



Christian Johansen

Test-Driven JavaScript Development

Developer's Library



FREE SAMPLE CHAPTER

SHARE WITH OTHERS



Test-Driven JavaScript Development

Developer's Library Series



Visit developers-library.com for a complete list of available products

The **Developer's Library Series** from Addison-Wesley provides practicing programmers with unique, high-quality references and tutorials on the latest programming languages and technologies they use in their daily work. All books in the Developer's Library are written by expert technology practitioners who are exceptionally skilled at organizing and presenting information in a way that's useful for other programmers.

Developer's Library books cover a wide range of topics, from open-source programming languages and databases, Linux programming, Microsoft, and Java, to Web development, social networking platforms, Mac/iPhone programming, and Android programming.



Test-Driven JavaScript Development

Christian Johansen

◆◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Johansen, Christian, 1982-
Test-driven JavaScript development / Christian Johansen.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-321-68391-5 (pbk. : alk. paper)
ISBN-10: 0-321-68391-9 (pbk. : alk. paper)
1. JavaScript (Computer program language) I. Title.
QA76.73.J39J64 2011
005.13'3--dc22 2010027298

Copyright © 2011 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-68391-5

ISBN-10: 0-321-68391-9

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

Second printing, May 2012

Acquisitions Editor
Trina MacDonald

Development Editor
Songlin Qiu

Managing Editor
John Fuller

Project Editor
Madhu Bhardwaj,
Glyph International

Project Coordinator
Elizabeth Ryan

Copy Editor
Mike Read

Indexer
Robert Swanson

Proofreader
David Daniels

Technical Reviewers
Andrea Giammarchi
Joshua Gross
Jacob Seidelin

Cover Designer
Gary Adair

Compositor
Glyph International

To Frøydis and Kristin, my special ladies.

This page intentionally left blank

Contents

Preface *xix*

Acknowledgments *xxv*

About the Author *xxvii*

Part I Test-Driven Development 1

1. Automated Testing 3

- 1.1 The Unit Test 4
 - 1.1.1 Unit Testing Frameworks 5
 - 1.1.2 `strftime` for JavaScript Dates 5
- 1.2 Assertions 9
 - 1.2.1 Red and Green 10
- 1.3 Test Functions, Cases, and Suites 11
 - 1.3.1 Setup and Teardown 13
- 1.4 Integration Tests 14
- 1.5 Benefits of Unit Tests 16
 - 1.5.1 Regression Testing 16
 - 1.5.2 Refactoring 17
 - 1.5.3 Cross-Browser Testing 17
 - 1.5.4 Other Benefits 17
- 1.6 Pitfalls of Unit Testing 18
- 1.7 Summary 18

2. The Test-Driven Development Process 21

- 2.1 Goal and Purpose of Test-Driven Development 21
 - 2.1.1 Turning Development Upside-Down 22
 - 2.1.2 Design in Test-Driven Development 22

2.2	The Process	23
2.2.1	Step 1: Write a Test	24
2.2.2	Step 2: Watch the Test Fail	25
2.2.3	Step 3: Make the Test Pass	26
2.2.3.1	You Ain't Gonna Need It	26
2.2.3.2	Passing the Test for <code>String.prototype.trim</code>	27
2.2.3.3	The Simplest Solution that Could Possibly Work	27
2.2.4	Step 4: Refactor to Remove Duplication	28
2.2.5	Lather, Rinse, Repeat	29
2.3	Facilitating Test-Driven Development	29
2.4	Benefits of Test-Driven Development	30
2.4.1	Code that Works	30
2.4.2	Honoring the Single Responsibility Principle	30
2.4.3	Forcing Conscious Development	31
2.4.4	Productivity Boost	31
2.5	Summary	31
3.	Tools of the Trade	33
3.1	xUnit Test Frameworks	33
3.1.1	Behavior-Driven Development	34
3.1.2	Continuous Integration	34
3.1.3	Asynchronous Tests	35
3.1.4	Features of xUnit Test Frameworks	35
3.1.4.1	The Test Runner	35
3.1.5	Assertions	36
3.1.6	Dependencies	37
3.2	In-Browser Test Frameworks	37
3.2.1	YUI Test	38
3.2.1.1	Setup	38
3.2.1.2	Running Tests	40
3.2.2	Other In-Browser Testing Frameworks	40
3.3	Headless Testing Frameworks	41
3.3.1	Crosscheck	42
3.3.2	Rhino and <code>env.js</code>	42
3.3.3	The Issue with Headless Test Runners	42
3.4	One Test Runner to Rule Them All	42
3.4.1	How <code>JsTestDriver</code> Works	43
3.4.2	<code>JsTestDriver</code> Disadvantages	44
3.4.3	Setup	44
3.4.3.1	Download the Jar File	44
3.4.3.2	Windows Users	45
3.4.3.3	Start the Server	45
3.4.3.4	Capturing Browsers	46

3.4.3.5	Running Tests	46
3.4.3.6	JsTestDriver and TDD	48
3.4.4	Using JsTestDriver From an IDE	49
3.4.4.1	Installing JsTestDriver in Eclipse	49
3.4.4.2	Running JsTestDriver in Eclipse	50
3.4.5	Improved Command Line Productivity	51
3.4.6	Assertions	51
3.5	Summary	52
4.	Test to Learn	55
4.1	Exploring JavaScript with Unit Tests	55
4.1.1	Pitfalls of Programming by Observation	58
4.1.2	The Sweet Spot for Learning Tests	59
4.1.2.1	Capturing Wisdom Found in the Wild	59
4.1.2.2	Exploring Weird Behavior	59
4.1.2.3	Exploring New Browsers	59
4.1.2.4	Exploring Frameworks	60
4.2	Performance Tests	60
4.2.1	Benchmarks and Relative Performance	60
4.2.2	Profiling and Locating Bottlenecks	68
4.3	Summary	69
Part II	JavaScript for Programmers	71
5.	Functions	73
5.1	Defining Functions	73
5.1.1	Function Declaration	73
5.1.2	Function Expression	74
5.1.3	The Function Constructor	75
5.2	Calling Functions	77
5.2.1	The arguments Object	77
5.2.2	Formal Parameters and arguments	79
5.3	Scope and Execution Context	80
5.3.1	Execution Contexts	81
5.3.2	The Variable Object	81
5.3.3	The Activation Object	82
5.3.4	The Global Object	82
5.3.5	The Scope Chain	83
5.3.6	Function Expressions Revisited	84
5.4	The this Keyword	87
5.4.1	Implicitly Setting this	88
5.4.2	Explicitly Setting this	89
5.4.3	Using Primitives As this	89
5.5	Summary	91

