to access information about the user's current position. Of course, this can be done only if the user explicitly allows that information to be shared with the page. Whenever a page tries to access geolocation information, the browser displays a dialog asking for permission to share that information. Figure 25-1 shows the dialog as shown in Chrome.



FIGURE 25-1

The Geolocation API is implemented as `navigator.geolocation` and has three methods. The first method is `getCurrentPosition()`, which triggers the confirmation dialog to allow access to the geolocation information. This method accepts three arguments: a success callback function, an optional failure callback function, and an optional options object.

The success callback receives a `Position` object as its only argument, and that object has two properties: `coords` and `timestamp`. The `coords` object must contain the following information about the location:

- `latitude` — The latitude given in degrees.
- `longitude` — The longitude given in degrees.
- `accuracy` — The accuracy of the coordinates in meters. The higher the number, the less accurate.

A browser may also optionally include the following properties:

- `altitude` — The height of the position in meters or `null` if not available.
- `altitudeAccuracy` — The accuracy of the `altitude` in meters. The higher the number, the less accurate.
- `heading` — The compass direction in degrees, where 0 degrees is true north. If the direction can't be determined, the value is `Nan`.
- `speed` — The velocity in meters per second or `null` if the information can't be determined.

In practice, most web applications tend to use `latitude` and `longitude` more frequently than the other properties. For example, a common use is to draw a location of the user on a map:

```
navigator.geolocation.getCurrentPosition(function(position) {
    drawMapCenteredAt(position.coords.latitude, positions.coords.longitude);
});
```

The failure callback also receives an argument when called. The argument is an object that has two properties: `message` and `code`. The `message` property is a human-readable error message explaining why the error occurred, while the `code` property is a numeric value indicating the type of error: user denied permission (1), position isn't available (2), or timeout (3). In practice, most web applications simply log such errors but don't necessarily change the user interface as a result. For example:

```
navigator.geolocation.getCurrentPosition(function(position) {
    drawMapCenteredAt(position.coords.latitude, positions.coords.longitude);
}, function(error) {
    console.log("Error code: " + error.code);
    console.log("Error message: " + error.message);
});
```

The third argument to `getCurrentPosition()` is an `options` object for the type of information. There are three options that can be set: `enableHighAccuracy`, which is a Boolean value indicating that the best possible position is requested; `timeout`, which is the amount of time in milliseconds to wait for the position to be determined; and `maximumAge`, which is the number of milliseconds that the last coordinates can be used before a new location should be determined. For example:

```

navigator.geolocation.getCurrentPosition(function(position){
    drawMapCenteredAt(position.coords.latitude, position.coords.longitude);
}, function(error){
    console.log("Error code: " + error.code);
    console.log("Error message: " + error.message);
}, {
    enableHighAccuracy: true,
    timeout: 5000,
    maximumAge: 25000
});

```

All three options are optional and may be provided on their own or in combination with the others. Unless you really need very accurate information, it's advisable to keep `enableHighAccuracy` as `false` (the default). Enabling this option may require more time and in mobile devices may require the use of more power. Similarly, if you're not actively tracking the user's position, then `maximumAge` can be set to `Infinity` to always use the last coordinates.

If you want to track the user's position, you can use another method called `watchPosition()`. This method accepts the exact same arguments as `getCurrentPosition()`. In practice, `watchPosition()` is the same as calling `getCurrentPosition()` periodically. Upon first calling the method, the current position is retrieved and the success or error callback executed. After that, `watchPosition()` waits for a signal from the system that the position has changed (it does not poll the position).

The call to `watchPosition()` returns a numeric identifier that is used to track the watch operation. This value can be used to cancel the watch by passing it to the `clearWatch()` method (similar to using `setTimeout()` and `clearTimeout()`). For example:

```

var watchId = navigator.geolocation.watchPosition(function(position){
    drawMapCenteredAt(position.coords.latitude, position.coords.longitude);
}, function(error){
    console.log("Error code: " + error.code);
    console.log("Error message: " + error.message);
});

clearWatch(watchId);

```

This example calls `watchPosition()` and stores the returned identifier in `watchId`. Later, `watchId` is passed to `clearWatch()` to cancel the watch operation.

Geolocation is supported in Internet Explorer 9+, Firefox 3.5+, Opera 10.6+, Safari 5+, Chrome, Safari for iOS, and WebKit for Android. For an excellent geolocation example, see <http://html5demos.com/geo>.

FILE API

One of the major pain points of web applications has been the inability to interact with files on a user's computer. Since before 2000, the only way to deal with files was to place `<input type="file">` into a form and leave it at that. The File API is designed to give web developers access to files on the client computer in a secure manner that allows for better interaction with those files. The File API is supported in Internet Explorer 10+, Firefox 4+, Safari 5.0.5+, Opera 11.1+, and Chrome.

The File API is still based around the file input field of a form but adds the ability to access the file information directly. HTML5 adds a `files` collection to DOM for the file input element. When one or more files are selected in the field, the `files` collection contains a sequence of `File` objects that represent each file. Each `File` object has several read-only properties, including:

- `name` — The file name on the local system.
- `size` — The size of the file in bytes.
- `type` — A string containing the MIME type of the file.
- `lastModifiedDate` — A string representing the last time the file was modified. This property has been implemented only in Chrome.

For instance, you can retrieve information about each file selected by listening for the `change` event and then looking at the `files` collection:



```
var fileList = document.getElementById("files-list");
EventUtil.addHandler(fileList, "change", function(event) {
    var files = EventUtil.getTarget(event).files,
        i = 0,
        len = files.length;

    while (i < len) {
        console.log(files[i].name + " (" + files[i].type + ", " + files[i].size +
                    " bytes)");
        i++;
    }
});
```

[FileAPIExample01.htm](#)

This example simply outputs the information about each file to the console. This ability alone is a big step forward for web applications, but the File API goes further by allowing you to actually read data from the files via the `FileReader` type.

The `FileReader` Type

The `FileReader` type represents an asynchronous file reading mechanism. You can think of `FileReader` as similar to `XMLHttpRequest`, only it is used for reading files from the filesystem as opposed to reading data from the server. The `FileReader` type offers several methods to read in file data:

- `readAsText(file, encoding)` — Reads the file as plain text and stores the text in the `result` property. The second argument, the encoding type, is optional.
- `readAsDataURL(file)` — Reads the file and stores a data URI representing the files in the `result` property.
- `readAsBinaryString(file)` — Reads the file and stores a string where each character represents a byte in the `result` property.
- `readAsArrayBuffer(file)` — Reads the file and stores an `ArrayBuffer` containing the file contents in the `result` property.

These various ways of reading in a file allow for maximum flexibility in dealing with the file data. For instance, you may wish to read an image as a data URI in order to display it back to the user, or you may wish to read a file as text in order to parse it.

Since the read happens asynchronously, there are several events published by each `FileReader`. The three most useful events are `progress`, `error`, and `load`, which indicate when more data is available, when an error occurred, and when the file is fully read, respectively.

The `progress` event fires roughly every 50ms and has the same information available as the XHR `progress` event: `lengthComputable`, `loaded`, and `total`. Additionally, the `FileReader`'s `result` property is readable during the `progress` event even though it may not contain all of the data yet.

The `error` event fires if the file cannot be read for some reason. When the `error` event fires, the `error` property of the `FileReader` is filled in. This object has a single property, `code`, which is an error code of 1 (file not found), 2 (security error), 3 (read was aborted), 4 (file isn't readable), or 5 (encoding error).

The `load` event fires when the file has been successfully loaded; it will not fire if the `error` event has fired. Here's an example using all three events:



```
var filesList = document.getElementById("files-list");
EventUtil.addHandler(filesList, "change", function(event){
    var info = "",
        output = document.getElementById("output"),
        progress = document.getElementById("progress"),
        files = EventUtil.getTarget(event).files,
        type = "default",
        reader = new FileReader();

    if (/image/.test(files[0].type)){
        reader.readAsDataURL(files[0]);
        type = "image";
    } else {
        reader.readAsText(files[0]);
        type = "text";
    }

    reader.onerror = function(){
        output.innerHTML = "Could not read file, error code is " +
                           reader.error.code;
    };

    reader.onprogress = function(event){
        if (event.lengthComputable){
            progress.innerHTML = event.loaded + "/" + event.total;
        }
    };
}

reader.onload = function(){

    var html = "";

    switch(type){
        case "image":
            html = "<img src=\"" + reader.result + "\">";
            break;
    }
}
```

```

        case "text":
            html = reader.result;
            break;

    }
    output.innerHTML = html;
};

});

```

FileAPIExample02.htm

This code reads a file from a form field and displays it on the page. If the file has a MIME type indicating it's an image, then a data URI is requested and, upon `load`, this data URI is inserted as an image into the page. If the file is not an image, then it is read in as a string and output as is into the page. The `progress` event is used to track and display the bytes of data being read, while the `error` event watches for any errors.

You can stop a read in progress by calling the `abort()` method, in which case an `abort` event is fired. After the firing of `load`, `error`, or `abort`, an event called `loadend` is fired. The `loadend` event indicates that all reading has finished for any of the three reasons.

The `readAsText()` and `readAsDataURL()` methods are supported across all implementing browsers. Internet Explorer 10 PR2 does not implement `readAsBinaryString()` or `readAsArrayBuffer()`.

Partial Reads

In some cases you may want to read only parts of a file instead of the whole file. To that end, the `File` object has a method called `slice()`, which is implemented as `mozSlice()` in Firefox and `webkitSlice()` in Chrome; Safari as of version 5.1 does not implement this method. The `slice()` method accepts two arguments: the starting byte and the number of bytes to read. This method returns an instance of `Blob`, which is actually the super type of `File`. The following function normalizes `slice()` across the various implementations:



```

function blobSlice(blob, startByte, length){
    if (blob.slice){
        return blob.slice(startByte, length);
    } else if (blob.webkitSlice){
        return blob.webkitSlice(startByte, length);
    } else if (blob.mozSlice){
        return blob.mozSlice(startByte, length);
    } else {
        return null;
    }
}

```

FileAPIExample03.htm

A `Blob` also has `size` and `type` properties, as well as the `slice()` method for further cutting down the data. You can read from a `Blob` by using a `FileReader` as well. This example reads just the first 32 bytes from a file:



Available for
download on
Wrox.com

```

var filesList = document.getElementById("files-list");
EventUtil.addHandler(filesList, "change", function(event){
    var info = "",
        output = document.getElementById("output"),
        progress = document.getElementById("progress"),
        files = EventUtil.getTarget(event).files,
        reader = new FileReader(),
        blob = blobSlice(files[0], 0, 32);

    if (blob){
        reader.readAsText(blob);

        reader.onerror = function(){
            output.innerHTML = "Could not read file, error code is " +
                reader.error.code;
        };

        reader.onload = function(){
            output.innerHTML = reader.result;
        };
    } else {
        alert("Your browser doesn't support slice() .");
    }
});
```

[FileAPIExample03.htm](#)

Reading just parts of a file can save time, especially when you're just looking for a specific piece of data, such as a file header.

Object URLs

Object URLs, also sometimes called *blob URLs*, are URLs that reference data stored in a `File` or `Blob`. The advantage of object URLs is that you don't need to read the file contents into JavaScript in order to use them. Instead, you simply provide the object URL in the appropriate place. To create an object URL, use the `window.URL.createObjectURL()` method and pass in the `File` or `Blob` object. Chrome implements this as `window.webkitURL.createObjectURL()`, so the following function can be used to normalize this functionality:

```

function createObjectURL(blob){
    if (window.URL){
        return window.URL.createObjectURL(blob);
    } else if (window.webkitURL){
        return window.webkitURL.createObjectURL(blob);
    } else {
        return null;
    }
}
```

[FileAPIExample04.htm](#)



Available for
download on
Wrox.com

```
var fileList = document.getElementById("files-list");
EventUtil.addHandler(fileList, "change", function(event) {
    var info = "",
        output = document.getElementById("output"),
        progress = document.getElementById("progress"),
        files = EventUtil.getTarget(event).files,
        reader = new FileReader(),
        url = createObjectURL(files[0]);

    if (url) {
        if (/image/.test(files[0].type)) {
            output.innerHTML = "<img src=\"" + url + "\">";
        } else {
            output.innerHTML = "Not an image.";
        }
    } else {
        output.innerHTML = "Your browser doesn't support object URLs.";
    }
});
```

FileAPIExample04.htm

By feeding the object URL directly into an `` tag, there is no need to read the data into JavaScript first. Instead, the `` tag goes directly to the memory location and reads the data into the page.

Once the data is no longer needed, it's best to free up the memory associated with it. Memory cannot be freed as long as an object URL is in use. You can indicate that the object URL is no longer needed by passing it to `window.URL.revokeObjectURL()` (`window.webkitURL.revokeObjectURL()` in Chrome). To handle both implementations, use this function:

```
function revokeObjectURL(url) {
    if (window.URL) {
        window.URL.revokeObjectURL(url);
    } else if (window.webkitURL) {
        window.webkitURL.revokeObjectURL(url);
    }
}
```

All object URLs are freed from memory automatically when the page is unloaded. Still, it is best to free each object URL as it is no longer needed to ensure the memory footprint of the page remains as low as possible.

Object URLs are supported in Internet Explorer 10+, Firefox 4+, and Chrome.

Drag-and-Drop File Reading

Combining the HTML5 Drag-and-Drop API with the File API allows you to create interesting interfaces for the reading of file information. After creating a custom drop target on a page, you can

drag files from the desktop and drop them onto the drop target. This fires the `drop` event just like dragging and dropping an image or link would. The files being dropped are available on `event.dataTransfer.files`, which is a list of `File` objects just like those available on a file input field.

The following example prints out information about files that are dropped on a custom drop target in the page:



Available for
download on
Wrox.com

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

function handleEvent(event){
    var info = "",
        output = document.getElementById("output"),
        files, i, len;

    EventUtil.preventDefault(event);

    if (event.type == "drop"){
        files = event.dataTransfer.files;
        i = 0;
        len = files.length;

        while (i < len){
            info += files[i].name + " (" + files[i].type + ", " + files[i].size +
                   " bytes)<br>";
            i++;
        }

        output.innerHTML = info;
    }
}

EventUtil.addHandler(droptarget, "dragenter", handleEvent);
EventUtil.addHandler(droptarget, "dragover", handleEvent);
EventUtil.addHandler(droptarget, "drop", handleEvent);
```

[FileAPIExample05.htm](#)

As with earlier drag-and-drop examples, you must cancel the default behavior of `dragenter`, `dragover`, and `drop`. During the `drop` event, the files become available on `event.dataTransfer.files`, and you can read their information at that time. One of the more popular ways to take advantage of this functionality is in a drag-and-drop file upload system using `XMLHttpRequest`.

File Upload with XHR

Since the File API gives you access to the contents of a file, it follows that you can use this to upload files directly to the server using XHR. Of course, you can very easily upload the contents of a file by passing the data into the `send()` method and using a POST request. However, then you would be passing the contents of the file, which you would need to grab on the server and save into another file. Ideally, you want to upload the file as if it were part of a form submission.

This can be done quite easily using the `FormData` type, first introduced in Chapter 21. Create a new `FormData` object and pass any `File` object as the value to the `append()` method. Then, pass the `FormData` object to the XHR `send()` method and you've successfully mimicked uploading a file using a form:



Available for download on
Wrox.com

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

function handleEvent(event) {
    var info = "",
        output = document.getElementById("output"),
        data, xhr,
        files, i, len;

    EventUtil.preventDefault(event);

    if (event.type == "drop") {
        data = new FormData();
        files = event.dataTransfer.files;
        i = 0;
        len = files.length;

        while (i < len) {
            data.append("file" + i, files[i]);
            i++;
        }

        xhr = new XMLHttpRequest();
        xhr.open("post", "FileAPIExample06Upload.php", true);
        xhr.onreadystatechange = function(){
            if (xhr.readyState == 4){
                alert(xhr.responseText);
            }
        };
        xhr.send(data);
    }
}

EventUtil.addHandler(droptarget, "dragenter", handleEvent);
EventUtil.addHandler(droptarget, "dragover", handleEvent);
EventUtil.addHandler(droptarget, "drop", handleEvent);
```

[FileAPIExample06.htm](#)

This example creates a `FormData` object with keys for each file, such as `file0`, `file1`, `file2`. Note that no extra processing is necessary for specifying a file as a form value. There's no need to use a `FileReader`, just pass in the `File` object itself.

Using `FormData` to upload files ensures that everything is handled as if it were a normal form on the server. That means useful shortcuts like the PHP `$_FILES` array are still available on the server for uploading files. Firefox 4+, Safari 5+, and Chrome all support `FormData` and uploading files in this way.

WEB TIMING

Page performance is always an area of concern for web developers. Up until recently, the only way to measure in-page performance characteristics was through increasingly complex and clever uses of the JavaScript Date object. The Web Timing API changes that by exposing internal browser metrics through a JavaScript API, allowing developers to directly access this information and do as they please with it. Unlike other APIs in this chapter, Web Timing is actually already a W3C Recommendation, though its adoption by browsers is relatively slow.

The center of Web Timing is the `window.performance` object. All metrics related to the page, both those already defined and those in the future, exist on this object. The Web Timing specification starts by defining two properties of `performance`.

The `performance.navigation` property is an object containing multiple properties relating to the page navigation. The properties are:

- `redirectCount` — The number of redirects before the page was loaded.
- `type` — A numeric constant for the type of navigation that has just occurred:
 - `performance.navigation.TYPE_NAVIGATE (0)` — The page was loaded for the first time.
 - `performance.navigation.TYPE_RELOAD (1)` — The page was reloaded.
 - `performance.navigation.TYPE_BACK_FORWARD (2)` — The page was navigated to using either the Back or the Forward button.

The `performance.timing` property, on the other hand, provides numerous properties that are simply time stamps (number of milliseconds since the epoch) of when various events occurred. The properties are:

- `navigationStart` — When navigation to a page begins.
- `unloadEventStart` — When the unload event for the previous page started. This is filled in only if the previous page is of the same origin as the new page; otherwise, it is 0.
- `unloadEventEnd` — When the unload event for the previous page completed. This is filled in only if the previous page is of the same origin as the new page; otherwise, it is 0.
- `redirectStart` — When a redirect was started for the current page, but only if the redirect happens within the same origin. Otherwise, this value is 0.
- `redirectEnd` — When a redirect was ended for the current page, but only if the redirect happens within the same origin. Otherwise, this value is 0.
- `fetchStart` — When the page begins to be fetched via HTTP GET.
- `domainLookupStart` — When the DNS lookup for the page begins.
- `domainLookupEnd` — When the DNS lookup for the page ends.
- `connectStart` — When the browser attempts to connect to the server.
- `connectEnd` — When the browser successfully connected to the server.
- `secureConnectionStart` — When an SSL connection was attempted from the browser. This value is 0 when SSL is not used.

- `requestStart` — When the browser starts requesting the page.
- `responseStart` — When the browser receives the first byte from the page.
- `responseEnd` — When the browser has received all of the page.
- `domLoading` — When `document.readyState` changed to “loading”.
- `domInteractive` — When `document.readyState` changed to “interactive”.
- `domContentLoadedEventStart` — When the `DOMContentLoaded` event is about to fire.
- `domContentLoadedEventEnd` — When the `DOMContentLoaded` event has fired and executed all event handlers.
- `domComplete` — When `document.readyState` changed to “complete”.
- `loadEventStart` — When the `load` event is just about to fire.
- `loadEventEnd` — When the `load` event has fired and all event handlers have executed.

Using the difference between various times can give you a good idea about how a page is being loaded into the browser and where the potential bottlenecks are hiding. For an excellent example of the Web Timing API, visit <http://webtimingdemo.appspot.com/>.

The Web Timing API is supported in Internet Explorer 10+ and Chrome.

WEB WORKERS

As web applications continue to increase in complexity, the need to do complex calculations has also increased. Long-running JavaScript processes cause the browser to freeze the user interface, which can be experienced as a “frozen” screen to the user. Web Workers solve this problem by executing JavaScript behind the scenes. Browsers may choose to implement Web Workers in any number of ways, including threads, background processes, and processes run on other processor cores. The actual implementation details aren’t as important as the freedom to run JavaScript without negatively affecting the user interface.

Web Workers are currently supported in Internet Explorer 10+, Firefox 3.5+, Safari 4+, Opera 10.6+, Chrome, and Safari for iOS 5.

Using a Worker

You can create a new Web Worker by instantiating a `Worker` object and passing in the file name containing JavaScript that the worker should execute. For example:

```
var worker = new Worker("stufftodo.js");
```

This line of code causes the browser to download `stufftodo.js` but the worker doesn’t actually start until it receives a message. You pass a message to the worker using the `postMessage()` method (similar to XDM):

```
worker.postMessage("start!");
```

The message can be any serializable value, though unlike XDM, all supporting browsers accept object arguments for `postMessage()` (Safari 4 was the last browser to support Web Workers and only support string messages). So you can feel free to pass data in as any sort of object, as in this example:

```
worker.postMessage({
    type: "command",
    message: "start!"
});
```

Generally speaking, any values that can be serialized into JSON structures can also be passed using `postMessage()`. This means that, as with XDM, values are copied into workers rather than passed directly.

The worker communicates back to the page through two events: `message` and `error`. The `message` event behaves the same as in XDM, with data from the worker arriving through `event.data`. This data may also be any type of serializable value:

```
worker.onmessage = function(event) {
    var data = event.data;

    //do something with data
}
```

The `error` event is the way that the worker indicates it could not complete a given task. It fires when an error occurs during JavaScript execution inside of the worker. The `event` object for the `error` event has three properties: `filename`, which is the file name in which the error occurred; `lineno`, which is the line number of the error in that file; and `message`, which is the complete error message.

```
worker.onerror = function(event) {
    console.log("ERROR: " + event.filename + " (" + event.lineno + "): " +
               event.message);
};
```

It's a good idea to always provide an `onerror` event handler for Web Workers, even if it does nothing else but log an error. Otherwise, workers will silently fail when an error occurs.

You can completely stop a worker at any point in time by calling the `terminate()` method. Doing so means that the worker is stopped immediately and does not finish any remaining processing (`error` and `message` events are not fired).

```
worker.terminate(); //stop the worker immediately
```

Worker Global Scope

The most important thing to understand about a Web Worker is that its JavaScript is executed in a completely different scope than code in the web page. There is a different global object and different objects and methods available inside of a Web Worker. Inside of a Web Worker, there is no access to the DOM and, indeed, no way to affect the appearance of a page in any way.

The global object inside of a Web Worker is the `worker` object itself. That means accessing either `this` or `self` in the global scope will result in accessing the working object. There is also a minimal environment inside of the worker to allow it to process data:

- A minimal `navigator` object containing `onLine`, `appName`, `appVersion`, `userAgent`, and `platform` properties.
- A read-only `location` object.
- `setTimeout()`, `setInterval()`, `clearTimeout()`, and `clearInterval()`.
- The `XMLHttpRequest` constructor.