- 6. Applied Functions and Closures 93**
 - 6.1 Binding Functions 93
 - 6.1.1 Losing `this`: A Lightbox Example 93
 - 6.1.2 Fixing `this` via an Anonymous Function 95
 - 6.1.3 `Function.prototype.bind` 95
 - 6.1.4 Binding with Arguments 97
 - 6.1.5 Currying 99
 - 6.2 Immediately Called Anonymous Functions 101
 - 6.2.1 Ad Hoc Scopes 101
 - 6.2.1.1 Avoiding the Global Scope 101
 - 6.2.1.2 Simulating Block Scope 102
 - 6.2.2 Namespaces 103
 - 6.2.2.1 Implementing Namespaces 104
 - 6.2.2.2 Importing Namespaces 106
 - 6.3 Stateful Functions 107
 - 6.3.1 Generating Unique Ids 107
 - 6.3.2 Iterators 109
 - 6.4 Memoization 112
 - 6.5 Summary 115

- 7. Objects and Prototypal Inheritance 117**
 - 7.1 Objects and Properties 117
 - 7.1.1 Property Access 118
 - 7.1.2 The Prototype Chain 119
 - 7.1.3 Extending Objects through the Prototype Chain 121
 - 7.1.4 Enumerable Properties 122
 - 7.1.4.1 `Object.prototype.hasOwnProperty` 124
 - 7.1.5 Property Attributes 126
 - 7.1.5.1 `ReadOnly` 126
 - 7.1.5.2 `DontDelete` 126
 - 7.1.5.3 `DontEnum` 126
 - 7.2 Creating Objects with Constructors 130
 - 7.2.1 `prototype` and `[[Prototype]]` 130
 - 7.2.2 Creating Objects with `new` 131
 - 7.2.3 Constructor Prototypes 132
 - 7.2.3.1 Adding Properties to the Prototype 132
 - 7.2.4 The Problem with Constructors 135
 - 7.3 Pseudo-classical Inheritance 136
 - 7.3.1 The `Inherit` Function 137
 - 7.3.2 Accessing `[[Prototype]]` 138
 - 7.3.3 Implementing `super` 139
 - 7.3.3.1 The `-super` Method 140

7.3.3.2	Performance of the <code>super</code> Method	143
7.3.3.3	A <code>_super</code> Helper Function	143
7.4	Encapsulation and Information Hiding	145
7.4.1	Private Methods	145
7.4.2	Private Members and Privileged Methods	147
7.4.3	Functional Inheritance	148
7.4.3.1	Extending Objects	149
7.5	Object Composition and Mixins	150
7.5.1	The <code>Object.create</code> Method	151
7.5.2	The <code>tddjs.extend</code> Method	153
7.5.3	Mixins	157
7.6	Summary	158
8.	ECMAScript 5th Edition	159
8.1	The Close Future of JavaScript	159
8.2	Updates to the Object Model	161
8.2.1	Property Attributes	161
8.2.2	Prototypal Inheritance	164
8.2.3	Getters and Setters	166
8.2.4	Making Use of Property Attributes	167
8.2.5	Reserved Keywords as Property Identifiers	170
8.3	Strict Mode	171
8.3.1	Enabling Strict Mode	171
8.3.2	Strict Mode Changes	172
8.3.2.1	No Implicit Globals	172
8.3.2.2	Functions	172
8.3.2.3	Objects, Properties, and Variables	174
8.3.2.4	Additional Restrictions	174
8.4	Various Additions and Improvements	174
8.4.1	Native JSON	175
8.4.2	<code>Function.prototype.bind</code>	175
8.4.3	Array Extras	175
8.5	Summary	176
9.	Unobtrusive JavaScript	177
9.1	The Goal of Unobtrusive JavaScript	177
9.2	The Rules of Unobtrusive JavaScript	178
9.2.1	An Obtrusive Tabbed Panel	179
9.2.2	Clean Tabbed Panel Markup	181
9.2.3	TDD and Progressive Enhancement	182
9.3	Do Not Make Assumptions	183
9.3.1	Don't Assume You Are Alone	183
9.3.1.1	How to Avoid	183

- 9.3.2 Don't Assume Markup Is Correct 183
 - 9.3.2.1 How to Avoid 184
- 9.3.3 Don't Assume All Users Are Created Equal 184
 - 9.3.3.1 How to Avoid 184
- 9.3.4 Don't Assume Support 184
- 9.4 When Do the Rules Apply? 184
- 9.5 Unobtrusive Tabbed Panel Example 185
 - 9.5.1 Setting Up the Test 186
 - 9.5.2 The `tabController` Object 187
 - 9.5.3 The `activateTab` Method 190
 - 9.5.4 Using the Tab Controller 192
- 9.6 Summary 196

10. Feature Detection 197

- 10.1 Browser Sniffing 198
 - 10.1.1 User Agent Sniffing 198
 - 10.1.2 Object Detection 199
 - 10.1.3 The State of Browser Sniffing 200
- 10.2 Using Object Detection for Good 200
 - 10.2.1 Testing for Existence 201
 - 10.2.2 Type Checking 201
 - 10.2.3 Native and Host Objects 202
 - 10.2.4 Sample Use Testing 204
 - 10.2.5 When to Test 206
- 10.3 Feature Testing DOM Events 207
- 10.4 Feature Testing CSS Properties 208
- 10.5 Cross-Browser Event Handlers 210
- 10.6 Using Feature Detection 213
 - 10.6.1 Moving Forward 213
 - 10.6.2 Undetectable Features 214
- 10.7 Summary 214