As you can see, the environment of a worker is quite limited as compared to the page environment.

When a page calls `postMessage()` on a worker, that data is transmitted asynchronously to the worker and results in a message event firing inside of the worker. So to respond to data that is sent from a page, you must create an `onmessage` event handler:

```
//inside worker code
self.onmessage = function(event){
    var data = event.data;

    //do something with the data
};
```

Keep in mind that `self` in this example is actually a reference to the worker inside of the worker global scope (a different object than the instance of `Worker` inside the page). Once the worker has finished processing, data can be sent back to the page by calling `postMessage()` as well. For example, the following assumes that an array of numbers is passed in and needs to be sorted. The sorted array is then passed back to the page:



Available for
download on
Wrox.com

```
//inside worker code
self.onmessage = function(event){
    var data = event.data;

    //remember, by default sort() does a string comparison
    data.sort(function(a, b){
        return a - b;
    });
    self.postMessage(data);
};
```

[WebWorkerExample01.js](#)

Messaging data back and forth is how a page and a worker communicate with one another. Calling `postMessage()` inside the worker results in an asynchronous message event firing on the instance of `Worker` in the page. If a page wanted to use this worker, it would do so as follows:

```
//in the page
var data = [23, 4, 7, 9, 2, 14, 6, 651, 87, 41, 7798, 24],
    worker = new Worker("WebWorkerExample01.js");

worker.onmessage = function(event){
```

```

var data = event.data;

//do something with the resulting array
};

//send the array to the worker for sorting
worker.postMessage(data);

```

[WebWorkerExample01.htm](#)

Sorting is exactly the kind of time-intensive operation that may be useful to offload into a worker so as not to block the user interface. Other examples are image processing, such as converting an image to grayscale, and cryptographic operations.

A worker may also stop itself completely at any time by calling the `close()` method. This is similar to the `terminate()` method that can be called from the page in that no further events are fired:

```

//inside worker code
self.close();

```

Including Other Scripts

Without the ability to create a new `<script>` element dynamically, it may seem impossible to add new scripts into a worker. Luckily, the worker global scope takes this into account and provides a method called `importScripts()`. This method accepts one or more URLs from which to load JavaScript. All of the loading is done synchronously, so code after `importScripts()` isn't executed until after all of the scripts have been loaded and executed. For example:

```

//inside worker code
importScripts("file1.js", "file2.js");

```

Even though `file2.js` may finish downloading before `file1.js`, they will be executed in the order in which they are specified. The scripts are executed in the worker global scope, so if they make use of page-specific JavaScript, then they may not work in a worker. Typically, worker code is a highly specialized piece of code rather than something that is shared with a page.

The Future of Web Workers

There's a lot of work still being done on the Web Workers specification. The workers discussed in this section are now called *dedicated workers*, in that they are dedicated to a particular page and cannot be shared. The specification introduced a concept of *shared workers*, where a single worker may be shared by the same page opened in multiple tabs in a browser. While Safari 5, Chrome, and Opera 10.6 support shared workers, the specification is not yet final and may or may not undergo changes.

Debates over what should and should not be accessible inside of workers also continue. Some believe that workers should have access to every data store that the page has access to, meaning that the workers should have access to not just XHR but also `localStorage`, `sessionStorage`, Indexed DB, Web Sockets, Server-Sent Events, and so on. There seems to be significant support in this direction, so there will likely be some changes made to allow more in the worker global scope.

SUMMARY

Alongside HTML5 is a collection of JavaScript APIs that, while technically not part of the specification, tend to be lumped in with the HTML5 JavaScript APIs. Many of these APIs are still being defined but have gained cross-browser support such that they are worth talking about now.

- `requestAnimationFrame()` seeks to optimize JavaScript-based animations by signaling when an animation is running. This allows the browser to optimize the screen redraws.
- The Page Visibility API gives you insights into when the user is viewing the page and when the page is hidden from view.
- The Geolocation API is a way to determine, with permission, the user's location. This functionality is very popular for mobile web applications.
- The File API allows JavaScript to read data from files to either display, process, or upload those files. Combined with the HTML5 drag-and-drop functionality, you can easily create drag-and-drop file uploads.
- Web Timing gives you valuable performance insights into page load and render times.
- Web Workers allow you to run asynchronous JavaScript that will not block the user interface. This is very useful for complex calculations and data processing that would otherwise take up a lot of time and interfere with the user's ability to use the page.

A

ECMAScript Harmony

With the renewed interest in web development since 2004, conversations began taking place among browser vendors and other interested parties as to how JavaScript should evolve. Work on the fourth edition of ECMA-262 began based largely on two competing proposals: one for Netscape's JavaScript 2.0 and the other for Microsoft's JScript.NET. Instead of competing in the browser realm, the parties converged back into ECMA to hammer out a proposal for a new language based on JavaScript. Initially, work began on a proposal called ECMAScript 4, and for a long time, this seemed like the next evolutionary step for JavaScript. When a counterproposal called ECMAScript 3.1 was later introduced, it threw the future of JavaScript into question. After much debate, it was determined that ECMAScript 3.1 would be the next step for JavaScript and that a further effort, code-named Harmony, would seek to reconcile some features from ECMAScript 4 into ECMAScript 3.1.

ECMAScript 3.1 was ultimately renamed to ECMAScript 5 and standardized fairly quickly. The details of ECMAScript 5 have been covered throughout this book. As soon as ECMAScript 5 was finalized, work immediately began on Harmony. Harmony tries to keep to the spirit of ECMAScript 5, in making more incremental changes rather than radical language changes. While the details of Harmony, aka ECMAScript 6, are still developing as of 2011, there are several parts of the specification that have been finished. This appendix covers the parts of Harmony that will definitely make it into the final specification, though, keep in mind that the details of the final implementations may change from what's presented here.

GENERAL CHANGES

Harmony introduces several basic changes to ECMAScript. These aren't major changes for the language but rather the closing of some of the curiously open gaps in functionality.

Constants

One of the glaring weaknesses of JavaScript is its lack of formal constants. To rectify this, developers added constants as part of Harmony via the `const` keyword. Used in a manner similar to `var`, the `const` declaration lets you define a variable whose value cannot be changed once initialized. Here is the usage:

```
const MAX_SIZE = 25;
```

Constants may be defined anywhere a variable can be defined. Constant names cannot be the same as variable or function names declared in the same scope, so the following causes an error:

```
const FLAG = true;
var FLAG = false;      //error!
```

Aside from having immutable values, constants can be used just like any other variable. Any attempt to change the value is simply ignored, as shown here:

```
const FLAG = true;
FLAG = false;
alert(FLAG);      //true
```

Constants are supported in Firefox, Safari 3+, Opera 9+, and Chrome. In Safari and Opera, `const` acts just like `var` in that values can still be changed.

Block-Level and Other Scopes

One of the constant reminders throughout this book has been that JavaScript has no concept of block-level scope. This means that variables defined inside statement blocks act as if they were defined in the containing function. Harmony introduces the concept of block-level scoping through the introduction of the `let` keyword.

Similarly to `const` and `var`, a `let` declaration can be used at any point to define a variable and initialize its value. The difference is that the variable defined with `let` will disappear once execution has moved outside the block in which it was defined. For example, it's quite common to use the following construct:

```
for (var i=0; i < 10; i++) {
    //do something
}

alert(i);      //10
```

When the variable `i` is declared in this code, it is declared as local to the function in which the code resides. This means that the variable is still accessible after the `for` loop has finished executing. If `let` were used instead of `var`, the variable `i` would not exist after the loop completed. Consider the following:

```
for (let i=0; i < 10; i++) {
    //do something
}

alert(i);      //Error! i is undefined
```

If this code were to be executed, the last line would cause an error since the definition of `i` is removed as soon as the `for` loop completes. The result is an error, because you cannot perform any operations on an undeclared variable.

There are other ways to use `let` as well. You can create a `let` statement that specifically defines variables that should be used only with the next block of code, as in this example:

```
var num = 5;

let (num=10, multiplier=2){
    alert(num * multiplier);      //20
}

alert(num);      //5
```

In this code, the `let` statement defines an area within which the `num` variable is equal to 10 and the `multiplier` variable is equal to 2. This definition of `num` overrides the previously declared value using `var`, so within the `let` statement the result of multiplying by the `multiplier` is 20. Outside the `let` statement, the value of `num` remains 5. Since each `let` statement creates its own scope, the variable values inside it have no bearing on the values outside.

You can use a similar syntax to create a `let` expression where variable values are set only for a single expression. Here is an example:

```
var result = let(num=10, multiplier=2) num * multiplier;
alert(result); //20
```

Here, a `let` expression is used to calculate a value using two variables. The value is then stored in the `result` variable. After that point, the variables `num` and `multiplier` no longer exist.

Using block-level scopes in JavaScript gives you more control over which variables exist at what point during code execution.

FUNCTIONS

Most code is written in functions, so Harmony focuses on ways to improve functions and make them easier to use. As with other parts of Harmony, these changes focus on pain points for developers and implementers.

Rest and Spread Arguments

In Harmony, the `arguments` object is no more; you can't access undeclared arguments in it at all. There is, however, a way to indicate that you are expecting a variable number of arguments to be passed in through the use of *rest arguments*. Rest arguments are indicated by three dots followed by an identifier. This allows you to define the arguments that you know will be passed in and then collect the rest into an array. Here is an example:

```
function sum(num1, num2, ...nums) {
  var result = num1 + num2;
  for (let i=0, len=nums.length; i < len; i++) {
    result += nums[i];
  }
  return result;
}

var result = sum(1, 2, 3, 4, 5, 6);
```

This code defines a `sum()` method that accepts at least two arguments. It can accept additional arguments, and all of the remaining arguments are stored in the `nums` array. Unlike the `arguments` object, rest arguments are stored in an instance of `Array`, so all array methods are available. The `rest arguments` object is always an instance of `Array`, even if there are no rest arguments passed into the function.

Closely related to rest arguments are *spread arguments*. Spread arguments allow you to pass in an array and have each item be mapped to a particular argument in the function. The notation for spread arguments is the same as rest arguments, prepending three dots to a value. The only difference is that spread arguments take place at the time a function is called, whereas rest arguments are used at the time a function is defined. For example, instead of passing in individual numbers to the `sum()` method, you could use spread arguments as shown here:

```
var result = sum(...[1, 2, 3, 4, 5, 6]);
```

In this code, an array of arguments is passed to `sum()` as spread arguments. This example is the functional equivalent of the following:

```
var result = sum.apply(this, [1, 2, 3, 4, 5, 6]);
```

Default Argument Values

All arguments in an ECMAScript function are considered optional, since no check is done against the number of arguments that have been passed in. However, instead of manually checking to see which arguments have been provided, you can specify default values for arguments. If the arguments aren't formally passed in, then they get the given value.

To specify a default value for an argument, just add an equal sign and the default value after the argument definition, as in this example:

```
function sum(num1, num2=0) {
  return num1 + num2;
}

var result1 = sum(5);
var result2 = sum(5, 5);
```

The `sum()` function accepts two arguments, but the second one is optional and gets a default value of 0. The beauty of optional arguments is that it frees you from needing to check to see if the value was passed in and then using a special value; all of that is done for you.

Generators

A *generator* is an object that generates a sequence of values one at a time. With Harmony, you can create a generator by defining a function that returns a specific value using the `yield` operator. When a function is called that uses `yield`, a new `Generator` instance is created and returned. The `next()` method can then be called to retrieve the first value of the generator. When this happens, the original function is executed and stops execution when it comes to `yield`, returning the specified value. In this way, `yield` works in a similar manner to `return`. If `next()` is called again, code execution continues at the next statement following `yield` and then continues to run until `yield` is encountered again, at which point a new value is returned. Here is an example:

```
function myNumbers(){
    for (var i=0; i < 10; i++){
        yield i * 2;
    }
}

var generator = myNumbers();

try {
    while(true){
        document.write(generator.next() + "<br />");
    }
} catch(ex){
    //intentionally blank
} finally {
    generator.close();
}
```

When the function `myNumbers()` is called, a generator is returned. The `myNumbers()` function itself is very simple, containing a `for` loop that yields a value each time through the loop. Each call to `next()` causes another trip through the `for` loop and returns the next value. The first value is 0, the second is 2, the third is 4, and so on. When `myNumbers()` completes without calling `yield` (after the final loop iteration), calling `next()` throws a `StopIteration` error. So to output all numbers in the generator, a `while` loop is wrapped in a `try-catch` statement to prevent the error from stopping code execution.

If a generator is no longer needed, it's best to call the `close()` method. Doing so ensures that the rest of the original function is executed, including any `finally` blocks related to `try-catch` statements.

Generators are useful when a sequence of values needs to be produced and each subsequent value is somehow related to the previous one.

ARRAYS AND OTHER STRUCTURES

Another area of focus for Harmony is arrays. Arrays are one of the most frequently used data structures in JavaScript, and creating more intuitive and powerful ways to work with arrays was a priority for the language.

Iterators

An *iterator* is an object that iterates over a sequence of values and returns them one at a time. When you use a `for` loop or a `for-in` loop, you're typically iterating over values and processing them one at a time. Iterators provide the ability to do the same without using a loop. Harmony supports iterators for all types of objects.

To create an iterator for an object, use the `Iterator` constructor and pass in the object whose values should be iterated over. The `next()` method is used to retrieve the next value in the sequence. By default, this method returns an array whose first item is the index of the value (for arrays) or the name of the property (for objects) and whose second item is the value. When no further values are available, calling `next()` throws a `StopIteration` error. Here is an example:

```
var person = {
    name: "Nicholas",
    age: 29
};
var iterator = new Iterator(person);

try {
    while(true){
        let value = iterator.next();
        document.write(value.join(":") + "<br>");
    }
} catch(ex) {
    //intentionally blank
}
```

This code creates an iterator for the `person` object. The first time `next()` is called, the array `["name", "Nicholas"]` is returned, and the second call returns `["age", 29]`. The output from this code is as follows:

```
name:Nicholas
age:29
```

When an iterator is created for a nonarray object, the properties are returned in the same order as they would be in a `for-in` loop. This also means that only instance properties are returned, and the order in which the properties are returned varies upon implementation. Iterators created for arrays act in a similar manner, iterating over each position in the array, as shown here:

```
var colors = ["red", "green", "blue"];
var iterator = new Iterator(colors);

try {
    while(true){
        let value = iterator.next();
        document.write(value.join(":") + "<br>");
    }
} catch(ex) {
```

The output from this code is as follows:

```
0:red
1:green
2:blue
```

You can force only the property name or index to be returned from `next()` by passing a second argument, `true`, into the `Iterator` constructor, as shown here:

```
var iterator = new Iterator(colors, true);
```

With the second argument passed, each call to `next()` will return only the index of the value instead of an array containing both the index and the value.



It's possible to create your own iterators for custom types by defining the special method `__iterator__()`, which must return an object that has a `next()` method. This method will be called when an instance of your custom type is passed as an argument to the `Iterator` constructor.

Array Comprehensions

Array comprehensions are a way to initialize an array with specific values meeting certain criteria. This feature, introduced in Harmony, is a popular language construct in Python. The basic form of array comprehensions in JavaScript is as follows:

```
array = [ value for each (variable in values) condition ];
```

The `value` is the actual value to be included in the final array. This value is based on all the values in the `values` array. The `for each` construct loops over each value in `values` and stores the value in `variable`. If the optional `condition` is met, then `value` is added to the resulting array. Here is an example:

```
//original array
var numbers = [0,1,2,3,4,5,6,7,8,9,10];

//just copy all items into a new array
var duplicate = [i for each (i in numbers)];

//get just the even numbers
var evens = [i for each (i in numbers) if (i % 2 == 0)];

//multiply every value by 2
var doubled = [i*2 for each (i in numbers)];

//multiply every odd number by 3
var tripledOdds = [i*3 for each (i in numbers) if (i % 2 > 0)];
```

All of the array comprehensions in this code use `i` as a variable to iterate over all values in `numbers`. Some of them use conditions to filter the results of the array. Essentially, if the condition evaluates to `true`, the value is added to the array. The syntax is a little different from traditional JavaScript but is more succinct than writing your own `for` loop to accomplish the same task. Firefox (version 2 and later) is the only browser to implement this feature, and it requires the `type` attribute of the `<script>` element to be `"application/javascript;version=1.7"` to enable it.



The values portion of an array comprehension can also be a generator or an iterator.

Destructuring Assignments

It's quite common to have a group of values from which you want to extract one or more into individual variables. Consider the value returned from an iterator's `next()` method, which is an array containing the property name and value. In order to store each in its own variable, it would require two statements, as in this example:

```
var nextValue = ["color", "red"];
var name = nextValue[0];
var value = nextValue[1];
```

A *destructuring assignment* allows you to assign both array items into variables using a single statement such as this:

```
var [name, value] = ["color", "red"];
alert(name);      // "color"
alert(value);    // "red"
```

In traditional JavaScript syntax, an array literal cannot be on the left side of an assignment. Destructuring assignment introduces this syntax to indicate that the variables contained in the array to the left of the equal sign should be assigned the values contained in the array to the right of the equal sign. The result is that `name` is filled with `"color"` and `value` is filled with `"red"`.

If you don't want all of the values, you can provide variables just for the ones you want, as in this example:

```
var [, value] = ["color", "red"];
alert(value);    // "red"
```

Here, only the variable `value` is assigned, and it receives the value `"red"`.

You can use destructuring assignment in creative ways, such as to swap the values of two variables. In ECMAScript 3, swapping the values of two variables is typically done like this:

```
var value1 = 5;
var value2 = 10;

var temp = value1;
```

```
value1 = value2;
value2 = temp;
```

You can eliminate the need for the `temp` variable by using a destructuring array assignment, as in this example:

```
var value1 = 5;
var value2 = 10;

[value2, value1] = [value1, value2];
```

Destructuring assignment can also be accomplished with objects, like this:

```
var person = {
    name: "Nicholas",
    age: 29
};

var { name: personName, age: personAge } = person;

alert(personName); // "Nicholas"
alert(personAge); // 29
```

As with array literals, when an object literal occurs to the left of an equal sign, it's considered to be a destructuring assignment. This statement actually defines two variables, `personName` and `personAge`, that are filled with the matching information from the variable `person`. As with arrays, you can pick and choose which values to retrieve, as shown here:

```
var { age: personAge } = person;
alert(personAge); // 29
```

This modified code retrieves only the `age` property from the `person` object.

NEW OBJECT TYPES

Harmony introduces several new object types to the language. These object types focus on providing functionality that was previously available only to the JavaScript engine.

Proxy Objects

Harmony introduces the concept of proxies to JavaScript. A *proxy* is an object that presents an interface that doesn't necessarily act on the proxy object itself. For example, setting a property on a proxy object might actually call a function on another object. Proxies are a useful abstraction mechanism for exposing only a subset of information through an API while maintaining complete control over the data source.

A proxy object is created using the `Proxy.create()` method and passing in a handler object and optional `prototype` object:

```
var proxy = Proxy.create(handler);

//create proxy that has a prototype of myObject
var proxy = Proxy.create(handler, myObject);
```

The `handler` object comprises properties that define *traps*. Traps are functions that handle (trap) native functionality so that it can be handled in another way. There are seven *fundamental traps* that are considered important to implement for all proxies to ensure that the proxy object works in a predictable way without throwing errors:

- `getOwnPropertyDescriptor` — A function to call when `Object.getOwnPropertyDescriptor()` is called on the proxy. The function receives the property name as an argument. Should return a property descriptor or `null` if the property doesn't exist.
- `getPropertyDescriptor` — A function to call when `Object.getPropertyDescriptor()` is called on the proxy. (This is a new method in Harmony.) The function receives the property name as an argument. Should return a property descriptor or `null` if the property doesn't exist.
- `getOwnPropertyNames` — A function to call when `Object.getOwnPropertyNames()` is called on the proxy. The function receives the property name as an argument. Should return an array of strings.
- `getPropertyNames` — A function to call when `Object.getPropertyNames()` is called on the proxy. (This is a new method in Harmony.) The function receives the property name as an argument. Should return an array of strings.
- `defineProperty` — A function to call when `Object.defineProperty()` is called on the proxy. The function receives the property name and the property descriptor as arguments.
- `delete` — Defines a function that is called when the `delete` operator is used on a property of the object. The property name is passed in as an argument. Return `true` to indicate that the deletion succeeded or `false` if not.
- `fix` — Defines a function that is called when `Object.freeze()`, `Object.seal()`, or `Object.preventExtensions()` is called. Return `undefined` to throw an error when one of these methods is called on the proxy.

In addition to the fundamental traps, there are also six *derived traps*. Unlike fundamental traps, failing to define one or more derived traps will not cause errors. Each derived trap overrides a default JavaScript behavior.

- `has` — Defines a function that is called when the `in` operator is used on the object, such as "name" `in` `object`. The property name is passed in as an argument. Return `true` to indicate the property is contained on the object, `false` if not.
- `hasOwn` — Defines a function that is called when the `hasOwnProperty()` method is called on the proxy. The property name is passed in as an argument. Return `true` to indicate the property is contained on the object, `false` if not.

- `get` — Defines a function that is called when a property is read. The function receives two arguments, the object reference being read from and the property name. The object reference may be the proxy itself or may be an object inheriting from the proxy.
- `set` — Defines a function that is called when a property is written to. The function receives three arguments, the object reference being written to, the property name, and the property value. As with `get`, the object reference may be the proxy itself or may be an object inheriting from the proxy.
- `enumerate` — Defines a function that is called when the proxy is placed in a `for-in` loop. The function must return an array of strings containing the appropriate property names to be used in the `for-in` loop.
- `keys` — Defines a function that is called when `Object.keys()` is called on the proxy. As with `enumerate`, this function must return an array of strings.

Proxies are primarily used when you need to expose an API while keeping some underlying data from being manipulated directly. For example, suppose you want to implement a traditional stack data type. Even though arrays can act as stacks, you want to ensure that people use only `push()`, `pop()`, and `length`. In this case, you can create a proxy that works on an array but exposes only those three object members:

```
/*
 * Another ES6 Proxy experiment. This one creates a stack whose underlying
 * implementation is an array. The proxy is used to filter out everything
 * but "push", "pop", and "length" from the interface, making it a pure
 * stack where you can't manipulate the contents.
 */

var Stack = (function(){

    var stack = [],
        allowed = [ "push", "pop", "length" ];

    return Proxy.create({
        get: function(receiver, name){
            if (allowed.indexOf(name) > -1){
                if(typeof stack[name] == "function"){
                    return stack[name].bind(stack);
                } else {
                    return stack[name];
                }
            } else {
                return undefined;
            }
        }
    });

});

var mystack = new Stack();
mystack.push("hi");
```

```
mystack.push("goodbye");

console.log(mystack.length);      //1

console.log(mystack[0]);         //undefined
console.log(mystack.pop());      //"goodbye"
```

This code creates a constructor called `Stack`. Instead of working on `this`, the `Stack` constructor returns a proxy object that works on an array. The `get` trap is the only one defined, and it simply checks an array of allowed properties before returning the value. All disallowed properties end up returning `undefined` when referenced while `push()`, `pop()`, and `length` work as expected. The key piece of this code is the declaration of the `get` trap, which filters the object member retrieval based on the allowed members. If the member is a function, then it returns a bound version of the function so that it operates on the underlying array object instead of the proxy object itself.