Part III Real-World Test-Driven Development in JavaScript 217

11. The Observer Pattern 219

- 11.1 The Observer in JavaScript 220
 - 11.1.1 The Observable Library 220
 - 11.1.2 Setting up the Environment 221
- 11.2 Adding Observers 222
 - 11.2.1 The First Test 222
 - 11.2.1.1 Running the Test and Watching It Fail 222
 - 11.2.1.2 Making the Test Pass 223

11.2.2 Refactoring	225
11.3 Checking for Observers	226
11.3.1 The Test	226
11.3.1.1 Making the Test Pass	227
11.3.1.2 Solving Browser Incompatibilities	228
11.3.2 Refactoring	229
11.4 Notifying Observers	230
11.4.1 Ensuring That Observers Are Called	230
11.4.2 Passing Arguments	231
11.5 Error Handling	232
11.5.1 Adding Bogus Observers	232
11.5.2 Misbehaving Observers	233
11.5.3 Documenting Call Order	234
11.6 Observing Arbitrary Objects	235
11.6.1 Making the Constructor Obsolete	236
11.6.2 Replacing the Constructor with an Object	239
11.6.3 Renaming Methods	240
11.7 Observing Arbitrary Events	241
11.7.1 Supporting Events in <code>observe</code>	241
11.7.2 Supporting Events in <code>notify</code>	243
11.8 Summary	246
12. Abstracting Browser Differences: Ajax	247
12.1 Test Driving a Request API	247
12.1.1 Discovering Browser Inconsistencies	248
12.1.2 Development Strategy	248
12.1.3 The Goal	248
12.2 Implementing the Request Interface	249
12.2.1 Project Layout	249
12.2.2 Choosing the Interface Style	250
12.3 Creating an <code>XMLHttpRequest</code> Object	250
12.3.1 The First Test	251
12.3.2 <code>XMLHttpRequest</code> Background	251
12.3.3 Implementing <code>tddjs.ajax.create</code>	253
12.3.4 Stronger Feature Detection	254
12.4 Making Get Requests	255
12.4.1 Requiring a URL	255
12.4.2 Stubbing the <code>XMLHttpRequest</code> Object	257
12.4.2.1 Manual Stubbing	257
12.4.2.2 Automating Stubbing	258
12.4.2.3 Improved Stubbing	261
12.4.2.4 Feature Detection and <code>ajax.create</code>	263

- 12.4.3 Handling State Changes 263
- 12.4.4 Handling the State Changes 265
 - 12.4.4.1 Testing for Success 265
- 12.5 Using the Ajax API 269
 - 12.5.1 The Integration Test 269
 - 12.5.2 Test Results 270
 - 12.5.3 Subtle Trouble Ahead 271
 - 12.5.4 Local Requests 273
 - 12.5.5 Testing Statuses 274
 - 12.5.5.1 Further Status Code Tests 276
- 12.6 Making POST Requests 277
 - 12.6.1 Making Room for Posts 277
 - 12.6.1.1 Extracting `ajax.request` 278
 - 12.6.1.2 Making the Method Configurable 278
 - 12.6.1.3 Updating `ajax.get` 280
 - 12.6.1.4 Introducing `ajax.post` 281
 - 12.6.2 Sending Data 282
 - 12.6.2.1 Encoding Data in `ajax.request` 283
 - 12.6.2.2 Sending Encoded Data 284
 - 12.6.2.3 Sending Data with GET Requests 285
 - 12.6.3 Setting Request Headers 287
- 12.7 Reviewing the Request API 288
- 12.8 Summary 292

13. Streaming Data with Ajax and Comet 293

- 13.1 Polling for Data 294
 - 13.1.1 Project Layout 294
 - 13.1.2 The Poller: `tddjs.ajax.poller` 295
 - 13.1.2.1 Defining the Object 296
 - 13.1.2.2 Start Polling 296
 - 13.1.2.3 Deciding the Stubbing Strategy 298
 - 13.1.2.4 The First Request 299
 - 13.1.2.5 The `complete` Callback 300
 - 13.1.3 Testing Timers 303
 - 13.1.3.1 Scheduling New Requests 304
 - 13.1.3.2 Configurable Intervals 306
 - 13.1.4 Configurable Headers and Callbacks 308
 - 13.1.5 The One-Liner 311
- 13.2 Comet 314
 - 13.2.1 Forever Frames 314
 - 13.2.2 Streaming XMLHttpRequest 315
 - 13.2.3 HTML5 315
- 13.3 Long Polling XMLHttpRequest 315

- 13.3.1 Implementing Long Polling Support 316
 - 13.3.1.1 Stubbing Date 316
 - 13.3.1.2 Testing with Stubbed Dates 317
- 13.3.2 Avoiding Cache Issues 319
- 13.3.3 Feature Tests 320
- 13.4 The Comet Client 321
 - 13.4.1 Messaging Format 321
 - 13.4.2 Introducing `ajax.CometClient` 323
 - 13.4.3 Dispatching Data 323
 - 13.4.3.1 Adding `ajax.CometClient.dispatch` 324
 - 13.4.3.2 Delegating Data 324
 - 13.4.3.3 Improved Error Handling 325
 - 13.4.4 Adding Observers 327
 - 13.4.5 Server Connection 329
 - 13.4.5.1 Separating Concerns 334
 - 13.4.6 Tracking Requests and Received Data 335
 - 13.4.7 Publishing Data 338
 - 13.4.8 Feature Tests 338
- 13.5 Summary 339