Proxy Functions

In addition to creating proxy objects, you can also create *proxy functions* in Harmony. A proxy function is the same as a proxy object except that it is executable. Proxy functions are created by using the `Proxy.createFunction()` method and passing in a `handler` object, a call trap function, and an optional constructor trap function. For example:

```
var proxy = Proxy.createFunction(handler, function(){}, function(){});
```

The `handler` object has the same available traps as with proxy objects. The call trap function is the code to execute when the proxy function is executed, such as `proxy()`. The constructor trap is the code to execute when the proxy function is called using the `new` operator, such as `new proxy()`. If the constructor trap is not defined, then the call trap is used for the constructor trap as well.

Map and Set

The `Map` type, also called *simple map*, has a singular purpose: to store a list of key-value pairs. Developers typically use generic objects for this purpose, but that comes at a cost as keys can easily be confused with native properties. Simple maps keep keys and values separate from the object's properties to provide safer storage for this information. Some example usage:

```
var map = new Map();

map.set("name", "Nicholas");
map.set("book", "Professional JavaScript");

console.log(map.has("name"));    //true
console.log(map.get("name"));    //"Nicholas"

map.delete("name");
```

The basic API for simple maps is made up of `get()`, `set()`, `has()`, and `delete()`, each doing exactly what the name indicates. Keys can be primitive values for reference values.

Related to simple maps is the `Set` type. A set is simply a collection of items of which there are no duplicates. Unlike simple maps, sets are only keys and have no related value associated with them. The basic API has `add()` for adding items, `has()` for checking the existence of items, and `delete()` for removing items. Some example usage:

```
var set = new Set();
set.add("name");

console.log(set.has("name")); //true
set.delete("name");

console.log(set.has("name")); //false
```

The specification for both `Map` and `Set` are incomplete as of October 2011, so the details here may change before the JavaScript engines begin implementing them.

WeakMap

The `WeakMap` type is interesting in that it's the first ECMAScript feature that allows you to know when an object has been completely dereferenced. A `WeakMap` works in a similar manner to a simple map where you store a key-value pair. The big difference for a `WeakMap` is that the key must be an object and when the object no longer exists, the associated key-value pair is removed from the `WeakMap`. For example:

```
var key = {},
    map = new WeakMap();

map.set(key, "Hello!");

//dereference the key so the value is also removed
key = null;
```

The use case for `WeakMaps` is as yet unclear, but this construct also appears in Java as `WeakHashMap` and so this brings another data structure option to JavaScript.

StructType

One of the acknowledged downsides of JavaScript is the use of a single data type to represent all numbers. WebGL introduced typed arrays to help this problem, and ECMAScript 6 introduces typed structures to further integrate more numeric data types into the language. A *struct type* is analogous to a struct in C, where you combine multiple properties into a single record. Struct types in JavaScript allow you to create similar data structures by specifying properties and the type of data they contain. The initial implementation defines several block types:

- `uint8` — unsigned 8-bit integer
- `int8` — signed 8-bit integer
- `uint16` — unsigned 16-bit integer
- `int16` — signed 16-bit integer

- `uint32` — unsigned 32-bit integer
- `int32` — signed 32-bit integer
- `float32` — 32-bit floating point
- `float64` — 64-bit floating point

Block types contain a single value, and there are expected to be more than these initial eight added in the future.

A struct type is created by instantiating `StructType` with property definitions in the form of an object literal. For example:

```
var Size = new StructType({ width: uint32, height: uint32 });
```

This code creates a new struct type called `Size` with two properties: `width` and `height`. Each property holds an unsigned 32-bit integer. The variable `Size` is actually a constructor that can be used just like an object constructor. You initialize an instance of a struct type by passing an object literal with the property values into the constructor:

```
var boxSize = new Size({ width: 80, height: 60 });
console.log(boxSize.width); //80
console.log(boxSize.height); //60
```

Here, a new instance of `Size` is created with a `width` of 80 and a `height` of 60. These properties can be written to and read from but always must contain 32-bit unsigned integers.

It's possible to create more complex struct types by having each property defined as another struct type. For example:

```
var Location = new StructType({ x: int32, y: int32 });
var Box = new StructType({ size: Size, location: Location });

var boxInfo = new Box({ size: { width: 80, height: 60 }, location: { x: 0, y: 0 } });
console.log(boxInfo.size.width); //80
```

This example creates a simple struct type called `Location` and a complex struct type called `Box`, whose properties are defined as struct types. The `Box` constructor still accepts an object literal defining the values for each property and will type-check to ensure the values are the correct data type.

ArrayType

Closely related to struct type is array type. An array type allows creation of an array whose values are limited to a specific type, very similar to WebGL typed arrays. To create a new array type, call the `ArrayType` constructor and pass in the type of data it should hold and how many items will be in the array. For example:

```
var SizeArray = new ArrayType(Size, 2);
var boxes = new BoxArray([ { width: 80, height: 60 }, { width: 50, height: 50 } ]);
```

This code creates a new array type called `SizeArray` that is initialized to hold instances of `Size` with an allotment of two spaces in the array. Array types are initialized by passing in an array containing the data that should be converted, allowing literals to be passed in and coerced into the correct data type (as with struct types).

CLASSES

Developers have long clamored for an easy way to define Java-like *classes* in JavaScript, and ECMAScript 6 finally introduces this functionality into the language. Classes are syntactic sugar that overlay the current constructor- and prototype-based approach to types. Consider the following type definition:

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

Person.prototype.sayName = function() {
    alert(this.name);
};

Person.prototype.getOlder = function(years) {
    this.age += years;
};
```

The equivalent using the new class syntax is:

```
class Person {

    constructor(name, age) {
        public name = name;
        public age = age;
    }

    sayName() {
        alert(this.name);
    }

    getOlder(years) {
        this.age += years;
    }
}
```

The new class syntax begins with the keyword `class` followed by the name of the type. Inside the braces is where properties and methods are created. Methods no longer require the `function` keyword; instead just use the name of the method followed by parentheses. If the method is named `constructor`, then it acts as the constructor function for the class (the same as the `Person` function in the prior code). All other methods and properties defined within the class braces are applied to the prototype, so in this case, `sayName()` and `getOlder()` both end up on `Person.prototype`.

Within the constructor function, variables preceded by the keywords `public` or `private` are created as instance properties of the object. Both `name` and `age` are defined as public properties in this example.

Private Members

The classes proposal supports private members by default, both on the instance and on the prototype. The `private` keyword indicates that a member is private and cannot be accessed from outside of a class's methods. In order to access private members, a special syntax is used where the `private()` function is called on this and then the property may be accessed. For example, the following changes the `Person` class to have a private `age` property.

```
class Person {  
  
    constructor(name, age){  
        public name = name;  
        private age = age;  
    }  
  
    sayName(){  
        alert(this.name);  
    }  
  
    getOlder(years){  
        private(this).age += years;  
    }  
  
}
```

The syntax for accessing private properties is still being debated and will likely change in the future.

Getters/Setters

The new class syntax allows you to define getters and setters for properties directly, avoiding the extra step of calling `Object.defineProperty()`. The syntax is the same as for methods but has a preceding `get` or `set` keyword. For example:

```
class Person {  
  
    constructor(name, age){  
        public name = name;  
        public age = age;  
        private innerTitle = "";  
  
        get title(){  
            return innerTitle;  
        }  
  
        set title(value){  
            innerTitle = value;  
        }  
    }  
}
```

```

    }

    sayName() {
        alert(this.name);
    }

    getOlder(years) {
        this.age += years;
    }

}

```

This version of the `Person` class defines a getter and a setter for the `title` property. Each function operates on the `innerTitle` variable that is defined in the constructor. Getters and setters for prototype properties are also possible by using the same syntax outside of the constructor function.

Inheritance

A key advantage of using classes over the more traditional JavaScript syntax is the ease with which inheritance is achieved. Instead of worrying about constructor stealing and prototype chaining, a simple syntax that is shared with other languages is used: the `extends` keyword. For example:

```

class Employee extends Person {
    constructor(name, age) {
        super(name, age);
    }
}

```

This code creates a new `Employee` class as a subclass of `Person`. The prototype chaining occurs behind the scenes and constructor stealing is now formally supported by using the `super()` function. The preceding code is the logical equivalent of the following:

```

function Employee(name, age) {
    Person.call(this, name, age);
}

Employee.prototype = new Person();

```

In addition to this style of inheritance, classes can also specify an object to assign directly as its prototype by using the `prototype` keyword in place of `extends`:

```

var basePerson = {
    sayName: function(){
        alert(this.name);
    },
    getOlder: function(years){
        this.age += years;
    }
};

class Employee prototype basePerson {

```

```
constructor(name, age) {
    public name = name;
    public age = age;
}
}
```

In this example, `Employee.prototype` is assigned to the `basePerson` object directly, allowing for the same type of inheritance currently achieved through `Object.create()`.

MODULES

Modules (or *namespaces* or *packages*) are a popular concept in organizing JavaScript applications. Each module contains specific, unique functionality that is self-contained and separable from other modules. Though several ad hoc module formats have emerged in JavaScript, ECMAScript 6 seeks to formalize how modules are created and managed.

Modules operate in their own top-level execution context and so cannot pollute the global execution context in which they're imported. By default, all variables, functions, classes, and so on declared within a module are private to that module. You indicate that a member should be exposed to the outside world by using the `export` keyword in front of it. For example:

```
module MyModule {
    //export this stuff
    export let myobject = {};
    export function hello(){ alert("hello"); }

    //keep this stuff hidden
    function goodbye(){
        ...
    }
}
```

This module exports an object called `myobject` and a function called `hello()`. The module is used elsewhere, in a page or in another module, but imports one or both of the available members. Importation is accomplished using the `import` command:

```
//import just myobject
import myobject from MyModule;
console.log(myobject);

//import everything
import * from MyModule;
console.log(myobject);
console.log(hello);

//explicitly named imports
import {myobject, hello} from MyModule;
console.log(myobject);
console.log(hello);

//no import - use module directly
```

```
console.log(MyModule.myobject);  
console.log(MyModule.hello);
```

If the execution context has access to a module, then the module may be accessed directly to get the members that it's exporting. The process of importing simply brings individual members into the current execution context so that they may be accessed without referencing the module directly.

External Modules

Modules may be included dynamically by providing a URL from which the module is to be loaded. To do so, just add the URL after the module declaration:

```
module MyModule from "mymodule.js";  
import myobject from MyModule;
```

This declaration instructs the JavaScript engine to download `mymodule.js` and load the module named `MyModule` from it. Note that this call is blocking — the JavaScript engine will not continue processing code until the URL is downloaded and evaluated.

If you want to include something that the module exports without bringing in the entire module, you can specify that using the `import` directive:

```
import myobject from "mymodule.js";
```

Overall, modules are just a way to group together related functionality and protect the global scope from pollution.

B

Strict Mode

ECMAScript 5 was the first to introduce the concept of *strict mode*. Strict mode allows you to opt-in to stricter checking for JavaScript error conditions either globally or locally within a single function. The advantage of strict mode is that you'll be informed of errors earlier, so some of the ECMAScript quirks that cause programming errors will be caught immediately.

The rules of strict mode are important to understand, as the next version of ECMAScript will start with a base of strict mode. Strict mode is supported in Internet Explorer 10+, Firefox 4+, Safari 5.1+, and Chrome.

OPTING-IN

To opt-in to strict mode, use the strict mode *pragma*, which is simply a string that isn't assigned to any variable:

```
"use strict";
```

Using this syntax, which is valid even in ECMAScript 3, allows seamless fallback for JavaScript engines that don't support strict mode. The engines that support strict mode will enable it, while engines that don't will simply ignore the pragma as an unassigned string literal.

When the pragma is applied globally, outside of a function, strict mode is enabled for the entire script. That means adding the pragma to a single script that is concatenated with other scripts into a single file puts all JavaScript in the file into strict mode.

You can also turn on strict mode within a function only, such as:

```
function doSomething() {
    "use strict";

    //other processing
}
```

If you don't have complete control over all of the scripts on a page, then it's advisable to enable strict mode only within specific functions for which it has been tested.

VARIABLES

How and when variables get created is different in strict mode. The first change disallows accidental creation of global variables. In nonstrict mode, the following creates a global variable:

```
//Variable is not declared  
//Non-strict mode: creates a global  
//Strict mode: Throws a ReferenceError  
  
message = "Hello world!";
```

Even though `message` isn't preceded by the `var` keyword and isn't explicitly defined as a property of the global object, it is still automatically created as a global. In strict mode, assigning a value to an undeclared variable throws a `ReferenceError` when the code is executed.

A related change is the inability to call `delete` on a variable. Nonstrict mode allows this and may silently fail (returning `false`). In strict mode, an attempt to delete a variable causes an error:

```
//Deleting a variable  
//Non-strict mode: Fails silently  
//Strict mode: Throws a ReferenceError  
  
var color = "red";  
delete color;
```

Strict mode also imposes restrictions on variable names. Specifically, it disallows variables named `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static`, and `yield`. These are now reserved words that are intended for use in future ECMAScript editions. Any attempt to use these as variable names while in strict mode will result in a syntax error.

OBJECTS

In strict mode, object manipulation is more likely to throw errors than in nonstrict mode. Strict mode tends to throw errors in situations where nonstrict mode silently fails, increasing the likelihood of catching an error early on in development.

To begin, there are several cases where attempting to manipulate an object property will throw an error:

- Assigning a value to a read-only property throws a `TypeError`.
- Using `delete` on a nonconfigurable property throws a `TypeError`.
- Attempting to add a property to a nonextensible object throws a `TypeError`.

Another restriction on objects has to do with declaring them via object literals. When using an object literal, property names must be unique. For instance:

```
//Two properties with the same name
//Non-strict mode: No error, second property wins
//Strict mode: Throws a syntax error

var person = {
    name: "Nicholas",
    name: "Greg"
};
```

The object literal for `person` has two properties called `name` in this code. The second property is the one that ends up on `person` in nonstrict mode. In strict mode, this is a syntax error.

FUNCTIONS

First, strict mode requires that named function arguments be unique. Consider the following:

```
//Duplicate named arguments
//Non-strict mode: No error, only second argument works
//Strict mode: Throws a SyntaxError

function sum (num, num){
    //do something
}
```

In nonstrict mode, this function declaration doesn't throw an error. You'll be able to access the second `num` argument only by name while the first is accessible only through `arguments`.

The `arguments` object also has a slight behavior change in strict mode. In nonstrict mode, changes to a named argument are also reflected in the `arguments` object, whereas strict mode ensures that each are completely separate. For example:

```
//Change to named argument value
//Non-strict mode: Change is reflected in arguments
//Strict mode: Change is not reflected in arguments

function showValue(value) {
    value = "Foo";
    alert(value);           // "Foo"
    alert(arguments[0]);   //Non-strict mode: "Foo"
                           //Strict mode: "Hi"
}

showValue("Hi");
```

In this code, the function `showValue()` has a single named argument called `value`. The function is called with an argument of "Hi", which is assigned to `value`. Inside the function, `value` is changed to "Foo". In nonstrict mode, this also changes the value in `arguments[0]`, but in strict mode they are kept separate.

Another change is the elimination of `argumentscallee` and `argumentscaller`. In nonstrict mode, these refer to the function itself and the calling function, respectively. In strict mode, attempting to access either property throws a `TypeError`. For example:

```
//Attempt to access argumentscallee
//Non-strict mode: Works as expected
//Strict mode: Throws a TypeError

function factorial(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * argumentscallee(num-1)
    }
}

var result = factorial(5);
```

Similarly, the `caller` and `arguments` properties of a function now throw a `TypeError` when an attempt is made to read or write them. So in this example, attempts to access `factorial.caller` and `factorial.arguments` would also throw an error.

Also, as with variables, strict mode imposes restrictions on function names, disallowing functions named `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static`, and `yield`.

The last change to functions is disallowing function declarations unless they are at the top level of a script or function. That means functions declared, for instance, in an `if` statement are now a syntax error:

```
//Function declaration in an if statement
//Non-strict mode: Function hoisted outside of if statement
//Strict mode: Throws a syntax error

if (true){
    function doSomething(){
        //...
    }
}
```

This syntax is tolerated on all browsers in nonstrict mode but will now throw a syntax error in strict mode.

EVAL()

The much-maligned `eval()` function receives an upgrade in strict mode. The biggest change to `eval()` is that it will no longer create variables or functions in the containing context. For example:

```
//eval() used to create a variable
//Non-strict mode: Alert displays 10
//Strict mode: Throws an ReferenceError when alert(x) is called

function doSomething(){
    eval("var x=10");
    alert(x);
}
```

When run in nonstrict mode, this code creates a local variable `x` in the function `doSomething()` and that value is then displayed using `alert()`. In strict mode, the call to `eval()` does not create the variable `x` inside of `doSomething()` and so the call to `alert()` throws a `ReferenceError` because `x` is undeclared.

Variables and functions can be declared inside of `eval()`, but they remain inside a special scope that is used while code is being evaluated and then destroyed once completed. So the following code works without any errors:

```
"use strict";
var result = eval("var x=10, y=11;  x+y");
alert(result);      //21
```

The variables `x` and `y` are declared inside of `eval()` and are added together before returning their value. The `result` variable then contains 21, the result of adding `x` and `y`, even though `x` and `y` no longer exist by the time `alert()` is called.

EVAL AND ARGUMENTS

Strict mode now explicitly disallows using `eval` and `arguments` as identifiers and manipulating their values. For example:

```
//Redefining eval and arguments as variables
//Non-strict mode: Okay, no error.
//Strict-mode: Throws syntax error

var eval = 10;
var arguments = "Hello world!";
```

In nonstrict mode, you can overwrite `eval` and assign `arguments` to a value. In strict mode, this causes a syntax error. You can't use either as an identifier, which means all of the following use cases throw a syntax error:

- Declaration using `var`
- Assignment to another value
- Attempts to change the contained value, such as using `++`
- Used as function names

- Used as named function arguments
- Used as exception name in try-catch statement

COERCION OF THIS

One of the biggest security issues, and indeed one of the most confusing aspects of JavaScript, is how the value of `this` is coerced in certain situations. When using the `apply()` or `call()` methods of a function, a `null` or `undefined` value is coerced to the global object in nonstrict mode. In strict mode, the `this` value for a function is always used as specified, regardless of the value. For example:

```
//Access a property
//Non-strict mode: Accesses the global property
//Strict mode: Throws an error because this is null

var color = "red";

function displayColor() {
    alert(this.color);
}

displayColor.call(null);
```

This code passes `null` to `displayColor.call()`, which in nonstrict mode means the `this` value of the function is the global object. The result is an alert displaying "red". In strict mode, the `this` value of the function is `null`, so it throws an error when attempting to access a property of a `null` object.

OTHER CHANGES

There are several other changes to strict mode of which you need to be aware. The first is the elimination of the `with` statement. The `with` statement changes how identifiers are resolved and has been removed from strict mode as a simplification. An attempt to use `with` in strict mode results in a syntax error.

```
//Use of the with statement
//Non-strict mode: Allowed
//Strict mode: Throws a syntax error

with(location) {
    alert(href);
}
```

Strict mode also eliminates the *octal literal* from JavaScript. Octal literals begin with a zero and have traditionally been the source of many errors. An octal literal is now considered invalid syntax in strict mode.