14. Server-Side JavaScript with Node.js 341

- 14.1 The Node.js Runtime 341
 - 14.1.1 Setting up the Environment 342
 - 14.1.1.1 Directory Structure 342
 - 14.1.1.2 Testing Framework 343
 - 14.1.2 Starting Point 343
 - 14.1.2.1 The Server 343
 - 14.1.2.2 The Startup Script 344
- 14.2 The Controller 345
 - 14.2.1 CommonJS Modules 345
 - 14.2.2 Defining the Module: The First Test 345
 - 14.2.3 Creating a Controller 346
 - 14.2.4 Adding Messages on POST 347
 - 14.2.4.1 Reading the Request Body 348
 - 14.2.4.2 Extracting the Message 351
 - 14.2.4.3 Malicious Data 354
 - 14.2.5 Responding to Requests 354
 - 14.2.5.1 Status Code 354
 - 14.2.5.2 Closing the Connection 355
 - 14.2.6 Taking the Application for a Spin 356
- 14.3 Domain Model and Storage 358
 - 14.3.1 Creating a Chat Room 358
 - 14.3.2 I/O in Node 358

- 14.3.3 Adding Messages 359
 - 14.3.3.1 Dealing with Bad Data 359
 - 14.3.3.2 Successfully Adding Messages 361
 - 14.3.4 Fetching Messages 363
 - 14.3.4.1 The `getMessagesSince` Method 363
 - 14.3.4.2 Making `addMessage` Asynchronous 365
 - 14.4 Promises 367
 - 14.4.1 Refactoring `addMessage` to Use Promises 367
 - 14.4.1.1 Returning a Promise 368
 - 14.4.1.2 Rejecting the Promise 369
 - 14.4.1.3 Resolving the Promise 370
 - 14.4.2 Consuming Promises 371
 - 14.5 Event Emitters 372
 - 14.5.1 Making `chatRoom` an Event Emitter 372
 - 14.5.2 Waiting for Messages 375
 - 14.6 Returning to the Controller 378
 - 14.6.1 Finishing the `post` Method 378
 - 14.6.2 Streaming Messages with GET 380
 - 14.6.2.1 Filtering Messages with Access Tokens 381
 - 14.6.2.2 The `respond` Method 382
 - 14.6.2.3 Formatting Messages 383
 - 14.6.2.4 Updating the Token 385
 - 14.6.3 Response Headers and Body 386
 - 14.7 Summary 387
- 15. TDD and DOM Manipulation: The Chat Client 389**
- 15.1 Planning the Client 389
 - 15.1.1 Directory Structure 390
 - 15.1.2 Choosing the Approach 390
 - 15.1.2.1 Passive View 391
 - 15.1.2.2 Displaying the Client 391
 - 15.2 The User Form 392
 - 15.2.1 Setting the View 392
 - 15.2.1.1 Setting Up the Test Case 392
 - 15.2.1.2 Adding a Class 393
 - 15.2.1.3 Adding an Event Listener 394
 - 15.2.2 Handling the Submit Event 398
 - 15.2.2.1 Aborting the Default Action 398
 - 15.2.2.2 Embedding HTML in Tests 400
 - 15.2.2.3 Getting the Username 401
 - 15.2.2.4 Notifying Observers of the User 403
 - 15.2.2.5 Removing the Added Class 406
 - 15.2.2.6 Rejecting Empty Usernames 406
 - 15.2.3 Feature Tests 407

- 15.3 Using the Client with the Node.js Backend 408
- 15.4 The Message List 411
 - 15.4.1 Setting the Model 411
 - 15.4.1.1 Defining the Controller and Method 411
 - 15.4.1.2 Subscribing to Messages 412
 - 15.4.2 Setting the View 414
 - 15.4.3 Adding Messages 416
 - 15.4.4 Repeated Messages from Same User 418
 - 15.4.5 Feature Tests 420
 - 15.4.6 Trying it Out 420
- 15.5 The Message Form 422
 - 15.5.1 Setting up the Test 422
 - 15.5.2 Setting the View 422
 - 15.5.2.1 Refactoring: Extracting the Common Parts 423
 - 15.5.2.2 Setting `messageFormController`'s View 424
 - 15.5.3 Publishing Messages 425
 - 15.5.4 Feature Tests 428
- 15.6 The Final Chat Client 429
 - 15.6.1 Finishing Touches 430
 - 15.6.1.1 Styling the Application 430
 - 15.6.1.2 Fixing the Scrolling 431
 - 15.6.1.3 Clearing the Input Field 432
 - 15.6.2 Notes on Deployment 433
- 15.7 Summary 434

Part IV Testing Patterns 437

16. Mocking and Stubbing 439

- 16.1 An Overview of Test Doubles 439
 - 16.1.1 Stunt Doubles 440
 - 16.1.2 Fake Object 440
 - 16.1.3 Dummy Object 441
- 16.2 Test Verification 441
 - 16.2.1 State Verification 442
 - 16.2.2 Behavior Verification 442
 - 16.2.3 Implications of Verification Strategy 443
- 16.3 Stubs 443
 - 16.3.1 Stubbing to Avoid Inconvenient Interfaces 444
 - 16.3.2 Stubbing to Force Certain Code Paths 444
 - 16.3.3 Stubbing to Cause Trouble 445
- 16.4 Test Spies 445
 - 16.4.1 Testing Indirect Inputs 446
 - 16.4.2 Inspecting Details about a Call 446
- 16.5 Using a Stub Library 447

- 16.5.1 Creating a Stub Function 448
- 16.5.2 Stubbing a Method 448
- 16.5.3 Built-in Behavior Verification 451
- 16.5.4 Stubbing and Node.js 452
- 16.6 Mocks 453
 - 16.6.1 Restoring Mocked Methods 453
 - 16.6.2 Anonymous Mocks 454
 - 16.6.3 Multiple Expectations 455
 - 16.6.4 Expectations on the `this` Value 456
- 16.7 Mocks or Stubs? 457
- 16.8 Summary 458