```
//Use of octal literal  
//Non-strict mode: value is 8.  
//Strict mode: throws a syntax error.  
  
var value = 010;
```

As mentioned in the book, ECMAScript 5 also changed `parseInt()` for nonstrict mode, where octal literals are now considered decimal literals with a leading zero. For example:

```
//Use of octal literal in parseInt()  
//Non-strict mode: value is 8  
//Strict mode: value is 10  
  
var value = parseInt("010");
```


C

JavaScript Libraries

JavaScript *libraries* help to bridge the gap between browser differences and provide easier access to complex browser features. Libraries come in two forms: *general* and *specialty*. General JavaScript libraries provide access to common browser functionality and can be used as the basis for a web site or web application. Specialty libraries do only specific things and are intended to be used for only parts of a web site or web application. This appendix provides an overview of these libraries and some of their functionality, along with web sites that you can use as additional resources.

GENERAL LIBRARIES

General JavaScript libraries provide functionality that spans across topics. All general libraries seek to equalize browser interface and implementation differences by wrapping common functionality with new APIs. Some of the APIs look similar to native functionality, whereas others look completely different. General libraries typically provide interaction with the DOM, support for Ajax, and utility methods that aid in common tasks.

Yahoo! User Interface Library (YUI)

Yahoo! User Interface Library (YUI) is an open-source JavaScript and CSS library designed in an à la carte fashion. There isn't just one file for this library; instead there are multiple files provided in a variety of configurations, ensuring that you load only what you need. YUI covers all aspects of JavaScript, from basic utilities and helper functions to full-blown widgets. YUI has a dedicated team of software engineers at Yahoo! providing excellent documentation and support.

License: BSD License

Web site: www.yUILibrary.com

Prototype

Prototype is an open-source library that provides simple APIs for common web tasks. Originally developed for use in Ruby on Rails, Prototype is class-driven, aiming to provide class definition and inheritance for JavaScript. To that end, Prototype provides a number of classes that encapsulate common and complex functionality into simple API calls. As a single file, Prototype can be dropped into any page with ease. It is written and maintained by Sam Stephenson.

License: MIT License and Creative Commons Attribution-Share Alike 3.0 Unported

Web site: www.prototypejs.org/

The Dojo Toolkit

In the *Dojo Toolkit*, an open-source library modeled on a package system, groups of functionality are organized into packages that can be loaded on demand. Dojo provides a wide range of options and configurations, covering almost anything you want to do with JavaScript. The Dojo Toolkit was created by Alex Russell and is maintained by the employees and volunteers at the Dojo Foundation.

License: “New” BSD License or Academic Free License version 2.1

Web site: www.dojotoolkit.org/

MooTools

An open-source library designed to be compact and optimized, *MooTools* adds methods to native JavaScript objects to provide new functionality on familiar interfaces and to provide new objects. Its small size and simple API make MooTools a favorite among web developers.

License: MIT License

Web site: www.mootools.net/

jQuery

jQuery is an open-source library that provides a functional programming interface to JavaScript. It is a complete library whose core is built around using CSS selectors to work with DOM elements. Through call chaining, jQuery code looks more like a narrative description of what should happen rather than JavaScript code. This style of code has become popular among designers and prototypers. jQuery is written and maintained by John Resig.

License: MIT License or General Public License (GPL)

Web site: <http://jquery.com/>

MochiKit

An open-source library composed of several smaller utilities, *MochiKit* prides itself on being well-documented and well-tested, having a large amount of API and example documentation and hundreds of tests to ensure quality. MochiKit is written and maintained by Bob Ippolito.

License: MIT License or Academic Free License version 2.1

Web site: www.mochikit.com/

Underscore.js

While not strictly a general library, *Underscore.js* does provide some additional functionality for functional programming in JavaScript. The documentation talks about Underscore.js as a complement to jQuery, providing additional low-level functionality for working with objects, arrays, functions, and other JavaScript data types. Underscore.js is maintained by Jeremy Ashkenas of DocumentCloud.

License: MIT License

Web site: <http://documentcloud.github.com/underscore/>

INTERNET APPLICATIONS

Internet application libraries are designed to ease the development of an entire web application. Instead of providing small pieces to the application puzzle, they provide entire conceptual frameworks for rapid application development. Though these libraries may contain some low-level functionality, their goal is to help you develop web applications quickly.

Backbone.js

A minimal model-view-controller (MVC) open-source library built on top of Underscore.js, *Backbone.js* is optimized for single-page applications, allowing you to easily update parts of the page as application state changes. Backbone.js is maintained by Jeremy Ashkenas of DocumentCloud.

License: MIT License

Web site: <http://documentcloud.github.com/backbone/>

Rico

An open-source library that aims to make rich Internet-application development easier, *Rico* provides utilities for Ajax, animations, and styles and widgets. The library is maintained by a small team of volunteers, and development has slowed significantly as of 2008.

License: Apache License version 2.0

Web site: <http://openrico.org/>

qooxdoo

qooxdoo is an open-source library that aims to help with the entire web-application development cycle. qooxdoo implements its own versions of classes and interfaces to create a programming model similar to traditional object-oriented (OO) languages. The library includes a full GUI toolkit and compilers for simplifying the front-end build process. qooxdoo began as an internal library for the 1&1 web-hosting company (www.1and1.com) and later was released under an open-source license. 1&1 employs several full-time developers to maintain and develop the library.

License: GNU Lesser General Public License (LGPL) or Eclipse Public License (EPL)

Web site: www.qooxdoo.org/

ANIMATION AND EFFECTS

Animation and other visual effects have become a big part of web development. Getting smooth animation out of web pages is a nontrivial task, and several library developers have stepped up to provide easy-to-use and smooth animation and effects. Many of the general JavaScript libraries mentioned previously also feature animation.

script.aculo.us

A companion to Prototype, *script.aculo.us* provides easy access to cool animations using nothing more than CSS and the DOM. Prototype must be loaded before script.aculo.us can be used.

script.aculo.us is one of the most popular effects libraries, being used by major web sites and web applications around the world. The author, Thomas Fuchs, actively maintains script.aculo.us.

License: MIT License

Web site: <http://script.aculo.us/>

moo.fx

The *moo.fx* open-source animation library is designed to work on top of either Prototype or MooTools. Its goal is to be as small as possible (the latest version is 3KB) and to allow developers to create animations while writing as little code as necessary. moo.fx is included with MooTools by default; it can also be downloaded separately for use with Prototype.

License: MIT License

Web site: <http://moo.fx.mad4milk.net/>

Lightbox

Lightbox, a JavaScript library for creating simple image overlays on any page, requires both Prototype and script.aculo.us to create its visual effects. The basic idea is to allow users to view an image or a series of images in an overlay without leaving the current page. The “lightbox” overlay is customizable in both appearance and transitions. Lightbox is developed and maintained by Lokesh Dhakar.

License: Creative Commons Attribution 2.5 License

Web site: www.huddletogether.com/projects/lightbox2/

CRYPTOGRAPHY

As Ajax applications became more popular, there was an increasing need for cryptography on the browser to secure communications. Fortunately, several people have implemented common security algorithms in JavaScript. Most of these libraries aren’t officially supported by their authors but are used widely nonetheless.

JavaScript MD5

JavaScript MD5 is an open-source library that implements MD4, MD5, and SHA-1 secure hash functions. Author Paul Johnston and several other contributors have written this extensive library, one file per algorithm, for use in web applications. The home page gives an overview of hash functions and a brief discussion about vulnerabilities and appropriate uses.

License: BSD License

Web site: <http://pajhome.org.uk/crypt/md5/>

JavaScrypt

The *JavaScrypt* library implements MD5 and AES (256-bit) cryptography. JavaScrypt's web site offers lots of information about the history of cryptography and its usage in computers. Though lacking basic documentation about integrating the library into your web application, JavaScrypt's source code is full of advanced mathematical manipulations and computations.

License: Public domain

Web site: www.fourmilab.ch/javascrypt/

D

JavaScript Tools

Writing JavaScript is a lot like writing in any other programming language, with tools designed to make development easier. The number of tools available for JavaScript developers continues to grow, making it much easier to locate problems, optimize, and deploy JavaScript-based solutions. Some of the tools are designed to be used from JavaScript, whereas others can be run outside the browser. This appendix provides an overview of some of these tools, as well as additional resources for more information.

VALIDATORS

Part of the problem with JavaScript debugging is that there aren't many IDEs that automatically indicate syntax errors as you type. Most developers write some code and then load it into a browser to look for errors. You can significantly reduce the instances of such errors by validating your JavaScript code before deployment. *Validators* check basic syntax and provide warnings about style.

JSLint

JSLint is a JavaScript validator written by Douglas Crockford. It checks for syntax errors at a core level, going with the lowest common denominator for cross-browser issues. (It follows the strictest rules to ensure your code works everywhere.) You can enable Crockford's warnings about coding style, including code format, use of undeclared global variables, and more. Even though JSLint is written in JavaScript, it can be run on the command line through the Java-based Rhino interpreter, as well as through WScript and other JavaScript interpreters. The web site provides custom versions for each command-line interpreter.

Price: Free

Web site: www.jslint.com/

JSHint

JSHint is a fork of *JSLint* that provides more customization as to the rules that are applied. Like *JSLint*, it checks for syntax errors first and then looks for problematic coding patterns. Each *JSLint* check is also present in *JSHint*, but developers have better control over which rules to apply. Also similar to *JSLint*, *JSHint* can be run on the command line using Rhino.

Price: Free

Web site: www.jshint.com/

JavaScript Lint

Completely unrelated to *JSLint*, *JavaScript Lint* is a C-based JavaScript validator written by Matthias Miller. It uses SpiderMonkey, the JavaScript interpreter used by Firefox, to parse code and look for syntax errors. The tool has a fairly large collection of options that enable additional warnings about coding style, undeclared variables, and unreachable code. *JavaScript Lint* is available for both Windows and Macintosh, and the source code is available as well.

Price: Free

Web site: www.javascriptlint.com/

MINIFIERS

An important part of the JavaScript build process is crunching the output to remove excess characters. Doing so ensures that only the smallest number of bytes are transmitted to the browser for parsing and ultimately speeds up the user experience. There are several such *minifiers* available with varying compression ratios.

JSMin

JSMin is a C-based cruncher written by Douglas Crockford that does basic JavaScript compression. It primarily removes white space and comments to ensure that the resulting code can still be executed without issues. *JSMin* is available as a Windows executable with source code available in C and many other languages.

Price: Free

Web site: www.crockford.com/javascript/jsmin.html

Dojo ShrinkSafe

The same people responsible for the Dojo Toolkit have a tool called *ShrinkSafe*, which uses the Rhino JavaScript interpreter to first parse JavaScript code into a token stream and then use that to safely crunch the code. As with *JSMin*, *ShrinkSafe* removes extra white space (but not line breaks) and comments but also goes one step further and replaces the names of local variables with

two-character variable names instead. The result is smaller output than with JSMin without any risk of introducing syntax errors.

Price: Free

Web site: <http://shrinksafe.dojotoolkit.org/>

YUI Compressor

The YUI team has a cruncher called the *YUI Compressor*. Similar to ShrinkSafe, the YUI Compressor uses Rhino to parse JavaScript code into a token stream and then remove comments and white space and replace variable names. Unlike ShrinkSafe, the YUI Compressor also removes line breaks and performs several other micro-optimizations to save bytes here and there. Typically, files processed by the YUI Compressor are smaller than those processed with either JSMin or ShrinkSafe.

Price: Free

Web site: <http://yuilibrary.com/projects/yuicompressor>

UNIT TESTING

Test-driven development (TDD) is a software-development process built around the use of unit testing. Until recently, there weren't many tools for unit testing in JavaScript. Now, most JavaScript libraries use some form of unit testing on their own code, and some publish the unit-testing framework for others to use.

JsUnit

The original JavaScript unit-testing library is not tied to any particular JavaScript library. *JsUnit* is a port of the popular JUnit testing framework for Java. Tests are run in the page and may be set up for automatic testing and submission of results to a server. The web site contains examples and basic documentation.

Price: Free

Web site: www.jsunit.net/

YUI Test

Part of the Yahoo! User Interface Library (YUI), YUI Test can be used to test not only code using YUI but also any JavaScript on your site or application. YUI Test includes simple and complex assertions, as well as a way to simulate simple mouse and keyboard events. The framework is completely documented on the Yahoo! Developer Network, including examples, API documentation, and more. Tests are run in the browser, and results are output on the page. YUI uses YUI Test to test the entire library.

Price: Free

Web site: <http://yuilibrary.com/projects/yuitest/>

Dojo Object Harness (DOH)

The *Dojo Object Harness (DOH)* began as the internal unit-testing tool for Dojo before being released for everyone to use. As with the other frameworks, unit tests are run inside the browser.

Price: Free

Web site: www.dojotoolkit.org/

qUnit

qUnit is the unit-testing framework designed for use with jQuery. Indeed, jQuery itself uses qUnit for all of its testing. Despite this, qUnit has no dependency on jQuery and can be used to test any JavaScript code. qUnit prides itself on being a very simple unit-testing framework that lets people get up and running easily.

Price: Free

Web site: <https://github.com/jquery/qunit>

DOCUMENTATION GENERATORS

Most IDEs include documentation generators for the primary language. Since JavaScript has no official IDE, documentation has traditionally been done by hand or through repurposing documentation generators for other languages. However, there are now documentation generators specifically targeted at JavaScript.

JsDoc Toolkit

The *JsDoc Toolkit* was one of the first JavaScript documentation generators. It requires you to enter Javadoc-like comments into the source code, which are then processed and output as HTML files. You can customize the format of the HTML using one of the prebuilt JsDoc templates or you can create your own. The JsDoc Toolkit is available as a Java package.

Price: Free

Web site: <http://code.google.com/p/jsdoc-toolkit/>

YUI Doc

YUI Doc is YUI's documentation generator. The generator is written in Python, so it requires a Python runtime environment to be installed. YUI Doc outputs HTML files with integrated property and method searches implemented using the YUI's Autocomplete widget. As with JsDoc, YUI Doc requires Javadoc-like comments to be inserted into the source code. The default HTML can be changed through the modification of the default HTML template file and associated style sheet.

Price: Free

Web site: www.yuilexample.com/projects/yuidoc/

AjaxDoc

The goal of *AjaxDoc* is slightly different from that of the previous generators. Instead of creating HTML files for JavaScript documentation, it creates XML files in the same format as those created for .NET languages, such as C# and Visual Basic .NET. Doing so allows standard .NET documentation generators to create documentation as HTML files. *AjaxDoc* requires a format of documentation comments that is similar to the documentation comments for all .NET languages. *AjaxDoc* was created for use with ASP.NET Ajax solutions, but it can be used in standalone projects as well.

Price: Free

Web site: www.codeplex.com/ajaxdoc/

SECURE EXECUTION ENVIRONMENTS

As mashups became more popular, there was a greater need to allow JavaScript from outside parties to exist and function on the same page. This opens up several security issues regarding access to restricted functionality. The following tools aim to create secure execution environments in which JavaScript from a number of different sources can exist without interfering with one another.

ADsafe

Created by Douglas Crockford, *ADsafe* is a subset of JavaScript deemed safe for third-party scripts to access. For code to run within *ADsafe*, the page must include the *ADsafe* JavaScript library and be marked up in the *ADsafe* widget format. As a result, the code is assured to be safe for execution on any page.

Price: Free

Web site: www.adsafe.org/

Caja

Caja takes a unique approach to secure JavaScript execution. Similar to *ADsafe*, *Caja* defines a subset of JavaScript that can be used in a secure manner. *Caja* can then sanitize this JavaScript code and verify that it is doing only what it's supposed to. As part of the project, a language called *Cajita* is available, which is an even smaller subset of JavaScript functionality. *Caja* is still in its infancy but shows a lot of promise for allowing multiple scripts to run on the same page without the possibility of malicious activity.

Price: Free

Web site: <http://code.google.com/p/google-caja/>

INDEX

Symbols

\$\$, 157
\$&, 134, 157
\$', 134, 157
\$', 134, 157
+
 add operator, 61–62
 unary plus operator, 48–49
-
 subtract operator, 62–63
 unary minus operator, 48–49
/*, */ (block comment), 26
+= (add/assign), 68
& (bitwise AND operator), 52
| (bitwise OR operator), 52–53
^ (bitwise XOR operator), 53–54
-- (decrement operator), 46–48
/ (divide operator), 60
/= (divide/assign), 68
“ (double quotes)
 JSON strings, 692
 strings, 41–42
== (equal operator), 65–66
> (greater-than operator), 63–65
>= (greater-than-or-equal-to), 63–65
===(identically equal operator), 66–67
++ (increment operator), 46–48
<< (left shift), 54
<<= (left shift/assign), 68
< (less-than operator), 63–65
&& (logical AND operator), 57–58
! (logical NOT operator), 56–57
|| (logical OR operator), 58–59
%= (modulus assign), 68
% (modulus/remainder operator), 60–61
* (multiply operator), 59–60
*= (multiply/assign), 68
!= (not identically equal operator), 66–67

!= (not-equal operator), 65–66
>> (signed right shift), 54–55
>>= (signed right shift/assign), 68
// (single line comment), 26
‘ (single quotes), strings, 41–42
-= (subtract/assign), 68
>>> (unsigned right shift), 55–56
>>>= (unsigned right shift/assign), 68
\ “, 42
\ ‘, 42
\ (backslash), 42
\b (backspace), 42
\f (form feed), 42
\n (new line), 42
\r (carriage return), 42
\t (tab), 42
\unnnn, 42
\xnnn, 42
~ (bitwise NOT operator), 51

A

AAC, 604
abort, 452, 601