17. Writing Good Unit Tests 461

- 17.1 Improving Readability 462
 - 17.1.1 Name Tests Clearly to Reveal Intent 462
 - 17.1.1.1 Focus on Scannability 462
 - 17.1.1.2 Breaking Free of Technical Limitations 463
 - 17.1.2 Structure Tests in Setup, Exercise, and Verify Blocks 464
 - 17.1.3 Use Higher-Level Abstractions to Keep Tests Simple 465
 - 17.1.3.1 Custom Assertions: Behavior Verification 465
 - 17.1.3.2 Domain Specific Test Helpers 466
 - 17.1.4 Reduce Duplication, Not Clarity 467
- 17.2 Tests as Behavior Specification 468
 - 17.2.1 Test One Behavior at a Time 468
 - 17.2.2 Test Each Behavior Only Once 469
 - 17.2.3 Isolate Behavior in Tests 470
 - 17.2.3.1 Isolation by Mocking and Stubbing 470
 - 17.2.3.2 Risks Introduced by Mocks and Stubs 471
 - 17.2.3.3 Isolation by Trust 472
- 17.3 Fighting Bugs in Tests 473
 - 17.3.1 Run Tests Before Passing Them 473
 - 17.3.2 Write Tests First 473
 - 17.3.3 Heckle and Break Code 474
 - 17.3.4 Use JsLint 474
- 17.4 Summary 475

Bibliography 477

Index 479

Preface

Author’s Vision for the Book

Over the recent years, JavaScript has grown up. Long gone are the glory days of “DHTML”; we are now in the age of “Ajax,” possibly even “HTML5.” Over the past years JavaScript gained some killer applications; it gained robust libraries to aid developers in cross-browser scripting; and it gained a host of tools such as debuggers, profilers, and unit testing frameworks. The community has worked tirelessly to bring in the tools they know and love from other languages to help give JavaScript a “real” development environment in which they can use the workflows and knowledge gained from working in other environments and focus on building quality applications.

Still, the JavaScript community at large is not particularly focused on automated testing, and test-driven development is still rare among JavaScript developers—in spite of working in the language with perhaps the widest range of target platforms. For a long time this may have been a result of lacking tool support, but new unit testing frameworks are popping up all the time, offering a myriad of ways to test your code in a manner that suits you. Even so, most web application developers skimp on testing their JavaScript. I rarely meet a web developer who has the kind of confidence to rip core functionality right out of his application and rearrange it, that a strong test suite gives you. This confidence allows you to worry less about breaking your application, and focus more on implementing new features.

With this book I hope to show you that unit testing and test-driven development in JavaScript have come a long way, and that embracing them will help you write better code and become a more productive programmer.

What This Book is About

This book is about programming JavaScript for the real world, using the techniques and workflow suggested by Test-Driven Development. It is about gaining confidence in your code through test coverage, and gaining the ability to fearlessly refactor and organically evolve your code base. It is about writing modular and testable code. It is about writing JavaScript that works in a wide variety of environments and that doesn't get in your user's way.

How This Book is Organized

This book has four parts. They may be read in any order you're comfortable with. Part II introduces a few utilities that are used throughout the book, but their usage should be clear enough, allowing you to skip that part if you already have a solid understanding of programming JavaScript, including topics such as unobtrusive JavaScript and feature detection.

Part I: Test-Driven Development

In the first part I'll introduce you to the concept of automated tests and test-driven development. We'll start by looking at what a unit test is, what it does, and what it's good for. Then we'll build our workflow around them as I introduce the test-driven development process. To round the topic off I'll show you a few available unit testing frameworks for JavaScript, discuss their pros and cons, and take a closer look at the one we'll be using the most throughout the book.

Part II: JavaScript for Programmers

In Part II we're going to get a deeper look at programming in JavaScript. This part is by no means a complete introduction to the JavaScript language. You should already either have some experience with JavaScript—perhaps by working with libraries like jQuery, Prototype, or the like—or experience from other programming languages. If you're an experienced programmer with no prior experience with JavaScript, this part should help you understand where JavaScript differs from other languages, especially less dynamic ones, and give you the foundation you'll need for the real-world scenarios in Part III.

If you're already well-versed in advanced JavaScript concepts such as closures, prototypal inheritance, the dynamic nature of `this`, and feature detection, you may want to skim this part for a reminder, or you may want to skip directly to Part III.

While working through some of JavaScript’s finer points, I’ll use unit tests to show you how the language behaves, and we’ll take the opportunity to let tests drive us through the implementation of some helper utilities, which we’ll use throughout Part III.

Part III: Real-World Test-Driven Development in JavaScript

In this part we’ll tackle a series of small projects in varying environments. We’ll see how to develop a small general purpose JavaScript API, develop a DOM dependent widget, abstract browser differences, implement a server-side JavaScript application, and more—all using test-driven development. This part focuses on how test-driven development can help in building cleaner APIs, better modularized code and more robust software.

Each project introduces new test-related concepts, and shows them in practice by implementing a fully functional, yet limited piece of code. Throughout this part we will, among other things, learn how to test code that depends on browser APIs, timers, event handlers, DOM manipulation, and asynchronous server requests (i.e., “Ajax”). We will also get to practice techniques such as stubbing, refactoring, and using design patterns to solve problems in elegant ways.

Throughout each chapter in this part, ideas on how to extend the functionality developed are offered, giving you the ability to practice by improving the code on your own. Extended solutions are available from the book’s website.¹

I’ve taken great care throughout these projects to produce runnable code that actually does things. The end result of the five chapters in Part III is a fully functional instant messaging chat client and server, written exclusively using test-driven development, in nothing but JavaScript.

Part IV: Testing Patterns

The final part of the book reviews some of the techniques used throughout Part III from a wider angle. Test doubles, such as mocks and stubs, are investigated in closer detail along with different forms of test verification. Finally, we review some guidelines to help you write good unit tests.

Conventions Used in This Book

JavaScript is the name of the language originally designed by Brendan Eich for Netscape in 1995. Since then, a number of alternative implementations have

1. <http://tddjs.com>

surfaced, and the language has been standardized by ECMA International as ECMA-262, also known as ECMAScript. Although the alternative implementations have their own names, such as Microsoft's JScript, they are generally collectively referred to as "JavaScript," and I will use JavaScript in this sense as well.

Throughout the text, monospaced font is used to refer to objects, functions, and small snippets of code.

Who Should Read This Book

This book is for programmers—especially those who write, or are interested in writing JavaScript. Whether you're a Ruby developer focusing primarily on Ruby on Rails; a Java or .Net developer working with web applications; a frontend web developer whose primary tools are JavaScript, CSS, and HTML; or even a backend developer with limited JavaScript experience, I hope and think you will find this book useful.

The book is intended for web application developers who need a firmer grasp of the finer details of the JavaScript language, as well as better understanding on how to boost their productivity and confidence while writing maintainable applications with fewer defects.

Skills Required For This Book

The reader is not required to have any previous knowledge of unit testing or test-driven development. Automated tests are present through the whole book, and reading should provide you with a strong understanding of how to successfully use them.

Equally, the reader is not required to be a JavaScript expert, or even intermediate. My hope is that the book will be useful to programmers with very limited JavaScript experience and savvy JavaScripters alike. You are required, however, to possess some programming skills, meaning that in order to fully enjoy this book you should have experience programming in some language, and be familiar with web application development. This book is not an introductory text in any of the basic programming related topics, web application-specific topics included.

The second part of the book, which focuses on the JavaScript language, focuses solely on the qualities of JavaScript that set it apart from the pack, and as such cannot be expected to be a complete introduction to the language. It is expected that you will be able to pick up syntax and concepts not covered in this part through examples using them.

In particular, Part II focuses on JavaScript's functions and closures; JavaScript's object model, including prototypal inheritance; and models for code-reuse. Additionally, we will go through related programming practices such as unobtrusive JavaScript and feature detection, both required topics to understand for anyone targeting the general web.

About the Book's Website

The book has an accompanying website, <http://tddjs.com>. At this location you will find all the code listings from the book, both as zip archives and full Git repositories, which allow you to navigate the history and see how the code evolves. The Git repositories are especially useful for the Part III sample projects, where a great deal of refactoring is involved. Navigating the history of the Git repositories allows you to see each step even when they simply change existing code.

You can also find my personal website at <http://cjohansen.no> in which you will find additional articles, contact information, and so on. If you have any feedback regarding the book, I would love to hear back from you.

This page intentionally left blank

Acknowledgments

Quite a few people have made this book possible. First of all I would like to commend Trina MacDonald, my editor at Addison-Wesley, for being the one who made all of this possible. Without her, there would be no book, and I deeply appreciate her initiative as well as her ongoing help and motivation while I stumblingly worked my way through my first book.

I would also like to extend my gratitude toward the rest of the team working with me on this book; Songlin Qiu for making sure the text is comprehensible and consistent, and for keeping sane while reviewing a constantly changing manuscript. Her insights and suggestions have truly made the book better than I could ever manage on my own. The same can be said for my technical reviewers, Andrea Giammarchi, Jacob Seidelin, and Joshua Gross. Their impressive attention to detail, thoughtful feedback, and will to challenge me have helped clarify code, remove errors, and generally raise the quality of both code samples and surrounding prose, as well as the structure of the book. Last, but not least, Olivia Basego helped me cope with the administrative side of working with a publisher like Addison-Wesley and some challenges related to living in Norway while writing for an American publisher.

Closer to home, my employers and coworkers at Shortcut AS deserve an honorable mention. Their flexibility in allowing me to occasionally take time off to write and their genuine interest in the book at large have been very motivating and key to finishing the manuscript in time. In particular I would like to thank Marius Mårnes Mathiesen and August Lilleaas for frequent discussions of a truly inspiring and insightful nature, as well as feedback on early drafts.

Last, but definitely not least; Frøydis and Kristin, friends and bandmates who have given me space to complete this project and stayed patient while I've been

zombie-like tired after long nights of writing, unavailable for various occasions, and generally chained to the kitchen table for months (that's right, I wrote this book in the kitchen)—thank you for your support.

Finally I would like to extend my appreciation for the open source community at large. Without it, this book would not be what it is. Open source is what ultimately got me into writing in the first place. It kept my blog alive; it crossed my path with my editor's; and now it is responsible for the book you're holding in your hands. Most of the code throughout the book would not have been possible were it not for people tirelessly putting out top-notch code for anyone to freely peruse, modify, and use.

All software involved in my part of the production of this book are open source as well. The book was written entirely in Emacs, using the document preparation system LaTeX. A host of minor open source tools have been involved in the workflow, many of which are native citizens in my operating system of choice—GNU Linux.

When the book hits the streets, it will have brought with it at least one new open source project, and I hope I will contribute many more in the years to come.

About the Author

Christian Johansen lives in Oslo, Norway, where he currently works for Shortcut AS, a software company focusing on open source technology, web applications, and mobile applications. Originally a student in informatics, mathematics, and digital signal processing, Christian has spent his professional career specializing in web applications and frontend technologies such as JavaScript, CSS, and HTML, technologies he has been passionate about since around the time the HTML 4.01 spec was finalized.

As a consultant, Christian has worked with many high profile companies in Norway, including leading companies within the finance and telecom sector, where he has worked on small and big web applications ranging from the average CMS-backed corporate website via e-commerce to self service applications.

In later years Christian has been an avid blogger. Derived from the same desire to share and contribute to the community that gave him so much for free, Christian has involved himself in and contributed to quite a few open source projects.

After working on several projects with less than trivial amounts of JavaScript, Christian has felt the pain of developing “the cowboy style.” In an attempt at improving code quality, confidence, and the ability to modify and maintain code with greater ease, he has spent a great deal of his time both at work and in his spare time over the last few years investigating unit testing and test-driven development in JavaScript. Being a sworn TDD-er while developing in traditional server-side languages, the cowboy style JavaScript approach wasn’t cutting it anymore. The culmination of this passion is the book you now hold in your hands.

This page intentionally left blank

Tools of the Trade

In Chapter 1, *Automated Testing*, we developed a very simple `testCase` function, capable of running basic unit tests with test case setup and teardown methods. Although rolling our own test framework is a great exercise, there are many frameworks already available for JavaScript and this chapter explores a few of them.