abs(), 169
accessibility, 364, 459, 471, 779, 784
accessor properties, 176–178
acos(), 169
activation object, 91
adaptable, maintainable code, 802
add operator (+), 61–62
add/assign (+=), 68
addColorStop(), 565
addElement(), 598
addEventlistener(), 438, 439, 440, 443
addHandler(), 441, 442, 740, 756, 757
additive operators, 61–63
addNamespace(), 686
addParameter(), 663, 664, 665

addTen(), 88
Adobe Flash, 3, 262, 263, 598, 602, 701
ADsafe, 895
advanced functions, 731–743
advanced JavaScript techniques, 731–764
advanced timers, 746–754
AES cryptography, 889
Ajax, 701–729
 ASP.NET Ajax solutions, 895
 Asynchronous JavaScript + XML, 701
 Comet, 721–723
 security, 728–729
 SSE, 723–725
 communication errors, 627–628
 CORS, 714–719
 in browsers, 714–717
 credentialed requests, 718
 cross-browser, 718–719
 GET requests, 703,
 707–708, 709, 714,
 715, 717, 718
 in Internet Explorer,
 714–716
 POST requests, 708–709,
 714, 715, 716, 717
 preflighted requests,
 717–718
 textures, 587
 cross-domain Ajax
 communication techniques, 719–728
 form serialization, 540–542
 image pings, 719–721
 Professional Ajax, 2nd Edition (Wiley), 729
 remote scripting, 701
 security, 728–729
 XHR, 701–712
 createXHR(), 702, 703,
 736, 737, 738

Ajax (*continued*)

file uploads with, 849–850, 856
 FormData type, 710–711
 GET requests, 707–708
 HTTP headers, 706–707
 Level 1, 710
 Level 2, 710
`overrideMimeType()`, 711–712
 POST requests, 708–709
 progress events, 712–714
`send()`, 704, 705, 708, 710, 712, 715, 850
 timeouts, 711
 usage, 703–706

“Ajax: A New Approach to Web Applications” (Garrett), 701

AjaxDoc, 895
`alert()`, 32, 110, 111, 164, 187, 253, 881
 alerts, debugging *v.*, 630, 633
 almost standards mode, 21–22. *See also* standards mode
`alpha`, 576
`anchor()`, 161
 AND
 bitwise AND operator (`&`), 52
 logical AND operator (`&&`), 57–58
 Android
 devices, 267, 287, 492
 operating system, 299, 712
 user-agent detection, 285–286
 animation loops, 836
 animation/visual effects
 libraries, 888
 `mozRequestAnimationFrame`
 `Frame`, 835, 837–839
 `requestAnimationFrame()`, 835–839, 856
 anonymous functions (lambda functions), 218
 block-level scoping, 228–230
 closures *v.*, 221
`anotherFactorial()`, 220
`anotherSum()`, 137
 antialias, 576
 APIs, 835–856
 Constraint Validation, 530–531
 Element Traversal, 360–361, 379
 properties, 360–361
 white space, 360
 File API, 843–850, 856
 Geolocation API, 841–843, 856

Indexed Database API, 786–799
 Page Visibility API, 839–841, 856
`requestAnimationFrame()`, 835–839, 856
 Selectors API, 357–360, 379
 `matchesSelector()`, 359–360
 `querySelector()`, 358, 359
 `querySelectorAll()`, 358–359, 379
 Web Timing, 851–852, 856
 Web Workers, 852–855, 856
`appendChild()`, 314, 315, 318, 335, 344, 387, 420, 481, 482, 537, 636, 825
`appendData()`, 338
 application cache, 766–768
 application logic/event handlers, decouple, 807–809
`apply()`, 144, 145, 146, 166, 167, 183, 207, 208, 739, 742, 882
`arc()`, 556, 557
`arcTo()`, 556
 argument passing, 88–89
 arguments, 80–83
 default values, 860
 rest, 859–860
 spread, 859–860
 arguments object, 80, 81, 82, 84, 88, 91, 141, 144, 145, 741, 860, 879
 arrays, 106–122
 conversion methods, 110–112
 creating, 107–108
 defined, 106–107
 detecting, 110
 ECMAScript Harmony, 861–865
 iterative methods, 119–121
 iterators, 862–863
 JSON, 693–694
 location methods, 118–119
 manipulation methods, 116–118
 null comparison, 812
 queue methods, 113–114
 raw image data, 568
 reduction methods, 121–122
 reordering methods, 114–116
 safe type detection, 110, 621, 731–733
 sizes, 109
 stack methods, 112–113
 typed, 571–576

array buffer views, 571–573
 array chunking, 751
 array comprehensions, 863–864
 array literals
 notation, 107–108
 usage, 823
 Array type, 106–122, 170
 array types, 870–871
 ArrayBuffer, 571
 Ashkenas, Jeremy, 887
`asin()`, 170
 ASP.NET Ajax solutions, 895
`assert()`, 635
 assignment operators, 67–68
`async` attribute, 13, 17–18, 23
 Asynchronous JavaScript + XML, 701. *See also* Ajax
 asynchronous scripts, 17–18
`atan()`, 170
`atan2()`, 170
`attachEvent()`, 439–440, 447
`Attr` type, 345–346
 attribute nodes, 345–346
 attributes. *See also specific attributes*
 accessing, E4X, 678–679
 getting, 330–332
 setting, 332–333
 uniforms and, 581
 attributes property, 333–335
`<audio>` element, 598–605
 Audio type, 604–605
 codec support detection, 603–604
 custom media players, 602–603
 events, 601–602
 properties, 599–601
 Audio type, 604–605
 automatic tab forward, 528–529
`autoplay`, 599

B

Backbone.js, 887
`backcolor`, 544
 back-forward cache, 487
`backslash (\|)`, 42
`backspace (\b)`, 42
`BaseComponent`, 235, 236, 237
`beforecopy`, 526
`beforecut`, 526
`beforepaste`, 526
`beforeunload`, 483–484
`beginPath()`, 556
 best practices
 deployment, 827–833, 834
 build process, 827–829

- compression, 830–833
validation, 829–830
- maintainability, 801–814, 833–834
 maintainable code, 802–809, 833–834
 programming practices, 809–814
- performance, 814–827, 834
 bitwise operators, 821
 double interpretation, 820–821
 loop optimization, 817–819
 minimize statement count, 821–823
 native methods, 821
 optimize DOM
 interactions, 824–827
 property lookup, 816–817
 rolling loops, 819–820
 scope-awareness, 814–816
 switch statements, 821
- `bezierCurveTo()`, 556
- `big()`, 161
- `bind()`, 146, 739, 740, 742, 743
- bitwise AND operator (`&`), 52
- bitwise NOT operator (`~`), 51
- bitwise operators, 49–59, 821
- bitwise OR operator (`|`), 52–53
- bitwise XOR operator (`^`), 53–54
- `Blob`, 846
- blob URLs, 847–848
- block comment (`/* */`), 26
- blocking characters, 525–526
- block-level scopes, 93–96, 228–230, 858–859
- `blur()`, 518
- `blur` event, 458, 519, 520
- `<body>, <script>` elements in, 16
- `bold()`, 161
- `bold`, 544
- BOM (Browser Object Model), 239–269
 defined, 9–10
 elements, 268–269
 history object, 267–268
 location object, 255–259
 navigator object, 259–265
 screen object, 265–267
 window object, 165–166, 239–255
- `Boolean()`, 34, 56, 69
- Boolean operators, 56–59
- Boolean type, 34–35, 146, 148–149
- bracket notation, 106
- brackets, square, 107, 108, 129, 174
- break statement, 73–75
- browser detection, 274–275.
See also client detection
- Browser Object Model. *See* BOM
- browsers. *See also* client detection; *specific browsers*
- CORS in, 714–717
 - cross-browser event handlers, 441–442
 - cross-browser event object, 449–451
 - cross-browser XML processing, 649–650
 - cross-browser XPath, 657–660
 - cross-browser XSLT, 667–668
 - DOM support, 8–9
 - ECMAScript support, 5–6
 - error reporting, 607–614
 - history, user-agent detection, 277–286
 - identifying, 291–294
 - XML DOM support, 641–650
 - XPath support, 651–660
 - XSLT support, 660–668
- bubbles, 443
- buffered, 599
- bufferedBytes, 599
- bufferingRate, 599
- bufferingThrottled, 599
- buffers, WebGL, 579–580
- build process, 827–829
- `buildUrl()`, 93
- built-in objects, 161–171, 215
- `<button>` element, 512
- buttons, mouse events, 466–467
- C**
- Caja, 895
- Cajita, 895
- `call()`, 145–146, 166, 183, 207, 208, 752, 754, 882
- `callee`, 141, 220, 880
- `caller` property, 142–143
- `callSomeFunction()`, 139, 140
- camel case, 26
- cancelable, 443
- `cancelBubble`, 447
- `canplay`, 601
- `canplaythrough`, 601
- `canPlayType()`, 603
- `canshowcurrentframe`, 601
- `<canvas>` element, 551–589
- basic usage, 551–553
 - 2D drawing context, 553–570
 - compositing, 569–570
- defined, 588–589
- fills, 553
- gradients, 565–567
- images, 563–564
- paths, 556–557
- patterns, 567
- raw image data, 567–569
- rectangles, 553–556
- shadows, 564
- strokes, 553
- text, 557–559
- transformations, 559–562
- WebGL, 571–588
- buffers, 579–580
- constants, 577
- context, 576–588
- coordinates, 578–579
- defined, 589
- drawing, 584–586
- errors, 580
- GLSL, 580–584, 589
- JavaScript *v.*, 580
- method names, 578
- OpenGL, 571, 576, 577, 578, 579, 580, 581, 582, 587, 589
- pixels, 587–588
- shaders, 580–584
- support, 588
- textures, 587
- 3D drawing context, 551, 571, 589
- triangles, 584–586
- typed arrays, 571–576
- typed views, 573–576
- viewports, 578–579
- `CanvasGradient`, 565
- capability detection, 271–275, 306
- browser detection *v.*, 274–275
 - safe, 273–274
- carriage return (`\r`), 42
- Cascading Style Sheets. *See* CSS
- case-sensitivity, 25–26
- `CDATASection` type, 342–343
- change event, 519, 520
- `changeColor()`, 91, 92
- character codes, keyboard events, 474
- character literals, 42–43
- character methods, `String` type, 152
- character set properties, HTML5, 366
- characters, blocking, 525–526
- `charAt()`, 152
- `charCodeAt()`, 152

charset property, 13, 366, 398, 831
checkValidity(), 532–533
childElementCount, 360
children, element, 336–337
children property, 374
Chrome
 DOM support, 9
 ECMAScript support, 6
 error reporting, 613–614
 user-agent detection, 283–284
chunk(), 751–752
circular references, 97–98, 101, 227, 228
class attribute, 327, 330, 361, 362, 363, 364, 383
class keyword, 327, 871
classes. *See also* object-oriented programming; reference types
class-related additions, HTML5, 361–364
ECMAScript and, 173, 180, 215
ECMAScript Harmony, 871–874
getElementsByClassName(), 361–362, 810
reference types *v.*, 103, 170
classical inheritance, 207–208
classList property, 362–364
className property, 327, 328, 330, 362, 364
clearColor(), 578
clearData(), 527, 598
clearInterval(), 252, 854
clearRect(), 553, 555
clearTimeout(), 252, 854
clearWatch(), 843
click event, 459
client coordinates, 461
client detection
 capability detection, 271–275, 306
 quirks detection, 275–276, 307
 user-agent detection, 276–306, 307
client dimensions, elements, 403–404
client-side data storage. *See also* offline web applications
cookies, 768–778
 HTTP-only, 778
 in JavaScript, 770–773
 parts, 769–770
 restrictions, 769
 security considerations, 778
 subcookies, 773–778

IndexedDB, 786–799
 concurrency issues, 798–799
 databases, 787–788
 indexes, 796–798
 key ranges, 794–795
 limitations/restrictions, 799
 object stores, 788–789
 querying with cursors, 791–793
 setting cursor direction, 795–796
 transactions, 790–791
Internet Explorer persistent user data, 778–779
Web Storage, 780–786
 globalStorage object, 783–784
 limitations/restrictions, 786
 localStorage object, 784–785
 sessionStorage object, 781–783
 storage event, 785–786
 Storage type, 780–781
clip(), 556
clipboard events, 526–528, 550
cloneNode(), 315, 316, 387, 390
cloning
 DOM ranges, 424
 Internet Explorer ranges, 428
close(), 248, 324, 326, 724, 855, 861
closePath(), 556
closures
 anonymous functions *v.*, 221
 garbage collection and, 227–228
 overview, 237–238
 privileged methods, 231–233, 234, 235, 238
 scope chains and, 221–224
 this object, 225–227
 variables and, 224–225
code
 conventions, 802–805, 833–834
 naming functions/variables, 803–804
 readability, 803
 loosely coupled, 805–809
 maintainable, 802–809
 readability, 803
 validation, 829–830
code injection, 164
code size, 830
coercion, of this value, 882
collapsing
 DOM ranges, 422–423
 Internet Explorer ranges, 427
COM (Component Object Model) objects, 97, 98, 227, 228, 263, 273, 309, 312, 637, 638, 732
combination constructor/prototype pattern, 197–198
combination inheritance, 209–210, 215
Comet, 721–723. *See also* Ajax security, 728–729
 SSE, 723–725
comma operator, 68
comment nodes, 341–342
Comment type, 341–342
comments
 block comment /* */ , 26
 defined, 26
 Javadoc-like, 894
 readability, 803
 single line comment // , 26
 type comments, 804–805
compareDocumentPosition(), 375, 376
comparing
 DOM ranges, 423–424
 Internet Explorer ranges, 427–428
comparison functions, 115–116
compatibility mode, 114, 280, 365
compatMode, 246, 365
complex selection
 DOM ranges, 417–419
 Internet Explorer ranges, 425–426
Component Object Model. *See* COM objects
compositing, 569–570
composition events, 478–479
compositionend, 478
compositionstart, 478
compositionupdate, 478
compound assignment operators, 68
compression, 830–833
 file, 830–832
 HTTP, 832–833
 YUI Compressor, 831–832, 893
computed styles, 394–396
concat(), 116, 152–153, 742, 752
concurrency issues, IndexedDB, 798–799
conditional operator, 67

[[Configurable]] attribute, 174, 175, 177, 178, 240, 744
confirm(), 253, 254
 conformance
 DOM conformance detection, 323–324
 ECMAScript, 4–5
 console, log debugging messages to, 631–633
 constants
 best practice usage, 813–814
 ECMAScript Harmony, 858
 WebGL context, 577
 Constraint Validation API, HTML5, 530–534
 construction/manipulation, XML, 682–685
 constructor pattern, 181–184
 durable, 200–201
 hybrid constructor/prototype pattern, 197–198
 parasitic, 199–200
 constructor stealing, 207–208
 constructors
 defined, 45, 103, 733
 as functions, 182–183, 733
 problems with, 183–184
 scope-safe, 733–735
contains(), 374–376
contentDocument, 390
contenteditable, 543, 545, 549, 550
contentWindow, 390
context. *See* scopes
contextmenu, 482–483
continue statement, 73–75
controls, 599
conversions
 arrays, 110–112
 to Boolean values, 34
 64-bit to 32-bit, 49–51
 string case, 156
 to strings, 43–44
convertToArray(), 313
cookies, 768–778
 HTTP-only, 778
 in JavaScript, 770–773
 parts, 769–770
 restrictions, 769
 security considerations, 778
 subcookies, 773–778
coordinates, WebGL, 578–579
copy, 526, 544
copy(), 683
CORS (Cross-Origin Resource Sharing), 714–719
 in browsers, 714–717
 credentialed requests, 718
 cross-browser, 718–719
 GET requests, 703, 707–708, 709, 714, 715, 717, 718
 in Internet Explorer, 714–716
 POST requests, 708–709, 714, 715, 716, 717
 preflighted requests, 717–718
 textures, 587
cos(), 170
 coupled code, loosely, 805–809
createAttribute(), 345, 346
createAttributeNS(), 385
createCDataSection(), 343
createComparisonFunction(), 141, 219, 221, 222, 223
createDocument(), 387–388, 641–642
createDocumentType(), 387–388
createElement(), 273, 335, 336, 825
createElementNS(), 385
createExpression(), 651
createFunctions(), 224, 225
createHTMLDocument(), 388
createLinearGradient(), 565, 566
createlink, 544
createNSResolver(), 651
createPerson(), 99, 180, 181
createRadialGradient(), 566
createRectLinearGradient(), 566
createStreamingClient(), 723
createTextNode(), 339
createTextRange(), 424, 523, 524
createXHR(), 702, 703, 736, 737, 738
createXSLTTemplate(), 662–663
 credentialed requests, 718
 Crockford, Douglas, 200, 210, 211, 234, 691, 732, 829, 891, 892, 895
 cross-browser CORS, 718–719
 cross-browser event handlers, 441–442
 cross-browser event object, 449–451
 cross-browser XML processing, 649–650
 cross-browser XPath, 657–660
 cross-browser XSLT, 667–668
 cross-document messaging (XDM)
 `JSON.stringify()`, 592
 `postMessage()`, 591–592, 852–853
 security, 591, 606
 Web Messaging, 593
 cross-domain Ajax communication techniques, 719–728
 Cross-Origin Resource Sharing. *See* CORS
 cross-site request forgery (CSRF), 715, 728
 cross-site scripting (XSS) attacks, 715
 crunchers, 892–893
 cryptography, libraries, 888–889
 CSRF (cross-site request forgery), 715, 728
CSS (Cascading Style Sheets), 321.
 See also style sheets
 DOM Level 1 support, 7
 JavaScript/CSS coupling, 806–807
 rules, 398–401
 creating, 399–400
 deleting, 400–401
cssStyleRule object, 398
cssStyleSheet type, 396, 397
cssText property, 350, 392, 393, 398
 curly braces
 expression context, 104
 statement context, 27, 104
currentLoop, 599
currentSrc, 599
currentTarget, 443
currentTime, 599
curry(), 741–743
 cursors
 cursor trail, 758
 querying with, 791–793
 setting cursor direction, 795–796
 custom data attributes, HTML5, 366–367
 custom events, 755–758
cut, 526, 544

D

data format, JSON as, 691
 data properties, 174–176
 data storage. *See also* offline web
 applications
 cookies, 768–778
 HTTP-only, 778
 in JavaScript, 770–773

data storage (*continued*)
 parts, 769–770
 restrictions, 769
 security considerations, 778
 subcookies, 773–778
 IndexedDB, 786–799
 concurrency issues, 798–799
 databases, 787–788
 indexes, 796–798
 key ranges, 794–795
 limitations/restrictions, 799
 object stores, 788–789
 querying with cursors, 791–793
 setting cursor direction, 795–796
 transactions, 790–791
 Internet Explorer persistent user data, 778–779
 Web Storage, 780–786
 globalStorage object, 783–784
 limitations/restrictions, 786
 localStorage object, 784–785
 sessionStorage object, 781–783
 storage event, 785–786
 Storage type, 780–781

data types. *See* primitive types
 databases, IndexedDB, 787–788
 dataTransfer object, 595–598
 dataunavailable, 601
 Date type, 122–128, 170
 date-formatting methods, 125–126
 date-time component methods, 126–128
 inherited methods, 124–125
 date-formatting methods, 125–126
 Date.now(), 124
 Date.parse(), 122–123, 123
 date-time component methods, 126–128
 Date.UTC(), 122, 123
 dblclick event, 459
 debugging. *See also* error handling
 alerts *v.*, 630, 633
 maintainable code, 802
 script debugger, 608
 shaders, 584
 techniques, 630–635
 validators, 891–892

decodeURI(), 163
 decodeURIComponent(), 163
 decoupling
 CSS/JavaScript, 806–807
 HTML/JavaScript, 805–806
 decrement operator (–), 46–48
 dedicated workers, 855
 default argument values, 860
 default prototypes, 203–204
 defaultCharset property, 366
 defaultPlaybackRate, 600
 defaultView property, 387
 defer attribute, 13, 16–17, 23
 deferred scripts, <script> elements, 16–17
 defineProperty, 866
 defining multiple properties, 178–179
 delete, 544
 deleteData(), 338
 deleteRule(), 397, 400
 deletion, splice(), 117
 deployment best practices, 827–833, 834
 build process, 827–829
 compression, 830–833
 validation, 829–830
 depth, 576
 dereferencing variables, 99, 100, 101
 derived traps, 866–867
 descendants(), 681
 destructuring assignments, 864–865
 detach(), 424
 detachEvent(), 439–440
 detail, 443
 detection
 arrays, 110
 offline web applications, 765–766
 plug-ins, 262–264
 device events, 490–494
 devicemotion, 494
 deviceorientation, 492–494
 MozOrientation, 491–492
 orientationchange, 490–491
 devicemotion, 494
 deviceorientation, 492–494
 Dhakar, Lokesh, 888
 DHTML (Dynamic HTML), 7, 8, 309
 diff(), 80
 displayInfo(), 106
 divide operator (/), 60
 divide/assign (/=), 68

doAdd(), 81, 82
 document fragments, 344–345
 document modes, 21–22, 373–374
 almost standards mode, 21–22
 quirks mode, 21–22, 245, 365, 373, 374, 392, 404, 405, 457, 462
 standards mode, 21–22, 245, 246, 365, 373, 374, 392, 404, 405, 457, 462, 807
 document object
 document writing, 324–326
 HTMLODocument instance, 316, 317, 318
 special collections, 322–323
 Document Object Model. *See* DOM
 Document type, 316–326
 constructor, 317
 DOM Level 2 changes, 385
 prototype, 317
 document writing, 324–326
 document.anchors, 322
 document.applets, 322
 documentation generators, 894–895
 document.doctype, 317, 318, 343
 document.domain, 319–320
 documentElement property, 317
 document.execCommand(), 543, 546, 547, 548, 550
 document.forms, 322
 DocumentFragment type, 344–345
 document.head property, 365
 document.hidden, 839, 840, 841
 document.images, 323
 document.links, 323
 DocumentType type, 343–344
 characteristics, 343
 DOM Level 2 changes, 386–387
 document.VisibilityState, 840, 841
 DOH. *See* Dojo Object Harness
 Dojo Object Harness (DOH), 894
 Dojo ShrinkSafe, 892–893
 Dojo Toolkit, 886, 892
 DOM (Document Object Model), 309–355. *See also* events; nodes
 Attr type, 345–346
 CDATASection type, 342–343
 changes, 382–390
 Comment type, 341–342
 conformance detection, 323–324
 defined, 6, 309

Document type, 316–326
 DocumentFragment type, 344–345
 DocumentType type, 343–344
 dynamic scripts, 346–348, 400, 628
 dynamic styles, 348–350, 400, 628
 Element type, 326–337
 event flow, 433–434
 event object, 442–446
 event simulation, 502–508
 focus management, 364
 interactions, optimizing, 824–827
 <link> element, 348–350
 live update, 824–825
 locating elements, 320–322
 manipulations, performance issues, 355
 Node type, 310–316
 NodeList objects, 353–354
 other DOMs, 8
 overview, 6–9
 ranges, 415–424
 clean up, 424
 cloning, 424
 collapsing, 422–423
 comparing, 423–424
 complex selection, 417–419
 inserting content, 421–422
 interacting with content, 419–421
 simple selection, 416–417
 reason for, 7
 <script> elements, 346–348
 <style> element, 348–350
 styles, properties/methods, 392–394
 support in browsers, 8–9
 <table> element, 350–353
 Text type, 337–341
 working with, 346–354

DOM events. *See events*

DOM extensions, 357–380
 Element Traversal, 360–361, 379
 properties, 360–361
 white space, 360
 growth of, 380
 HTML5, 361–372, 380
 proprietary, 372–379, 380
 Selectors API, 357–360, 379
 matchesSelector(), 359–360

querySelector(), 358, 359
 querySelectorAll(), 358–359, 379

DOM Level 0
 description, 8
 event handlers, 437

DOM Level 1, 7–8, 309, 381

DOM Level 2
 Core module, 381–387, 428
 Document type, 385
 DocumentType type, 386–387
 Element type, 385–386
 event handlers, 438–439
 HTML module, 381, 388
 NamedNodeMap, 386
 specifications, 428
 Styles module, 381, 390–401, 429
 Traversal and Range module, 381, 408–428, 429
 Views module, 381, 387–388
 XML DOM support in browsers, 641–642

DOM Level 3
 compareDocumentPosition(), 375, 376
 Core, 382
 description, 8
 events, 451, 452, 458, 459, 471, 474, 475–476, 478, 502
 isDefaultNamespace(), 384
 isEqualNode(), 389
 isSameNode(), 389
 keyboard events, 475–476
 lookupNamespaceURI(), 384
 lookupPrefix(), 384
 setUserData(), 389
 textContent property, 377, 378
 XPath, 651–656

DOMActivate, 452
 DOMAttrModified, 479
 DOMCharacterDataModified, 479
 DOMContentLoaded, 484–485
 DOMFocusIn event, 458
 DOMFocusOut event, 458
 DOMNodeInserted, 479
 DOMNodeInsertedIntoDocument, 479
 DOMNodeRemoved, 479
 DOMNodeRemovedFromDocument, 479
 DOMParser type, 642–644
 DOMSubtreeModified, 479
 dot notation, 106

double interpretation, 820–821
 double quotes (“)
 JSON strings, 692
 strings, 41–42

double-escaped metacharacters, 130

do-while statement, 70

drag, 593

drag-and-drop functionality
 dataTransfer object, 595–598
 drop targets, 594
 dropEffect property, 596–597
 effectAllowed property, 596–597
 events, 593–594
 file reading, 848–849
 file uploads, 849–850, 856
 mouse events, 758–764

dragDrop(), 597

DragDrop object, 759, 760, 762, 763

dragend, 593

dragenter, 594

draggable, 597

draggable attribute, 597

dragleave, 594

dragover, 594

dragstart, 593

drawArrays(), 584, 585

drawElements(), 579, 584, 585

drawImage(), 563

drawing. *See also* <canvas>
 element
 gradients, 565–567
 images, 563–564
 paths, 556–557
 patterns, 567
 rectangles, 553–556
 shadows, 564
 text, 557–559
 WebGL, 584–586

drawing transformations, 559–562

dropEffect property, 596–597

Duff's device, 819–820

durable constructor pattern, 200–201

durable objects, 200–201

duration, 600

durationchange, 601

Dynamic HTML (DHTML), 7, 8, 309

dynamic prototype pattern, 198–199

dynamic scripts, DOM, 346–348, 400, 628. *See also* <script> elements

dynamic styles, DOM, 348–350, 400, 628

E

E4X. *See* ECMAScript for XML
 ECMA (European Computer Manufacturers Association), 2
 ECMAScript. *See also* strict mode
 classes and, 173, 180, 215
 conformance, 4–5
 definition, 3
 editions, 3–4
 elements, 83–84
 JavaScript *v.*, 3, 83
 language basics, 25–84
 OO languages *v.*, 103
 strict mode, 877
 support in browsers, 5–6
 syntax, 25–28
 tamper-proof objects, 743–746, 764, 809
 try-catch statement, 607
 ECMAScript for XML (E4X), 671–691
 accessing attributes, 678–679
 characteristics, 689–690
 enabling, 689
 general usage, 676–679
 namespaces, 674–675, 686–688
 node types, 679–691
 parsing options, 685–686
 purpose, 689
 QName type, 675–676
 querying, 681–682, 690
 XML type, 672–673
 xmllist type, 673–674
 ECMAScript Harmony, 857–875
 array comprehensions, 863–864
 array types, 870–871
 arrays, 861–865
 block-level scope, 858–859
 classes, 871–874
 destructuring assignments, 864–865
 functions, 859–861
 general changes, 857–859
 generators, 861
 history, 857
 inheritance, 873–874
 iterators, 862–863
 modules, 874–875
 proxy functions, 868
 proxy objects, 865–868
 set type, 869
 simple maps, 868–869
 struct types, 869–870

WeakMap type, 869
 effectAllowed property, 596–597
 Eich, Brendan, 2
 Element Traversal, 360–361, 379
 properties, 360–361
 white space, 360
 Element type, 326–337
 description, 326
 DOM Level 2 changes, 385–386
 elements. *See also* media elements; specific elements
 attributes
 getting, 330–332
 setting, 332–333
 children, 336–337
 client dimensions, 403–404
 creating, 335–336
 dimensions, 401–408
 HTML, 327–330
 locating, DOM, 320–322
 NoScope, 368–369
 offset dimensions, 401–403
 scoped, 368–369
 scroll dimensions, 404–406
 styles, accessing, 391–392
 emerging APIs. *See also* APIs
 File API, 843–850, 856
 Geolocation API, 841–843, 856
 Page Visibility API, 839–841, 856
 requestAnimationFrame
 Frame(), 835–839, 856
 Web Timing, 851–852, 856
 Web Workers, 852–855, 856
 emptied, 601
 empty, 601
 encodeURI(), 162–163
 encodeURIComponent(), 162–163, 627, 708, 771, 772
 ended, 600, 601
 e-notation, 36, 150
 entityReference
 Expansion, 410
 [[Enumerable]] attribute, 174, 175, 177, 178, 191, 194, 276, 744
 equal
 assignment operators, 67–68
 equal operator (==), 65–66
 greater-than-or-equal-to (>=), 63–65
 identically equal operator (===), 66–67
 not identically equal operator (!==), 66–67
 not-equal operator (!=), 65–66
 equal operator (==), 65–66
 equality operators, 65–67
 Error, 616
 error, 452
 error event, 622–623
 error handling, 607, 614–640.
 See also debugging strategies, 623–630
 try-catch statement, 615–619
 ECMAScript, 607
 error types, 616–618
 finally clause, 616
 throwing errors *v.*, 621–622
 usage, 618–619
 error reporting, browser, 607–614
 Error type, 617
 errors
 communication, 627–628
 data type, 625–627
 fatal, 628–629
 identification, 623–628
 IndexedDB databases, 787–788
 Internet Explorer, 635–639
 “invalid character,” 637
 “member not found,” 637–638
 “operation aborted,” 635–637
 syntax, 638–639
 “system cannot locate resource specified,” 639
 unknown runtime, 638
 logging, 629–630
 nonfatal, 628–629
 “object expected,” 607, 617
 throwing, 619–622, 634–635
 type coercion, 624–625
 WebGL, 580
 escaped metacharacters, 129–130
 European Computer Manufacturers Association. *See* ECMA
 eval(), 163–164, 695, 820, 880–881
 EvalError, 616
 evaluate(), 651, 652, 653, 654, 655, 656, 658
 events, 431–509. *See also* specific events
 <audio> element, 601–602
 clipboard, 526–528, 550

- complexity, 431
 composition, 478–479
 custom, 755–758
 defined, 431, 755
 device, 490–494
 - `devicemotion`, 494
 - `deviceorientation`, 492–494
 - `MozOrientation`, 491–492
 - `orientationchange`, 490–491
 DOM Level 3, 451, 452, 458, 459, 471, 474, 475–476, 478, 502
 drag-and-drop, 593–594
 error, 622–623
 focus, 458–459
 form field, 519–520
 gesture, 494–495, 497–498
 - `gesturechange`, 497
 - `gestureend`, 497
 - `gesturestart`, 497
 history, 431
 HTML5, 451, 482–490
 - `beforeunload`, 483–484
 - `contextmenu`, 482–483
 - `DOMContentLoaded`, 484–485
 - `hashchange`, 489–490
 - `pagehide`, 487–489
 - `pageshow`, 487–489
 - `readystatechange`, 485–487
 keyboard, 471–478
 - character codes, 474
 - on devices, 477–478
 - DOM Level 3 changes, 475–476
 - key codes, 472–474
 - simulating, 504–506
 load, 452–456
 memory considerations, 509
 mouse, 459–476
 - accessibility issues, 471
 - buttons, 466–467
 - client coordinates, 461
 - modifier keys, 463–464
 - page coordinates, 462
 - related elements, 464–466
 - screen coordinates, 462–463
 - simulating, 503–504
 - touch device support, 470–471
 mousewheel, 468–470
 mutation, 479–482
 - node insertion, 481–482
 - node removal, 480–481
 observer pattern, 431, 755, 764
 performance considerations, 509
 progress events, 712–714
 proprietary, 431, 451, 494, 495, 509
 simulation, 502–509
 SSE, 723–725, 727–728
 text, 472, 476–477
`textInput`, 471, 472, 476–477
 touch, 494–497
 - `touchcancel`, 495
 - `touchend`, 495
 - `touchmove`, 495
 - `touchstart`, 495
 UI, 452–457
`<video>` element, 601–602
 wheel, 451, 460
 event bubbling, 432
 event capturing, 433
 event delegation, 498–500
 - defined, 498
 - performance, 826
 event flow, 432–434
 “Event Handler Scope” (Smith), 436
 event handlers (event listeners), 434–442
 - application logic/event handlers coupling, 807–809
 - `cloneNode()`, 316
 - cross-browser, 441–442
 - defined, 434
 - DOM Level 0, 437
 - DOM Level 2, 438–439
 - HTML, 434–436
 - memory issues, 371
 - `onerror`, 622, 640, 715, 716, 720, 721, 791, 853
 - removing, 500–502
 event listeners. *See* event handlers
 event object, 442–451
 - cross-browser, 449–451
 - DOM, 442–446
 - Internet Explorer, 447–449
 - methods, 443–444
 - properties, 443–444
 eventPhase, 443
 EventUtil object, 449, 453, 455, 469, 513
 every(), 119, 120
 exec(), 132–133, 134, 136, 156
 execCommand(), 543, 546, 547, 548, 550
 execution contexts. *See* scopes
 exp(), 169
 experimental-webgl, 576
 expiration, cookies, 770
 extendable, maintainable code, 802
 Extensible HyperText Markup Language. *See* XHTML
 external files, `<script>` elements, 15–16, 20
 external modules, 875

F

- `factorial()`, 141, 142, 220
 factorial functions, 141, 220
 factory pattern, 180–181
 fake URL, 606
 fatal errors, 628–629
 FIFO (first-in first-out), 113
 File API, 843–850, 856
 file compression, 830–832
 file reading, drag-and-drop, 848–849
 file uploads, with XHR, 849–850, 856
 FileReader type, 844–846
`fill()`, 556
`fillRect()`, 553, 554, 556, 566
 fills, 553
`fillStyle`, 553, 562
`fillText()`, 557, 558, 559
 filter(), 119, 120
 Firebug, 609–610, 631
 Firefox
 - DOM support, 9
 - ECMAScript support, 6
 - enabling E4X, 689
 - error reporting, 609–610
 - Gecko rendering engine, 280–282
 - JavaScript version progression, 10`firstChild()`, 413, 414
`firstElementChild`, 360
 first-in first-out (FIFO), 113
`fixed()`, 161
 flags, regular expressions, 128–129
 Flash, 3, 262, 263, 598, 602, 701

Float32Array, 574
 Float64Array, 574
 floating-point values, 36–37
 flow-control statements. *See*
 statements
`focus()`, 517–518
`focus` event, 458
`focus` events, 458–459
`focus` management, 364
`focusin` event, 458
`focusout` event, 458
`fontcolor()`, 161
`fontname`, 544
`fontsize()`, 161
`fontsize`, 544
`for` statement, 71–72
`forEach()`, 119, 120
`for-each-in` loop, 688
`forecolor`, 544
`for-in` statement, 72–73
 forms (web forms), 511–550
 basics, 511–520
 resetting, 513–514
 rich text editing, 542–549, 550
 select boxes, 534–539, 550
 adding options, 537–538
 creating, 534
 moving options, 539
 removing options,
 538–539
 reordering options, 539
 selecting options, 536–537
 submitting, 512–513
 text boxes, 520–534
 input filtering, 524–528
 text selection,
 521–524, 549
`<form>` elements, 511–512
`form feed (f)`, 42
 form fields, 514–520
 automatic tab forward,
 528–529
 events, 519–520
 methods, 517–518
 properties, 516–517
 validation, 530–534
 form serialization, 540–542
`formatblock`, 544
`FormData` type, 710–711
 fragment shaders, 580, 581
 frames, window relationships and,
 241–244
`fromCharCode()`, 161, 474, 525
 frozen objects, 745–746
 Fuchs, Thomas, 888
 functions, 78–83, 136–146.
See also arguments

advanced, 731–743
 arguments, 80–83
 comparison, 115–116
 constructors as, 182–183, 733
 ECMAScript Harmony,
 859–861
 execution contexts,
 90–91, 100
 factorial, 141, 220
 global, 810–812, 814–815
 internals, 141–143
 lazy loading, 736–738
 loosely typed, 215
 methods, 143–146
 naming conventions,
 803–804
 as objects, 136, 143, 170
 overloading, 83, 138
 properties, 143–146
 safe type detection, 110, 621,
 731–733
 strict mode, 80, 879–880
`typeof` operator, 31
 as values, 139–141
 function binding, 738–741
 function currying, 741–743
 function declaration hoisting,
 138–139, 218
 function declarations, 138–139,
 217–218, 237
 function expressions, 217–238
 characteristics, 217–219
 function declarations *v.*,
 138–139, 217–218, 237
 named, 139, 220, 221
 function keyword, 78, 137, 217,
 218, 229, 871
 function names, 25, 92, 136, 137,
 138, 139, 142, 803
 function throttling, 752–754
 Function type, 136–146, 170
 fundamental traps, 866

G

game systems identification,
 301–302. *See also* user-agent
 detection
 garbage collection, 96–100
 closures and, 227–228
 mark-and-sweep, 96–97, 100
 memory management, 99–100
 performance issues, 98–99
 reference counting,
 97–98, 101
 Garrett, James, 701

Gecko rendering engine, 280–282
 general libraries, 885
 generators, 861
 Geolocation API, 841–843, 856
 gesture events, 494–495, 497–498
`gesturechange`, 497
`gestureend`, 497
`gesturestart`, 497
`[[Get]]`, 177, 744
 GET requests, 703, 707–708, 709,
 714, 715, 717, 718, 728, 851
`getAttribute()`, 330, 331, 332,
 334, 345, 346, 779, 800
`getAttributeNS()`, 385
`getBoundingClientRect()`, 406,
 407, 408
`getColor()`, 95, 96
`getComputedStyle()`, 394,
 395, 429
`getContext()`, 552, 576, 577
`getCurrentPosition()`,
 842, 843
`getDate()`, 127
`getDay()`, 127
`getElement()`, 272
`getElementById()`, 272, 320,
 321, 357, 512
`getElementsByClassName()`,
 361–362, 810
`getElementsByName()`, 322
`getElementsByTagName()`, 320,
 321, 322, 337, 342, 357, 362,
 365, 643, 827
`getElementsByTagNameNS()`, 385
`getEvent()`, 450
`getFullYear()`, 126
`getHours()`, 127
`getMonth()`, 126
`getNamedItem()`, 333
`getNamedItemNS()`, 386
`getOwnPropertyDescriptor`, 179,
 180, 189, 866
`getOwnPropertyNames`, 866
`getPropertyCSSValue()`, 392,
 393, 394
`getPropertyDescriptor`, 866
`getPropertyNames`, 866
`getPropertyValue()`, 392,
 393, 394
`getSeconds()`, 127
`getSelection()`, 547, 548
`getStyleSheet()`, 397, 398
`getSubValue()`, 203, 204,
 205, 206

`getSuperValue()`, 203, 204, 205, 206
`getTarget()`, 449, 450
`getters/setters`, 872–873
`getTime()`, 126
`getTimezoneOffset()`, 128
 getting attributes, 330–332
`getUserData()`, 389, 390
`getUTCDate()`, 127
`getUTCDay()`, 127
`getUTCFullYear()`, 126
`getUTCHours()`, 127
`getUTCMilliseconds()`, 128
`getUTCMinutes()`, 127
`getUTCMonth()`, 126
`getUTCSeconds()`, 127
`gl`, 576
`gl.bindBuffer()`, 579
`gl.bufferData()`, 579
`glClear()`, 578
`gl.compileShader()`, 582
`gl.createBuffer()`, 579
`gl.createShader()`, 582
`gl.createTexture()`, 587
`gl.drawArrays()`, 585–586
`gl.drawElements()`, 585–586
`gl.getAttribute`
 `Location()`, 583
`gl.getError()`, 580
`gl.getProgram`
 `Parameter()`, 584
`gl.getShader`
 `Parameter()`, 584
`gl.getUniform`
 `Location()`, 583
`global`, `RegExp` instance
 property, 131
 global execution context, 90, 100
 global lookups, 814–815
 Global object, 162–166, 171
 properties, 164–165
 URI-encoding methods,
 162–163
 window object, 165–166,
 239–255, 268
 global variables/functions,
 810–812, 814–815
`globalAlpha`, 569
`globalCompositionOperation`,
 569–570
`globalStorage` object, 783–784
`gl.shaderSource()`, 582
 GLSL. *See* OpenGL Shading
 Language
`GMT`, 122, 123, 126, 770
`go()`, 267, 268

graphics. *See* `<canvas>` element
`grayscale`, 569
 greater-than operator (`>`), 63–65
 greater-than-or-equal-to (`≥`), 63–65

H

`H.264`, 604
`handleKeyPress()`, 808
 handlers, registering, 264–265
 handling errors. *See* error handling
 Harmony. *See* ECMAScript
 Harmony
`hasAttribute()`, 385, 542
`hasAttributeNS()`, 385
`hasComplexContent()`, 680
`hasFeature()`, 323, 324, 388, 389,
 415, 466
`hasFlash()`, 264
 hash tables, 173. *See also* objects
`hashchange`, 489–490
`hasIEPlugin()`, 263
`hasOwnProperty()`, 45, 188, 190,
 191, 204, 658, 776, 866
`hasPlugin()`, 262
`hasPrototypeOf()`,
 190, 191
`hasQuickTime()`, 264
`hasSimpleContent()`, 680
 head property, 365
 height attribute, 552
 Heikkinen, Ilmari, 569
 hexadecimal format, 35, 39, 40, 41,
 42, 395
 hierarchy of nodes, 310
 history object, 267–268, 605–606
 history state management, 605–606
 host objects, 45, 273, 274
 HTML (HyperText Markup
 Language). *See also*
 elements; `<script>` elements;
 XHTML
 DHTML, 7, 8, 309
 DOM Level 2 HTML module,
 381, 388
 event handlers, 434–436
 HTML/JavaScript coupling,
 805–806
 methods, `String` type, 161
 XHTML *v.*, 18
 HTML elements, 327–330
 HTML5. *See also* APIs; `<canvas>`
 element
 character set properties, 366
 class-related additions,
 361–364

Constraint Validation API,
 530–534
 cross-document messaging
 (XDM)
`JSON.stringify()`, 592
`postMessage()`, 591–
 592, 852–853
 Web Messaging, 593
 custom data attributes,
 366–367
 DOM nodes, 361–372, 380
 drag-and-drop functionality
`dataTransfer` object,
 595–598
 drop targets, 594
`dropEffect` property,
 596–597
`effectAllowed` property,
 596–597
 events, 593–594
 file reading, 848–849
 file uploads, 849–850, 856
 events, 451, 482–490
`beforeunload`, 483–484
`contextmenu`, 482–483
`DOMContentLoaded`,
 484–485
`hashchange`, 489–490
`pagehide`, 487–489
`pageshow`, 487–489
`readystatechange`,
 485–487
 focus management, 364
 history state management,
 605–606
 markup insertion
`innerHTML` property,
 367–369
`innerText` property,
 376–378
`insertAdjacent
 HTML()`, 370–371
`outerHTML` property, 370,
 371, 638
`outerText` property,
 376, 378
 media elements
`<audio>`, 598–605
 codec support detection,
 603–604
 custom media players,
 602–603
`<video>`, 598–605
 scripting, 591–606
`scrollIntoView()`,
 372, 379

HTMLCollection objects, 321–323, 352, 826–827
HTMLDocument type
 changes, 364–365
 constructor, 317
 createHTMLDocument(), 388
 document object, 316,
 317, 318
 getElementsByName(), 322
 prototype, 317
htmlEscape(), 159
HTMLFormElement type,
 511–512. *See also* forms
HTMLFrameElement, 330, 390
HTMLIFrameElement, 330, 390
HTMLLinkElement, 329, 396
HTMLStyleElement, 329, 396
HTTP compression, 832–833
HTTP cookies. *See* cookies
HTTP headers, XHR, 706–707
HTTP streaming, 722–723,
 724, 729
HTTP-only cookies, 778
Hungarian notation, 804
hybrid constructor/prototype
 pattern, 197–198
HyperText Markup Language. *See*
 HTML

I

identically equal operator (==),
 66–67
identifier lookup, 95–96
identifiers, 26, 28, 29
IEEE 64 bit, 49, 574
IEEE-754-based numbers,
 35, 37, 49
if statement, 69
iframes
 document object, 390
 HTMLIFrameElement,
 330, 390
 rich text editing, 542–549, 550
 XDM, 593
ignoreCase, RegExp instance
 property, 131
image pings, 719–721
images
 draggable, 597
 drawing, 563–564
 raw image data, 567–569
 tag, 719, 848
importNode(), 387
importScripts(), 855
in operator, 189–192

increment operator (++), 46–48
indent, 544
Indexed Database API
 (IndexedDB), 786–799
 concurrency issues, 798–799
 databases, 787–788
 indexes, 796–798
 key ranges, 794–795
 limitations/restrictions, 799
 object stores, 788–789
 querying with cursors,
 791–793
 setting cursor direction,
 795–796
 transactions, 790–791
indexes, 796–798
indexOf(), 118, 119, 154, 155
infinity
 add operator, 61
 divide operator, 60
 multiply operator, 60
 range of values, 37
 subtract operator, 63
inheritance, 201–215
 classical, 207–208
 combination, 209–210, 215
 constructor stealing, 207–208
 ECMAScript Harmony,
 873–874
 parasitic, 211–212, 216
 parasitic combination,
 212–215, 216
 prototypal, 210–211
 prototype chaining, 202–207,
 215
 pseudoclassical, 209–210
inherited methods, Date type,
 124–125
inheritPrototype(), 214
inline JavaScript code, <script>
 elements, 14–15, 20
innerHTML property, 367–369, 825
innerText property, 376–378
<input> element, 512
input element types, 530–531
input filtering, 524–528
input patterns, 532
input property, 134–135
insertAdjacentHTML(), 370–371
insertBefore(), 314, 315, 344,
 481, 538, 539
insertChildAfter(), 683
insertChildBefore(), 684
insertData(), 338
insertHorizontalrule, 544
insertImage, 544
insertion, splice(), 117
insertNode(), 421
insertorderedlist, 544
insertparagraph, 544
insertRule(), 397, 399, 400
insertunorderedlist, 544
instanceof operator, 90, 100,
 110, 149, 151, 182, 193,
 200, 204, 236, 618, 658,
 689, 732, 813
instances, prototypes and, 204–205
Int8Array, 573
Int16Array, 573
Int32Array, 574
interacting with content
 DOM ranges, 419–421
 Internet Explorer ranges,
 426–427
interacting with rich text, 543–547
internals, function, 141–143
Internet application libraries,
 887–888
Internet Explorer
 CORS in, 714–716
 document modes, 373–374
 DOM support, 9
 ECMAScript support, 6
 error reporting, 608–609
 errors, 635–639
 “invalid character,” 637
 “member not found,”
 637–638
 “operation aborted,”
 635–637
 syntax, 638–639
 “system cannot locate
 resource specified,” 639
 unknown runtime, 638
event bubbling, 432
event handlers, 439–440
event object, 447–449
event simulation, 508–509
persistent user data, 778–779
ranges, 424–428
 cloning, 428
 collapsing, 427
 comparing, 427–428
 complex selection,
 425–426
 simple selection, 424–425
user-agent detection, 278–280
XML in, 644–649
XPath in, 656–657
XSLT in, 660–665
intervals, 251–253, 836–837
intuitive, maintainable code, 802

“invalid character,” 637
 iOS, user-agent detection, 285–286
 iPad, 285, 299
 iPhone, 285, 299, 305, 495
 Ippolito, Bob, 886
`isArray()`, 110, 732
`isDefaultNamespace()`, 384
`isEqualNode()`, 389
`isFinite()`, 37, 162
`isFunction()`, 732
`isHostMethod()`, 274
`isNaN()`, 38, 162
`isPrototypeOf()`, 45, 186, 204, 205, 210, 215
`isSameNode()`, 389
`isSupported()`, 388
`isXMLName()`, 689
 italic, 544
 italics(), 161
`item()`, 312, 321
 iterative methods, arrays, 119–121
 iterators, 862–863

J

Javadoc-like comments, 894
 JavaScript. *See also* BOM; DOM; ECMAScript; JSON; XML
 best practices
 deployment, 827–833, 834
 maintainability, 801–814, 833–834
 performance, 814–827, 834
 cookies in, 770–773
 CSS/JavaScript coupling, 806–807
 defined, 3, 11
 documentation generators, 894–895
 ECMAScript *v.*, 3, 83
 history, 1–2
 HTML/JavaScript coupling, 805–806
 overview, 1–11
 parts, 11
 secure execution
 environments, 895
 versions, 10–11
 WebGL *v.*, 580
 JavaScript libraries, 885–889
 animation/visual effects, 888
 Backbone.js, 887
 cryptography, 888–889
 Dojo Toolkit, 886
 general, 885
 Internet application, 887–888

jQuery, 357, 886, 887, 894
 Konqueror, 283
 MochiKit, 886–887
 moo.fx, 888
 MooTools, 886