In this chapter we will take a look at “the tools of the trade”—essential and useful tools to support a test-driven workflow. The most important tool is of course the testing framework, and after an overview of available frameworks, we will spend some time setting up and running `JsTestDriver`, the testing framework used for most of this book’s example code. In addition to a testing framework, this chapter looks at tools such as coverage reports and continuous integration.

3.1 xUnit Test Frameworks

In Chapter 1, *Automated Testing*, we coined *xUnit* as the term used to describe testing frameworks that lean on the design of Java’s `JUnit` and Smalltalk’s `SUnit`, originally designed by Kent Beck. The *xUnit* family of test frameworks is still the most prevalent way of writing automated tests for code, even though the past few years have seen a rise in usage for so-called *behavior-driven development* (or *BDD*) testing frameworks.

3.1.1 Behavior-Driven Development

Behavior-driven development, or BDD, is closely related to TDD. As discussed in Chapter 2, *The Test-Driven Development Process*, TDD is *not* about testing, but rather about design and process. However, due to the terminology used to describe the process, a lot of developers never evolve beyond the point where they simply write unit tests to verify their code, and thus never experience many of the advantages associated with using tests as a design tool. BDD seeks to ease this realization by focusing on an improved vocabulary. In fact, vocabulary is perhaps the most important aspect of BDD, because it also tries to normalize the vocabulary used by programmers, business developers, testers, and others involved in the development of a system when discussing problems, requirements, and solutions.

Another “double D” is Acceptance Test-Driven Development. In acceptance TDD, development starts by writing automated tests for high level features, based on acceptance tests defined in conjunction with the client. The goal is to pass the acceptance tests. To get there, we can identify smaller parts and proceed with “regular” TDD. In BDD this process is usually centered around *user stories*, which describe interaction with the system using a vocabulary familiar to everyone involved in the project. BDD frameworks such as Cucumber allow for user stories to be used as executable tests, meaning that acceptance tests can be written together with the client, increasing the chance of delivering the product the client had originally envisioned.

3.1.2 Continuous Integration

Continuous integration is the practice of integrating code from all developers on a regular basis, usually every time a developer pushes code to a remote version control repository. The continuous integration server typically builds all the sources and then runs tests for them. This process ensures that even when developers work on isolated units of features, the integrated whole is considered every time code is committed to the upstream repository. JavaScript does not need compiling, but running the entire test suite for the application on a regular basis can help catch errors early.

Continuous integration for JavaScript can solve tasks that are impractical for developers to perform regularly. Running the entire test suite in a wide array of browser and platform combinations is one such task. Developers working with TDD can focus their attention on a small representative selection of browsers, while the continuous integration server can test much wider, alerting the team of errors by email or RSS.

Additionally, it is common practice for JavaScript to be served minified—i.e., with unneeded white-space and comments stripped out, and optionally local identifiers munged to occupy fewer bytes—to preserve bytes over the wire. Both minifying code too aggressively or merging files incorrectly can introduce bugs. A continuous integration server can help out with these kinds of problems by running all tests on the full source as well as building concatenated and minified release files and re-running the test suite for them.

3.1.3 Asynchronous Tests

Due to the asynchronous nature of many JavaScript programming tasks such as working with `XMLHttpRequest`, animations and other deferred actions (i.e., any code using `setTimeout` or `setInterval`), and the fact that browsers do not offer a `sleep` function (because it would freeze the user interface), many testing frameworks provide a means to execute asynchronous tests. Whether or not asynchronous *unit* tests is a good idea is up for discussion. Chapter 12, *Abstracting Browser Differences: Ajax*, offers a more thorough discussion on the subject as well as an example.

3.1.4 Features of xUnit Test Frameworks

Chapter 1, *Automated Testing*, already introduced us to the basic features of the xUnit test frameworks: Given a set of test methods, the framework provides a test runner that can run them and report back the results. To ease the creation of shared test fixtures, test cases can employ the `setUp` and `tearDown` functions, which are run before and after (respectively) each individual test in a test case. Additionally, the test framework provides a set of assertions that can be used to verify the state of the system being tested. So far we have only used the `assert` method which accepts any value and throws an exception when the value is falsy. Most frameworks provide more assertions that help make tests more expressive. Perhaps the most common assertion is a version of `assertEqual`, used to compare actual results against expected values.

When evaluating test frameworks, we should assess the framework's test runner, its assertions, and its dependencies.

3.1.4.1 The Test Runner

The test runner is the most important part of the testing framework because it basically dictates the workflow. For example, most unit testing frameworks available for JavaScript today use an in-browser test runner. This means that tests must run inside a browser by loading an HTML file (often referred to as an HTML

fixture) that itself loads the libraries to test, along with the unit tests and the testing framework. Other types of test runners can run in other environments, e.g., using Mozilla's Rhino implementation to run tests on the command line. What kind of test runner is suitable to test a specific application depends on whether it is a client-side application, server-side, or maybe even a browser plugin (an example of which would be FireUnit, a unit testing framework that uses Firebug and is suitable for developing Firefox plugins).

A related concern is the test report. Clear fail/success status is vital to the test-driven development process, and clear feedback with details when tests fail or have errors is needed to easily handle them as they occur. Ideally, the test runner should produce test results that are easily integrated with continuous integration software.

Additionally, some sort of plugin architecture for the test runner can enable us to gather metrics from testing, or otherwise allow us to extend the runner to improve the workflow. An example of such a plugin is the test coverage report. A coverage report shows how well the test suite covers the system by measuring how many lines in production code are executed by tests. Note that 100% coverage does not imply that every thinkable test is written, but rather that the test suite executes each and every line of production code. Even with 100% coverage, certain sets of input can still break the code—it cannot guarantee the absence of, e.g., missing error handling. Coverage reports are useful to find code that is not being exercised by tests.

3.1.5 Assertions

A rich set of assertions can really boost the expressiveness of tests. Given that a good unit test clearly states its intent, this is a massive boon. It's a lot easier to spot what a test is targeting if it compares two values with `assertEqual(expected, actual)` rather than with `assert(expected == actual)`. Although `assert` is all we really need to get the job done, more specific assertions make test code easier to read, easier to maintain, and easier to debug.