 MSXML, 644, 660, 665, 669, 702
 Prototype, 810, 886
 qooxdoo, 887–888
 Rico library, 887
`script.aculo.us`, 888
 specialty, 885
 Underscore.js, 887
 Yahoo! User Interface Library, 811, 885, 893
 JavaScript Lint, 892
 JavaScript MD5 library, 889
 JavaScript Object Notation. *See* JSON
 “JavaScript split bugs: Fixed!” (Levithan), 159
 JavaScript tools, 891–895
 documentation generators, 894–895
 minifiers, 892–893
 secure execution
 environments, 895
 unit testing, 893–894
 validators, 891–892
 JavaScript library, 889
 Johnston, Paul, 889
`join()`, 111, 112, 542, 777
 JPEG encoding, 552
 jQuery, 357, 886, 887, 894
 .js extension, 15
 JsDoc Toolkit, 894
 JSHint, 892
 JSLint, 829–830, 891
 JSMin, 892
 JSON (JavaScript Object Notation), 691–700
 arrays, 693–694
 as data format, 691
 objects, 692–693, 695–696
 parsing, 694–700
 serialization, 694–700
 simple values, 692
 syntax, 691–694
 XML *v.*, 691, 700
 JSON object, 4, 695–696, 700, 732
 JSONP (JSON with padding), 721–722
`JSON.parse()`, 592, 695, 696, 699, 700
`JSON.stringify()`, 592, 695, 696, 697, 698, 699, 700

JsUnit, 893
`justifycenter`, 544
`justifyleft`, 544

K

key codes, 472–474
 key ranges, 794–795
 keyboard events, 471–478
 character codes, 474
 on devices, 477–478
 DOM Level 3 changes, 475–476
 key codes, 472–474
 simulating, 504–506
`keydown` event, 471–472
`keypress` event, 471–472
`keyup` event, 471–472
 keywords
 identifiers *v.*, 26, 28, 29
 list, 28–29
 King, Andrew B., 820
 Koch, Peter-Paul, 246
 Konqueror, 283

L

labeled statements, 73
 lambda functions. *See anonymous functions*
`language` attribute, 14
`lastChild()`, 413
`lastElementChild`, 360
 last-in first-out (LIFO), 112, 113
`lastIndexOf()`, 118, 119, 154, 155
`lastMatch` property, 134–135
`lastParen` property, 134–135
 lazy loading functions, 736–738
 left shift (`<<`), 54
 left shift/assign (`<<=`), 68
`leftContext` property, 134–135
`length` property, 143–144
 less-than operator (`<`), 63–65
 Levithan, Steven, 159
 libraries, JavaScript, 885–889
 animation/visual effects, 888
 Backbone.js, 887
 cryptography, 888–889
 Dojo Toolkit, 886
 general, 885
 Internet application, 887–888
 jQuery, 357, 886, 887, 894
 Konqueror, 283
 MochiKit, 886–887
 moo.fx, 888
 MooTools, 886

libraries, JavaScript (*continued*)

- MSXML, 644, 660, 665, 669, 702
- Prototype, 810, 886
- qooxdoo, 887–888
- Rico library, 887
- script.aculo.us, 888
- specialty, 885
- Underscore.js, 887
- Yahoo! User Interface Library, 811, 885, 893
- LIFO (last-in first-out), 112, 113
- Lightbox, 888
- lines, WebGL, 584, 585
- lineTo(), 556, 557
- link(), 161
- alink element, 348–350, 396
- links, draggable, 597
- listeners. *See* event handlers
- live update, DOM, 824–825
- LiveConnect, 632
- load event, 452–456, 601, 712–713
- loadeddata, 601
- loadedmetadata, 601
- loading XML files, Internet Explorer, 647–648
- loadstart, 601, 712
- loadStyles(), 349
- localeCompare(), 160
- localStorage object, 784–785
- locating elements, DOM, 320–322
- location methods
 - arrays, 118–119
 - strings, 154–155
- location object, 255–259
 - manipulating, 257–259
 - properties, 255–256
 - query string arguments, 256–257
- log(), 169
- logging debugging messages
 - to console, 631–633
 - to page, 633–634
- logging errors, 629–630
- logical AND operator (`&&`), 57–58
- logical NOT operator (`!`), 56–57
- logical OR operator (`||`), 58–59
- long polling, 721
- long-running script limit, 750
- lookupNamespaceURI(), 384
- lookupPrefix(), 384
- lookups
 - global, 814–815
 - properties, 816–817
- loop, 600
- loops

animation, 836

- break statement, 73–75
- continue statement, 73–75
- for-each-in, 688
- for-in, 72–73
- nested, 73, 74
- optimization, 817–819
- post-test, 70, 818, 819
- pretest, 70, 71, 819
- for statement, 71–72
- unrolling, 819–820
- while statement, 70

loosely coupled code, 805–809

- application logic/event handlers, 807–809
- CSS/JavaScript, 806–807
- HTML/JavaScript, 805–806

loosely typed

- functions, 215
- variables, 29, 30, 31, 85, 623, 625, 804, 833

M

maintainability, 801–814, 833–834

- external files *v.* inline JavaScript code, 20
- maintainable code, 802–809
 - characteristics, 802
 - code conventions, 802–805, 833–834
- programming practices, 809–814
 - constants, 813–814
 - global variables/functions, 810–812, 814–815
 - null comparisons, 812–813
 - object ownership, 809–810

“Making Image Filters with Canvas” (Heikkinen), 569

malformed URLs, 627

manifest file, 766–768

manipulating

- DOM, performance issues, 355
- location object, 257–259
- nodes, 314–315
- tables, 350–353

manipulation methods, arrays, 116–118

map(), 119, 120

Map type, 868–869

mark-and-sweep garbage collection, 96–97, 100

markup insertion

- `innerHTML` property, 367–369
- `innerText` property, 376–378
- `insertAdjacentHTML()`, 370–371
- `outerHTML` property, 370, 371, 638
- `outerText` property, 376, 378

mashups, 201, 367, 593, 895

match(), 156–157

matchesSelector(), 359–360

Math object, 166–170, 171

- `max()`, 167
- methods, 167–170
- `min()`, 167
- properties, 166–167
- rounding methods, 167–168

`Math.ceil()`, 167

Mathematical Markup Language (MathML), 8

`Math.floor()`, 167

MathML (Mathematical Markup Language), 8

`Math.random()`, 168–169

`Math.round()`, 168

`max()`, 167

MD4, 889

MD5, 889

`measureText()`, 559

media elements

- `<audio>`, 598–605
- codec support detection, 603–604
- custom media players, 602–603
- `<video>`, 598–605

media players, custom, 602–603

“member not found,” 637–638

memory management. *See also*

- performance
- event delegation, 498–500
- event handler removal, 500–502
- events, 509
- garbage collection, 99–100
- markup insertion, 371–372

metacharacters, 129–130

method names, WebGL, 578

methods. *See specific methods*

Michaux, Peter, 274

Microsoft.XmlDom, 644

Miller, Matthias, 892

MIME types, 14, 15, 20, 260, 261, 262, 263, 264, 265, 527, 552, 595, 603, 711, 712, 724, 833, 844, 846

mimicking block-level scoping, 228–230
`min()`, 167
 minifiers, 892–893
 minus
 decrement operator `(--)`, 46–48
 subtract operator `(-)`, 62–63
 unary minus operator `(-)`, 48–49
 mobile devices identification, 298–301. *See also user-agent detection*
 mobile viewports, 246
 MochiKit, 886–887
 modifier keys, 463–464
 module pattern, 234–236, 762, 763
 module-augmentation pattern, 236–237
 modules, ECMAScript Harmony, 874–875
 modulus operator `(%)`, 60–61
 modulus/assign `(%=)`, 68
moo.fx, 888
 MooTools, 886
 Mosaic, 20, 277, 278
 mouse events, 459–476. *See also specific mouse events*
 accessibility issues, 471
 buttons, 466–467
 client coordinates, 461
 drag-and-drop functionality, 758–764
 modifier keys, 463–464
 page coordinates, 462
 related elements, 464–466
 screen coordinates, 462–463
 simulating, 503–504
 touch device support, 470–471
 mousedown event, 459, 460, 466, 467, 470, 471, 496, 500, 597, 759, 760
 mouseenter event, 459, 460
 mouseleave event, 459, 460
 mousemove event, 459, 470, 471, 496, 593, 759, 760, 761
 mouseout event, 445, 459, 464, 465, 466, 471, 500, 503, 594
 mouseover event, 323, 434, 459, 464, 465, 471, 496, 500, 503, 594
 mouseup event, 459, 460, 466, 467, 470, 500, 760
 mousewheel event, 460, 468–470, 471
`moveEnd()`, 425, 426, 523
`moveStart()`, 425, 426, 523

`moveTo()`, 556, 557
 moving options, 539
 Mozilla Project, 5, 10, 284
 MozOrientation, 491–492
`mozRequestAnimationFrame`, 835, 837–839
 MP3, 604
`msRequestAnimationFrame()`, 838–839
`msvisibilitychange`, 840
 MSXML, 644, 660, 665, 669, 702
`MSXML.DOMDocument` versions, 644–645
 multiline property, 131, 134
 multiplicative operators, 59–61
 multiply operator `(*)`, 59–60
 multiply/assign `(*=)`, 68
 mutation events, 479–482
 node insertion, 481–482
 node removal, 480–481
 muted, 600

N

`$n`, 158
 named function expressions, 139, 220, 221
`NamedNodeMap`, 333, 343, 353, 386
`namespace()`, 675, 686
 Namespace objects, 674–675
`namespaceDeclarations()`, 687
 namespaces
 defined, 811
 E4X, 674–675, 686–688
 global variables/functions *v.*, 811–812
 modules *v.*, 874–875
 XML, 382–386, 655–656
 naming conventions, variables/functions, 803–804
 NaN (not a number), 37–38
 native methods, 821
 native object prototypes, 196
 navigating windows, 247–251
 navigator object, 259–265
 methods, 259–261
 properties, 259–262
 negative zero, 36
 nested loops, 73, 74
 Netscape Navigator
 DOM support, 9
 ECMAScript support, 6
 JavaScript version
 progression, 10
 user-agent detection, 278–280

networkState, 600
 new line `(\n)`, 42
 new operator
 constructors and, 733
 Object instances, 44, 103, 104
`nextElementSibling`, 360
`nextNode()`, 411, 412, 413
`nextSibling()`, 413, 414
 Nintendo Wii, 301, 302, 475, 477
`$nn`, 158
 NodeFilter type, 410
 NodeIterator type, 410–413
`nodeKind()`, 679–680
 NodeList objects, 312, 353–354
 nodeName property, 311
 nodes. *See also DOM*
 attribute, 345–346
 CDATASection, 342–343
 children property, 374
 comment, 341–342
 contains(), 374–376
 Document, 316–326
 DocumentFragment, 344–345
 DocumentType, 343–344
 DOM Level 2 changes, 384
 E4X and, 679–681
 Element, 326–337
 hierarchy, 310
 HTML5, 361–372, 380
 insertion, mutation events, 481–482
 manipulating, 314–315
 markup insertion
 innerHTML property, 367–369
 innerText property, 376–378
 insertAdjacentHTML(), 370–371
 outerHTML property, 370, 371, 638
 outerText property, 376, 378
 relationships, 312–314, 375–376
 removal, mutation events, 480–481
 Text, 337–341
 nodeValue property, 311
 Nokia Nseries, 300
 non-extensible objects, 744
 nonfatal errors, 628–629
`normalize()`, 316, 340, 341
 normalizing text nodes, 340–341

NoScope elements, 368–369
`<noscript>` element, 22, 23
 NOT
 bitwise NOT operator (~), 51
 logical NOT operator (!), 56–57
 not a number. *See* NaN
 not identically equal operator (!==), 66–67
 not-equal operator (!=), 65–66
 novalidate attribute, 533–534
 null comparisons, 812–813
 Null data type, 33–34
`Number()`, 38–39
`Number` type, 35–41
 number conversions, 38–41
 primitive wrapper, 146, 149–151
 range of values, 37

O

O notation, 816–817
 objects. *See also* reference types;
 specific objects
 built-in, 161–171, 215
 COM, 97, 98, 227, 228, 263, 273, 309, 312, 637, 638, 732
 constructor pattern, 181–184
 durable, 200–201
 hybrid constructor/
 prototype pattern, 197–198
 parasitic, 199–200
 creating, 173–174
 defined, 173
 durable, 200–201
 factory pattern, 180–181
 frozen, 745–746
 functions as, 136, 143, 170
 as hash tables, 173
 host, 45, 273, 274
 JSON, 692–693, 695–696
 NodeList, 312, 353–354
 non-extensible, 744
 ownership, respecting, 809–810
 properties
 accessor, 176–178
 data, 174–176
 defining multiple
 properties, 178–179
 reading property
 attributes, 179–180
 types of, 174

prototype pattern, 184–197
 dynamic, 198–199
 hybrid constructor/
 prototype pattern, 197–198
 reference types *v.*, 103, 170, 173
 sealed, 744–745
 strict mode, 878–879
 tamper-proof, 743–746, 764, 809
 Object data type, 44–45, 104–106
 methods, 45
 new operator, 44, 103, 104
 properties, 45
 object definitions, 103. *See also* reference types
 “object expected,” 607, 617
 object literals
 notation, 104–105
 usage, 823
 object masquerading, 207
 object stores, IndexedDB, 788–789
 object URLs, 847–848
`Object.defineProperty()`, 176, 177, 178, 194, 744, 866, 875
`Object.freeze()`, 745, 866
 object-oriented (OO) programming, 173–216
 ECMAScript *v.*, 103
 inheritance, 201–215
 classical, 207–208
 combination, 209–210, 215
 constructor stealing, 207–208
 parasitic, 211–212, 216
 parasitic combination, 212–215, 216
 prototypal, 210–211
 prototype chaining, 202–207, 215
 pseudoclassical, 209–210
 observer pattern, 431, 755, 764
 O’Callahan, Robert, 837
 octal literals, 35, 40, 882–883
 offline web applications, 765–800
 application cache, 766–768
 data storage
 cookies, 768–778
 IndexedDB, 786–799
 Internet Explorer
 persistent user data, 778–779
 Web Storage, 780–786
 offline detection, 765–766
 offset dimensions, 401–403
 onerror event handler, 622, 640, 715, 716, 720, 721, 791, 853
 onunload, 456, 489, 502
 OO programming. *See* object-oriented programming
`open()`, 325, 326, 639, 703–704
 OpenGL, 571, 576, 577, 578, 579, 580, 581, 582, 587, 589. *See also* WebGL
 OpenGL Shading Language (GLSL), 580–584, 589
OpenGL Shading Language (Rost), 582
 opening windows, 247–251
 open-source projects
 Backbone.js, 887
 code conventions, 802
 Dojo Toolkit, 886
 jQuery, 357, 886, 887, 894
 Konqueror, 283
 MochiKit, 886–887
 MooTools, 886
 Mozilla Project, 5, 10, 284
 Prototype library, 810, 886
 qooxdoo, 887–888
 Rico library, 887
 WebKit, 282
 Yahoo! User Interface Library, 811, 885, 893
 Opera
 DOM support, 9
 ECMAScript support, 6
 error reporting, 612–613
 user-agent detection, 284–285
 “operation aborted,” 635–637
 operators, 45–68. *See also* specific operators
 opting-in, to strict mode, 877–878
 options. *See also* select boxes
 adding, 537–538
 moving, 539
 removing, 538–539
 reordering, 539
 selecting, 536–537
 OR
 bitwise OR operator (`||`), 52–53
 logical OR operator (`||`), 58–59
 orientationchange, 490–491
 outdent, 544
 outerHTML property, 370, 371, 638
 outerText property, 376, 378
 overloading functions, 83, 138
`overrideMimeType()`, 711–712

P

page, logging debugging messages to, 633–634
 page coordinates, 462
 Page Visibility API, 839–841, 856
 pagehide, 487–489
 pageshow, 487–489
 parameters, XSLTProcessor, 666–667
 parasitic combination inheritance, 212–215, 216
 parasitic constructor pattern, 199–200
 parasitic inheritance, 211–212, 216
 parentNode(), 413
 parentWindow, 387
 parse(), 122–123, 592, 695, 696, 699, 700
 parseError property, 646
 parseFloat(), 41
 parseFromString(), 642, 643
 parseInt(), 39–41, 162, 239, 883
 parseXml(), 649, 650
 parsing
 E4X, 685–686
 JSON, 694–700
 partial function application, 741–743
 partial text selection, 523–524
 passing values, to shaders, 583–584
 paste, 526, 545
 paths, 556–557
 pattern attribute, 532
 pattern matching, 133, 136, 156–159, 290, 297, 531. *See also* regular expressions
 patterns, 567
 pause, 601
 pause(), 602–603
 paused, 600
 performance
 best practices, 814–827, 834
 bitwise operators, 821
 double interpretation, 820–821
 event handler removal, 500–502
 issues
 DOM manipulations, 355
 event delegation, 498–500, 826
 events, 509
 garbage collection, 98–99

markup insertion, DOM, 371–372
 loop optimization, 817–819
 minimize statement count, 821–823
 native methods, 821
 optimize DOM interactions, 824–827
 property lookup, 816–817
 rolling loops, 819–820
 scope-awareness, 814–816
 switch statements, 821
 performance.navigation property, 851
 performance.timing property, 851–852
 Perl, 1, 25, 59, 128, 136
Persistent Client State - HTTP Cookies, 768
 persistent user data, 778–779
 Person(), 182, 183
 PHP, 15, 42, 709, 722, 806, 850
 pings, image, 719–721
 pixels, reading, 587–588
 platform identification, 294–295.
See also user-agent detection
 play, 601
 play(), 602–603, 604–605
 playbackRate, 600
 played, 600
 playing, 601
 Playstation, 301, 302
 plug-ins, detecting, 262–264
 plus
 add operator (+), 61–62
 increment operator (++), 46–48
 unary plus operator (+), 48–49
 PNG encoding, 552
 points, WebGL, 584, 585
 pop(), 112–113
 pop-up blockers, 250–251
 pop-up windows, 247–250
 positive zero, 36
 POST requests, 639, 708–709, 714, 715, 716, 717, 728, 849
 postMessage(), 591–592, 852–853
 post-test loops, 70, 818, 819
 pow(), 169
 pragma, 27, 877
 preflighted requests, 717–718
 premultipliedAlpha, 576
 prependChild(), 684
 preserveDrawingBuffer, 576
 pretest loops, 70, 71, 819
 preventDefault(), 443, 445, 447, 448, 450, 482, 495, 513, 525
 previousElementSibling, 360
 previousNode(), 411, 412, 413
 previousSibling(), 413
 primitive types (data types), 31–45.
See also reference types; specific data types
 errors, 625–627
 type checking, 625, 627, 640, 812, 870
 typeof operator, 31, 89–90
 primitive values, 85–90
 copying, 86–87
 defined, 85, 100
 dynamic properties, 86
 reference values *v.* 85
 primitive wrapper types, 146–161, 170–171
 private members, 231, 872
 private variables, 231–234
 privileged method, 231–233, 234, 235, 238
Professional Ajax, 2nd Edition (Wiley), 729
 programming practices, 809–814
 constants, 813–814
 global variables/functions, 810–812, 814–815
 null comparisons, 812–813
 object ownership, 809–810
 progress, 601
 progress event, 713–714
 progress events, 712–714
 Progress Events specification, 712
 prompt(), 253, 254
 properties. *See also* objects; specific properties
 accessor, 176–178
 data, 174–176
 defining multiple properties, 178–179
 lookup, 816–817
 reading property attributes, 179–180
 types of, 174
 propertyIsEnumerable(), 45, 191, 204
 proprietary DOM extensions, 372–379, 380
 proprietary events, 431, 451, 494, 495, 509
 prototypal inheritance, 210–211
 [[Prototype]], 185, 186, 189, 195, 203, 204, 213

prototypes
 alternate syntax, 192–194
 default, 203–204
 dynamic nature, 194–196
 how work, 185–189
 instances and, 204–205
 native object, 196
 in operator and, 189–192
 problems with, 196–197

prototype chaining, 202–207, 215

Prototype JavaScript library, 810, 886

prototype pattern, 184–197
 dynamic, 198–199
 hybrid constructor/prototype pattern, 197–198

prototype property, 144, 184, 185, 187, 189, 190, 191, 193, 197, 215

proxy functions, 868

proxy objects, 865–868

pseudoclassical inheritance, 209–210

`push()`, 112–113

`pushState()`, 606

`putImageData()`, 569

Python, 863, 894

Q

`QName` type, 675–676

qooxdoo, 887–888

`quadraticCurveTo()`, 556

query string arguments, location object, 256–257

`queryCommandEnabled()`, 546

`queryCommandState()`, 546

`queryCommandValue()`, 546

querying
 with cursors, 791–793
 E4X, 681–682, 690

`querySelector()`, 358, 359

`querySelectorAll()`, 358–359, 379

queue methods, arrays, 113–114

quirks detection, 275–276, 307

quirks mode, 21–22, 245, 365, 373, 374, 392, 404, 405, 457, 462

qUnit, 894

R

`random()`, 168–169

range of values, `Number` type, 37

`RangeError`, 616

ranges. *See also* Traversal and Range module
 DOM, 415–424
 clean up, 424
 cloning, 424
 collapsing, 422–423
 comparing, 423–424
 complex selection, 417–419
 inserting content, 421–422
 interacting with content, 419–421
 simple selection, 416–417

Internet Explorer, 424–428
 cloning, 428
 collapsing, 427
 comparing, 427–428
 complex selection, 425–426
 simple selection, 424–425

`ratechange`, 601

raw image data, 567–569

readability, code conventions, 803

`readAsArrayBuffer()`, 844

`readAsBinaryString()`, 844

`readAsDataURL()`, 844

`readAsText()`, 844

reading pixels, 587–588

reading property attributes, 179–180

`readPixels()`, 587–588

`readyState` property, 365, 485, 486, 600, 601, 648, 705

`readystatechange`, 485–487

`rect()`, 556

rectangles, 553–556

recursive functions, 220–221, 237

`reduce()`, 121, 122

`reduceRight()`, 121, 122

reduction methods, arrays, 121–122

reference counting garbage collection, 97–98, 101

reference types, 103–171. *See also* primitive wrapper types
 classes *v.*, 103, 170
 object definitions, 103
 objects *v.*, 103, 170, 173
 primitive wrapper types *v.*, 147

reference values, 85–90. *See also* objects
 copying, 86–87
 defined, 85, 100, 103
 dynamic properties, 86
 primitive values *v.*, 85

`ReferenceError`, 616

`refresh()`, 264

`RegExp` constructor
 properties, 134–136
 regular expression creation, 129–130

`RegExp` instance
 methods, 132–134
 properties, 131

`RegExp` type, 128–136, 170

`registerContentHandler()`, 264–265

registering handlers, 264–265

`registerProtocolHandler()`, 264–265

regular expressions, 128–136
 creating, 128–130
 flags, 128–129
 metacharacters, 129–130
 support, 136
`typeof` operator, 31, 90

related elements, mouse events, 464–466

relational operators, 63–65

relationships, node, 312–314, 375–376

remainder operator (%), 60–61

remote scripting, 701. *See also* Ajax

`removeAttribute()`, 330, 333, 334, 345, 346, 779, 800

`removeAttributeNS()`, 385

`removeChild()`, 315, 318, 480, 500, 538

`removeEventListener()`, 438, 439, 443

`removeformat`, 545

`removeHandler()`, 441, 442, 756, 757

`removeNamedItem()`, 333, 334

`removeNamedItemNS()`, 386

`removeNamespace()`, 686

removing event handlers, 500–502

removing options, 538–539

rendering engine, identifying, 286–291

reordering methods, arrays, 114–116

reordering options, 539

repeating timers, 748–750

`replace()`, 157, 158, 258, 269, 684

`replaceChild()`, 315, 318, 335, 480, 481, 500

`replaceData()`, 338

replacement, `splice()`, 117

`replaceState()`, 605–606

`requestAnimationFrame()`, 835–839, 856
requests
 credentialed, 718
 GET, 703, 707–708, 709, 714, 715, 717, 718, 728, 851
 POST, 639, 708–709, 714, 715, 716, 717, 728, 849
 preflighted, 717–718
 reserved words, 28–29. *See also*
 keywords
 resetting forms, 513–514
 Resig, John, 810, 886
 `resize` event, 452, 456–457
 respect object ownership, 809–810
 rest arguments, 859–860
 retrieving selected text, 522
 `returnValue`, 447
 `reverse()`, 114, 115, 116, 626, 627
 `reverseSort()`, 626, 627
 RGB, 395, 553
 Rhino, 829, 830, 831, 891, 892, 893
 rich text editing, 542–549, 550
 Rico library, 887
 rightContext property, 134–135
 Rost, Rani J., 582
 `rotate()`, 560
 rounding errors, 36–37
 rounding methods, `Math` object, 167–168
 Ruby, 806
 Ruby on Rails, 886
 Russell, Alex, 721, 886

S

Safari
 DOM support, 9
 ECMAScript support, 6
 error reporting, 610–612
 WebKit, 282–283
 safe type detection, 110, 621, 731–733
 sandbox, 593
 `save()`, 562
 `sayHi()`, 79, 80, 81, 144, 164, 194, 212, 218, 219
 `sayName()`, 144, 174, 183, 184, 201, 810, 811
 Scalable Vector Graphics (SVG), 8, 383, 385
 `scale()`, 560
 scope chains
 augmentation, 92–93

 closures and, 221–224
 defined, 91
 performance, 814
 scoped element, 368–369
 scopes (execution contexts), 90–96
 block-level scopes, 93–96, 228–230, 858–859
 defined, 90, 100
 ECMAScript Harmony, 858–859
 functions, 90–91, 100
 global, 90, 100
 performance *v.*, 814–816
 worker global scope, 853–855
 scope-safe constructors, 733–735
 screen coordinates, 462–463
 screen object, 265–267
 script debugging, 608
 `<script>` elements, 13–23
 `async` attribute, 13, 17–18, 23
 asynchronous scripts, 17–18
 within `<body>`, 16
 `defer` attribute, 13, 16–17, 23
 deferred scripts, 16–17
 deprecated syntax, 20
 DOM and, 346–348
 external files, 15–16, 20
 inline JavaScript code, 14–15, 20
 `src` attribute, 14, 15
 `type` attribute, 14
 XHTML, 18–19
 `script.aculo.us`, 888
 scripting forms. *See* forms
 scroll dimensions, 404–406
 scroll event, 452, 457
 `scrollByLines()`, 379
 `scrollByPages()`, 379
 `scrollIntoView()`, 372, 379
 `scrollIntoViewIfNeeded()`, 379
 sealed objects, 744–745
 `search()`, 157
 secure execution environments, 895
 security
 Ajax/Comet, 728–729
 code injection, 164
 cookies, 778
 CORS, 714
 cross-document messaging, 591, 606
 cryptography libraries, 888–889
 CSRF, 715, 728
 Internet Explorer persistent user data, 779
 memory management, 99
 pop-up windows, 249–250
 this value, coercion of, 882
 Web Sockets, 725
 XSS attacks, 715
 seekable, 600
 seeked, 601
 seeking, 600, 601
 select, 452
 select boxes, 534–539, 550
 creating, 534
 options
 adding, 537–538
 moving, 539
 removing, 538–539
 reordering, 539
 selecting, 536–537
 select event, 521–522
 selectAll, 545
 selecting options, 536–537
 selecting text, 521–524, 549
 `selectNodes()`, 656, 657, 659, 660, 669
 Selectors API, 357–360, 379
 `matchesSelector()`, 359–360
 `querySelector()`, 358, 359
 `querySelectorAll()`, 358–359, 379
 `selectSingleNode()`, 656, 657, 658, 659, 660, 669
 semicolon
 JSON objects, 693
 statements, 27
 `send()`, 704, 705, 708, 710, 712, 715, 850
 serialization
 E4X, 685–686
 Internet Explorer, 647
 JSON, 694–700
 `serialize()`, 541, 542, 709
 `serializeToString()`, 644
 `serializeXml()`, 650
 server push, 721
 Server-Sent Events (SSE), 723–725, 727–728
 sessionStorage object, 781–783
 `set()`, 772, 775, 776
 `[[Set]]`, 177, 744, 745
 Set type, 869
 `setAll()`, 775, 776, 777
 `setAttribute()`, 330, 332, 333, 334, 345, 346, 521, 778, 779, 800
 `setAttributeNode()`, 346
 `setAttributeNodeNS()`, 386

setAttributeNS(), 386
setChildren(), 684
setData(), 527, 528, 595
setDate(), 127
setDragImage(), 598
setFullYear(), 126
setHours(), 127
setInterval(), 252, 836, 854
setMilliseconds(), 128
setMinutes(), 127
setMonth(), 127
setName(), 88, 89, 232, 234, 676
setNamedItem(), 333, 334
setNamedItemNS(), 386
setNamespace(), 686
setSelectionRange(), 523–524
setters/getters, 872–873
setTime(), 126
setTimeout(), 251–252, 638,
 722, 741, 746, 747, 749, 753,
 764, 820, 836, 837, 854
setting attributes, 332–333
setTransform(), 560
setUserData(), 389
setUTCDate(), 127
setUTCFullYear(), 126
setUTCHours(), 127
setUTCMilliseconds(), 128
setUTCMinutes(), 127
setUTCMonth(), 127
setUTCSeconds(), 128
SHA-1, 889
shaders, 580–584
shadowBlur, 564
shadowColor, 564
shadowOffsetX, 564
shadowOffsetY, 564
shadows, 564
shared workers, 855
shift(), 113, 751
short polling, 721
showMessage(), 435, 436
ShrinkSafe, 892–893
side effects, 21, 37, 46, 51, 87, 223,
 224, 809
sign bit, 50
signed right shift (`>>`), 54–55
signed right shift/assign (`>>=`), 68
simple maps, 868–869
simple selection
 DOM ranges, 416–417
 Internet Explorer ranges,
 424–425
simple values, JSON, 692
simulating events, 502–509
 DOM event simulation,
 502–508

Internet Explorer event
 simulation, 508–509
 keyboard events, 504–506
 mouse events, 503–504
sin(), 170
single line comment (`//`), 26
single quotes (`'`), strings, 41–42
singleton built-in objects, 161–171
singletons, 234–237, 238, 759, 762
64-bit to 32-bit conversion, 49–51
size, window objects, 245–246
slice(), 116–117, 153, 154, 312,
 741, 742, 846
small(), 161
SMIL (Synchronized Multimedia
 Integration Language), 8
Smith, Garrett, 436
social networking applications, 593
some(), 119, 120
sort(), 114, 115, 116, 140, 141,
 196, 273, 620, 626, 812
source, RegExp instance
 property, 131
special collections, document
 object, 322–323
specialty libraries, 885
Speed Up Your Site (King), 820
splice(), 117–118, 756
split(), 159
splitText(), 338, 341, 342
splitting text nodes, 341
spoofing, 277, 728
spread arguments, 859–860
sqrt(), 169
square brackets, 107, 108, 129, 174
src attribute, 14, 15
src property, 600
srcElement, 447
SSE (Server-Sent Events), 723–725,
 727–728
stack methods, arrays, 112–113
stalled, 602
standards mode, 21–22, 245, 246,
 365, 373, 374, 392, 404, 405,
 457, 462, 807
start, 600
startsWith(), 196
statements (flow-control
 statements). *See also specific
 statements*
 count, minimizing, 821–823
 curly braces, 27, 104
 labeled, 73
 semicolon and, 27
 syntax, 27–28
static private variables, 232–234
stencil, 576
Stephenson, Sam, 886
stopImmediatePro
 pagation(), 443
stopPropagation(), 444, 446,
 447, 448, 449, 450, 451
storage event, 785–786
Storage type, 780–781
streaming, HTTP, 722–723,
 724, 729
strict mode, 877–883
 defined, 27
 eval(), 163–164, 880–881
 functions, 80, 879–880
 object manipulation,
 878–879
 octal literals, 882–883
 opting-in, 877–878
 pragma, 27, 877
 variables, 878
stride value, 586
strike(), 161
String(), 44, 62, 114
String type, 41–45
 character methods, 152
 HTML methods, 161
 pattern matching methods,
 156–159
 primitive wrapper type, 146,
 151–161
stringify(), 592, 695, 700
strings
 case conversion, 156
 character literals, 42–43
 converting to, 43–44
 double quotes (“”), 41–42
 JSON, 692
 location methods, 154–155
 manipulation methods,
 152–154
 nature of, 43
 relational operators and, 64
 single quotes ('), 41–42
stroke(), 556, 557
strokeRect(), 553, 554, 556
strokes, 553
strokeStyle, 553
strokeText(), 557, 558, 559
struct types, 869–870
<style> element, 348–350, 396
style sheets, 396–401
 CSS rules, 398–401
 creating, 399–400
 deleting, 400–401
styles
 accessing element styles,
 391–392
 computed, 394–396

DOM, properties/methods, 392–394
 DOM and, 348–350
 DOM Level 2 Styles module, 381, 390–401, 429
StyleSheet property, 350, 396
sub(), 161
subarray(), 575
subcookies, 773–778
 submitting forms, 512–513
substr(), 153, 154
substring(), 147, 153, 154, 196, 522, 523, 625
substringData(), 338
subtract/assign (-=), 68
SubType, 202–203, 213
sum(), 79, 137, 144, 860
sup(), 161
SuperType, 202–203, 213
SVG (Scalable Vector Graphics), 8, 383, 385
 switch statements, 76–78, 821
Synchronized Multimedia Integration Language (SMIL), 8
syntax
 ECMAScript, 25–28
 errors, Internet Explorer, 638–639
 JSON, 691–694
SyntaxError, 617
 “system cannot locate resource specified,” 639
 system dialogs, 253–255

T

tab (\t), 42
tab forward behavior, 528–529
tabForward(), 529
<table> element, 350–353
 tables, manipulating, 350–353
 tag placement, **<script>** elements, 16
 tamper-proof objects, 743–746, 764, 809
tan(), 170
target, 444
 targets, drop, 594
<tbody>, 352, 353
test(), 30, 130, 131, 133, 134, 136, 288
 test-driven development, 893
text
 draggable, 597

drawing, 557–559
text boxes, 520–534
 input filtering, 524–528
 text selection, 521–524, 549
text events, 472, 476–477
text nodes, 337–341
 creating, 339–340
 normalizing, 340–341
 splitting, 341
text selection, 521–524, 549
Text type, 337–341
<textarea> element, 520–521
textContent property, 377, 378
textInput event, 471, 472, 476–477
textures, WebGL, 587
Theora, 604
 32-bit conversion, 64-bit to, 49–51
this object, 225–227
this value, coercion of, 882
 3D drawing context, 551, 571, 589.
See also WebGL
 3D graphics languages, 571
throttle(), 753–754
throw operator, 619
 throwing errors, 619–622, 634–635
 tightly coupled software, 805–806
 time-date component methods, 126–128
timeouts
 intervals and, 251–253
 XHR, 711
timers
 advanced, 746–754
 function throttling, 752–754
 repeating, 748–750
setInterval(), 252, 836, 854
setTimeout(), 746, 747, 749, 753
 yielding processes, 750–752
timeupdate, 602
toDataURL(), 552, 553, 563, 564
toDateString(), 125
toExponential(), 150, 151
toFixed(), 149, 150, 151
toGMTString(), 126, 772
toJSON(), 698–699
toLocaleDateString(), 126
toLocaleLowerCase(), 156
toLocaleString(), 45
 arrays, 110–111
 Date type, 124–125
 functions, 146
 Number type, 149
 regular expressions, 133–134
String type, 151
toLocaleTimeString(), 126
toLocaleUpperCase(), 156
toLowerCase(), 156
 tools, JavaScript, 891–895
toPrecision(), 150, 151
toString()
 arrays, 110–111
 converting to string, 43–44, 45
Date type, 124–125
 functions, 146
 Number type, 149
 regular expressions, 133–134
String type, 151
totalBytes, 600
toTimeString(), 126
 touch devices, mouse events, 470–471
touch events, 494–497
 properties, 495–496
touchcancel, 495
touchend, 495
touchmove, 495
touchstart, 495
toUpperCase(), 156
toUTCString(), 126
toXMLString(), 672, 673, 674, 685
<tr>, 352, 353
 transactions, IndexedDB, 790–791
transform(), 560
transformNode(), 660–661
transformToDocument(), 665, 666, 667, 668
transformToFragment(), 665, 666, 667
translate(), 560
 traps, proxy objects, 866–867
 Traversals and Range module, 381, 408–428, 429
 traversals, 408–415, 429
NodeIterator type, 410–413
TreeWalker, 413–415
TreeWalker, 413–415
 triangles, WebGL, 584–586
trim(), 155–156
trusted, 444
try-catch statement, 615–619
 ECMAScript, 607
 error types, 616–618
 finally clause, 616
 throwing errors *v.*, 621–622
 usage, 618–619

2D drawing context, 553–570.

See also WebGL
compositing, 569–570
defined, 588–589
fills, 553
gradients, 565–567
images, 563–564
paths, 556–557
patterns, 567
raw image data, 567–569
rectangles, 553–556
shadows, 564
strokes, 553
text, 557–559
transformations, 559–562
two's complement, 50–51
type attribute, 14
type checking, 625, 627, 640, 812, 870
type coercion errors, 624–625
type comments, 804–805
type detection, safe, 110, 621, 731–733
type property, 443, 445, 447, 517, 531, 534, 540, 542, 756
typed arrays, 571–576
typed views, 573–576
`TypeError`, 617
`typeof` operator, 31, 89–90

U

UI events, 452–457
`Uint8Array`, 573
`Uint16Array`, 574
`Uint32Array`, 574
unary minus operator `(-)`, 48–49
unary operators, 46–49
unary plus operator `(+)`, 48–49
Undefined data type, 32–33
undefined value, 29, 32, 33, 34, 39, 44, 691
underline, 545
`Underscore.js`, 887
understandable, maintainable code, 802
unexpected identifier error, 139
uniforms, attributes and, 581
unit testing, 893–894
Universal Time Code. *See* UTC
unknown runtime error, 638
`unlink`, 545
unload event, 452, 456
unrolling loops, 819–820
`unset()`, 773, 777

`unsetAll()`, 777
`unshift()`, 113–114
unsigned integer, 51
unsigned right shift `(>>)`, 55–56
unsigned right shift/
 `assign (>>=)`, 68
URI-encoding methods, Global object, 162–163
`URIError`, 617
URLs
 blob, 847–848
 constants and, 813, 814
 history object, 267
 malformed, 627
 XHR usage, 704, 728
user data, persistent, 778–779
`userAgent`, 261, 269, 854
user-agent detection, 276–306
 defined, 307
 history, 277–286
 identifying
 browsers, 291–294
 game systems, 301–302
 mobile devices, 298–301
 platforms, 294–295
 rendering engine, 286–291
 Windows operating systems, 295–298
 script, 303–306
spoofing, 277
when to use, 306
working with, 286–302
UTC (Universal Time Code), 122, 126, 127, 128

V

validation
 form fields, 530–534
 JSLint, 829–830, 891
validators, 891–892
 `[[Value]]` attribute, 175, 744
`valueOf()`, 39, 45
 arrays, 110–111
 Date type, 125
 functions, 146
 Number type, 149
 String type, 151
values, functions as, 139–141
variable object, 90
variables
 closures and, 224–225
 complexities, 85
 declaration, 94–95
 dereferencing, 99, 100, 101

garbage collection, 96–100
global, 810–812, 814–815
loosely typed, 29, 30, 31, 85, 623, 625, 804, 833
naming conventions, 803–804
overview, 29–30
primitive values, 85–90
private, 231–234
reference values, 85–90
strict mode, 878
type transparency,
 804–805
vertex shaders, 580–584
`<video>` element, 598–605
 codec support detection, 603–604
 custom media players, 602–603
events, 601–602
 properties, 599–601
`videoHeight`, 600
`videoWidth`, 600
viewports, WebGL, 578–579
views
 array buffer, 571–573
 typed, 573–576
visibilitychange event, 840
Visual Basic .NET, 895
visual effects/animation
 libraries, 888
 `mozRequestAnimationFrame`
 Frame, 835, 837–839
 `requestAnimationFrame`
 Frame(), 835–839, 856
volume, 600
volumechange, 602
Vorbis, 604

W

waiting, 602
`watchPosition()`, 843
WAV, 604
WeakMap type, 869
web applications. *See* offline web applications
web browsers. *See* browsers
web forms. *See* forms
Web Inspector, 613
Web Messaging, 593
Web Sockets, 725–728
Web Storage, 780–786
 `globalStorage` object, 783–784
limitations/restrictions, 786

localStorage object, 784–785
 sessionStorage object, 781–783
 storage event, 785–786
 Storage type, 780–781
 Web Timing, 851–852, 856
 Web Workers, 852–855, 856
 WebGL, 571–588
 buffers, 579–580
 constants, 577
 context, 576–588
 coordinates, 578–579
 defined, 589
 drawing, 584–586
 errors, 580
 GLSL, 580–584, 589
 JavaScript *v.*, 580
 method names, 578
 OpenGL, 571, 576, 577, 578, 579, 580, 581, 582, 587, 589
 pixels, 587–588
 shaders, 580–584
 support, 588
 textures, 587
 3D drawing context, 551, 571, 589
 triangles, 584–586
 typed arrays, 571–576
 typed views, 573–576
 viewports, 578–579
 webgl, 576
 WebKit, 282–283
 webkitRequestAnimationFrame
 Frame(), 838–839
 WebM, 604
 whatToShow, 410–411
 wheel events, 451, 460
 while statement, 70
 white space
 Element Traversal, 360
 JSON.stringify(), 697
 semicolons, 27
 trim(), 155
 width attribute, 552
 Wii, Nintendo, 301, 302, 475, 477
 Wiley, *Professional Ajax, 2nd Edition*, 729
 window objects, 165–166, 239–255, 268. *See also* Global object

global scope, 240–241
 intervals/timeouts, 251–253
 system dialogs, 253–255
 window.open(), 243, 247, 248, 249, 250, 251
 windows
 frames *v.*, 241–244
 navigating, 247–251
 opening, 247–251
 pop-up, 247–250
 position, 244–245
 size, 245–246
 Windows Mobile, 241, 300, 301, 305, 492
 Windows operating systems
 identification, 295–298. *See also* user-agent detection
 wire weight, 830
 with statement, 75–76, 815–816
 workers
 dedicated, 855
 global scope, 853–855
 shared, 855
 wrapper types. *See* primitive
 wrapper types
 [[Writable]] attribute, 175, 744, 745
 write(), 324, 325, 326
 writeln(), 324, 325, 326
 writing, document, 324–326
 WYSIWYG editing, 542

X

XDM. *See* cross-document messaging
 XHR (XMLHttpRequest), 701–712
 createXHR(), 702, 703, 736, 737, 738
 file uploads with, 849–850, 856
 FormData type, 710–711
 GET requests, 707–708
 HTTP headers, 706–707
 Level 1, 710
 Level 2, 710
 overrideMimeType(), 711–712
 POST requests, 708–709
 progress events, 712–714

send(), 704, 705, 708, 710, 712, 715, 850
 timeouts, 711
 usage, 703–706
 XHTML (Extensible HyperText Markup Language)
 HTML *v.*, 18
 innerHTML property, 369
 <script> elements, 18–19
 XML namespaces, 382–386
 XML, 641–669. *See also* Ajax;
 ECMAScript for XML;
 elements
 construction/manipulation,
 E4X, 682–685
 DOM support in browsers,
 641–650
 in Internet Explorer, 644–649
 JSON *v.*, 691
 literals, 672, 673, 675, 682, 685
 namespaces, 382–386, 655–656
 XML type, 672–673
 XMLHttpRequest. *See* XHR
 XMLList type, 673–674
 xmlns attribute, 382, 383, 384, 655
 XMLSerializer type, 644, 650, 669
 XPath support, browsers, 651–660
 XPathVariable values, 651–653
 XSLT support, browsers, 660–668
 XSLTProcessor type, 665–667
 XSS (cross-site scripting)
 attacks, 715

Y

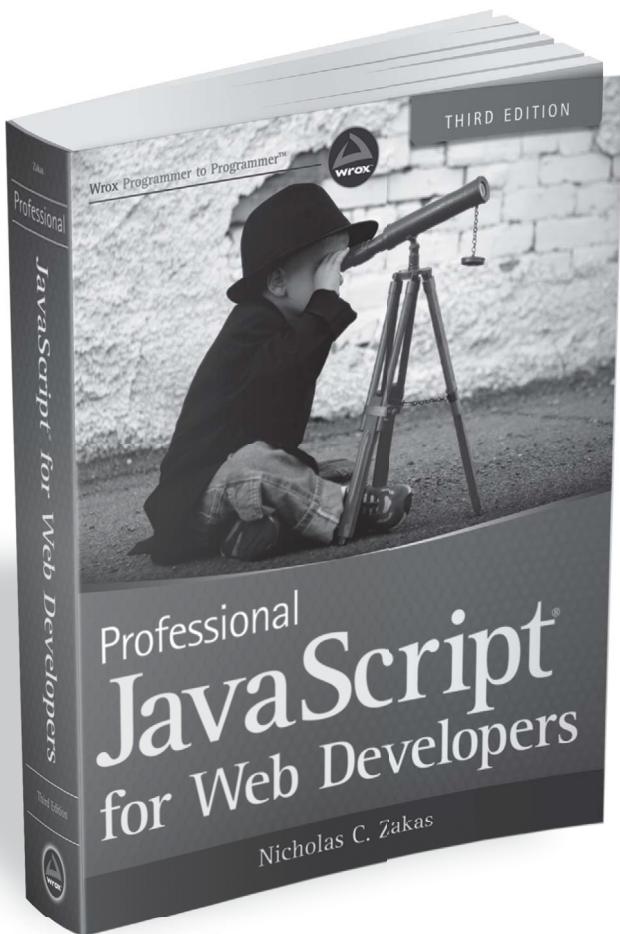
Yahoo! User Interface Library (YUI), 811, 885, 893
 yielding processes, 750–752
 YUI Compressor, 831–832, 893
 YUI Doc, 894
 YUI Test, 893

Z

zero, positive/negative, 36

Try Safari Books Online FREE for 15 days + 15% off for up to 12 Months*

Read this book for free online—along with thousands of others—
with this 15-day trial offer.



With Safari Books Online, you can experience searchable, unlimited access to thousands of technology, digital media and professional development books and videos from dozens of leading publishers. With one low monthly or yearly subscription price, you get:

- Access to hundreds of expert-led instructional videos on today's hottest topics.
- Sample code to help accelerate a wide variety of software projects
- Robust organizing features including favorites, highlights, tags, notes, mash-ups and more
- Mobile access using any device with a browser
- Rough Cuts pre-published manuscripts

START YOUR FREE TRIAL TODAY!

Visit www.safaribooksonline.com/wrox27 to get started.

*Available to new subscribers only. Discount applies to the Safari Library and is valid for first 12 consecutive monthly billing cycles. Safari Library is not available in all countries.



An Imprint of **WILEY**
Now you know.



Programmer to Programmer™

Connect with Wrox.

Participate

Take an active role online by participating in our P2P forums @ p2p.wrox.com

Wrox Blox

Download short informational pieces and code to keep you up to date and out of trouble

Join the Community

Sign up for our free monthly newsletter at newsletter.wrox.com

Wrox.com

Browse the vast selection of Wrox titles, e-books, and blogs and find exactly what you need

User Group Program

Become a member and take advantage of all the benefits

Wrox on twitter

Follow @wrox on Twitter and be in the know on the latest news in the world of Wrox

Wrox on facebook

Join the Wrox Facebook page at facebook.com/wroxpress and get updates on new books and publications as well as upcoming programmer conferences and user group events

Contact Us.

We love feedback! Have a book idea? Need community support?
Let us know by e-mailing wrox-partnerwithus@wrox.com

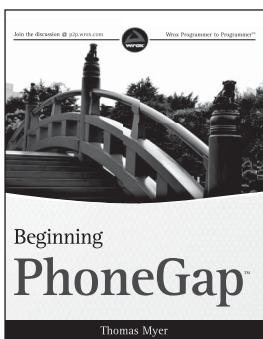
Related Wrox Books



Beginning iOS Application Development with HTML and JavaScript

ISBN: 978-1-1181-5900-2

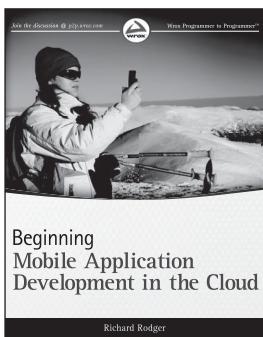
Beginning-to-intermediate web developers who want to apply their existing HTML/CSS/JavaScript skills to app development for the iPhone/iPad OS will love this book. The book enables developers who know these core technologies to use what they already know and get up to speed quickly. It introduces iOS development with web technologies, explains how to enable and optimize websites for the iPhone and iPad, explores user interface design, then moves into animation, special effects, building with web frameworks, and much more.



Beginning PhoneGap

ISBN: 978-1-1181-5665-0

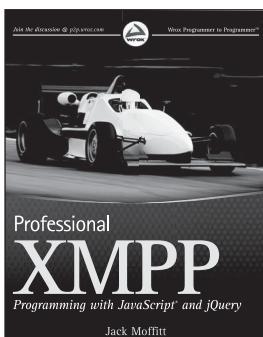
PhoneGap is an open source solution, ideal for web developers wanting to build cross-platform mobile apps without having to learn a new language. Using HTML, CSS, and Javascript, *Beginning PhoneGap* allows you to jump into the mobile world and develop apps for iPhone, Android, and the BlackBerry.



Beginning Building Mobile Application Development in the Cloud

ISBN: 978-1-1180-3469-9

The marketplace for apps is ever expanding, increasing the potential to make money. With this guide, you'll learn how to build cross-platform applications for mobile devices that are supported by the power of Cloud-based services such as Amazon Web Services. An introduction to Cloud-based applications explains how to use HTML5 to create cross-platform mobile apps and then use Cloud services to enhance those apps. You'll learn how to build your first app with HTML5 and set it up in the Cloud, while also discovering how to use jQuery to your advantage.



Professional XMPP Programming with JavaScript and jQuery

ISBN: 978-0-470-54071-8

XMPP is a robust protocol used for a wide range of applications, including instant messaging, multi-user chat, voice and video conferencing, collaborative spaces, real-time gaming, data synchronization, and search. This book teaches you how to harness the power of XMPP in your own apps and presents you with all the tools you need to build the next generation of apps using XMPP or add new features to your current apps. Featuring the JavaScript language throughout and making use of the jQuery library, the book contains several XMPP apps of increasing complexity that serve as ideal learning tools.