Assertions is one aspect where an exact port of the xUnit framework design from, e.g., Java leaves a little to be desired. To achieve good expressiveness in tests, it's helpful to have assertions tailored to specific language features, for instance, having assertions to handle JavaScripts special values such as `undefined`, `NaN` and `infinity`. Many other assertions can be provided to better support testing JavaScript, not just some arbitrary programming language. Luckily, specific assertions like those mentioned are easy to write piggybacking a general purpose `assert` (or, as is common, a `fail` method that can be called when the assertion does not hold).

3.1.6 Dependencies

Ideally, a testing framework should have as few dependencies as possible. More dependencies increase the chance of the mechanics of the framework not working in some browser (typically older ones). The worst kind of dependency for a testing framework is an obtrusive library that tampers with the global scope. The original version of JsUnitTest, the testing framework built for and used by the Prototype.js library, depended on Prototype.js itself, which not only adds a number of global properties but also augments a host of global constructors and objects. In practice, using it to test code that was not developed with Prototype.js would prove a futile exercise for two reasons:

- Too easy to accidentally rely on Prototype.js through the testing framework (yielding green tests for code that would fail in production, where Prototype.js would not be available)
- Too high a risk for collisions in the global scope (e.g., the MooTools library adds many of the same global properties)

3.2 In-Browser Test Frameworks

The original JavaScript port of the JUnit framework was JsUnit, first released in 2001. Not surprisingly, it has in many ways set the standard for a lot of testing frameworks following it. JsUnit runs tests in a browser: The test runner prompts for the URL to a test file to execute. The test file may be an HTML test suite which links to several test cases to execute. The tests are then run in sandboxed frames, and a green progress bar is displayed while tests are running. Obviously, the bar turns red whenever a test fails. JsUnit still sees the occasional update, but it has not been significantly updated for a long time, and it's starting to lag behind. JsUnit has served many developers well, including myself, but there are more mature and up-to-date alternatives available today.

Common for the in-browser testing frameworks is how they require an HTML fixture file to load the files to test, the testing library (usually a JavaScript and a CSS file), as well as the tests to run. Usually, the fixture can be simply copy-pasted for each new test case. The HTML fixture also serves the purpose of hosting dummy markup needed for the unit tests. If tests don't require such markup, we can lessen the burden of keeping a separate HTML file for each test case by writing a script that scans the URL for parameters naming library and test files to load, and then load them dynamically. This way we can run several test cases from the same HTML fixture simply by modifying the URL query string. The fixture could of course also be generated by a server-side application, but be careful down this route. I advise you

to keep things simple—complicated test runners greatly decreases the likelihood of developers running tests.

3.2.1 YUI Test

Most of the major JavaScript libraries available today have their own unit testing framework. YUI from Yahoo! is no exception. YUI Test 3 can be safely used to test arbitrary JavaScript code (i.e., it has no obtrusive dependencies). YUI Test is, in its own words, “not a direct port from any specific xUnit framework,” but it “does derive some characteristics from nUnit and JUnit,” with nUnit being the .NET interpretation of the xUnit family of frameworks, written in C#. YUI Test is a mature testing framework with a rich feature set. It supports a rich set of assertions, test suites, a *mocking* library (as of YUI 3), and asynchronous tests.

3.2.1.1 Setup

Setup is very easy thanks to YUI’s loader utility. To get quickly started, we can link directly to the YUI seed file on the YUI server, and use `YUI.use` to fetch the necessary dependencies. We will revisit the `strftime` example from Chapter 1, *Automated Testing*, in order to compare YUI Test to the `testCase` function introduced in that chapter. Listing 3.1 shows the HTML fixture file, which can be saved in, e.g., `strftime-yui-test.html`.

Listing 3.1 YUI Test HTML fixture file

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Testing Date.prototype.strftime with YUI</title>
    <meta http-equiv="content-type"
          content="text/html; charset=UTF-8">
  </head>
  <body class="yui-skin-sam">
    <div id="yui-main"><div id="testReport"></div></div>
    <script type="text/javascript"
      src="http://yui.yahooapis.com/3.0.0/build/yui/yui-min.js">
    </script>
    <script type="text/javascript" src="strftime.js">
    </script>
    <script type="text/javascript" src="strftime_test.js">
    </script>
  </body>
</html>
```

The `strftime.js` file contains the `Date.prototype.strftime` implementation presented in Listing 1.2 in Chapter 1, *Automated Testing*. Listing 3.2 shows the test script, save it in `strftime-test.js`.

Listing 3.2 `Date.prototype.strftime` YUI test case

```
YUI({
  combine: true,
  timeout: 10000
}).use("node", "console", "test", function (Y) {
  var assert = Y.Assert;

  var strftimeTestCase = new Y.Test.Case({
    // test case name - if not provided, one is generated
    name: "Date.prototype.strftime Tests",

    setUp: function () {
      this.date = new Date(2009, 9, 2, 22, 14, 45);
    },

    tearDown: function () {
      delete this.date;
    },

    "test %Y should return full year": function () {
      var year = Date.formats.Y(this.date);

      assert.isNumber(year);
      assert.areEqual(2009, year);
    },

    "test %m should return month": function () {
      var month = Date.formats.m(this.date);

      assert.isString(month);
      assert.areEqual("10", month);
    },

    "test %d should return date": function () {
      assert.areEqual("02", Date.formats.d(this.date));
    },

    "test %y should return year as two digits": function () {
      assert.areEqual("09", Date.formats.y(this.date));
    },
  });
});
```

```
"test %F should act as %Y-%m-%d": function () {
    assert.areEqual("2009-10-02", this.date.strftime("%F"));
}
});

//create the console
var r = new Y.Console({
    newestOnTop : false,
    style: 'block'
});

r.render("#testReport");
Y.Test.Runner.add(strftimeTestCase);
Y.Test.Runner.run();
});
```

When using YUI Test for production code, the required sources should be downloaded locally. Although the loader is a convenient way to get started, relying on an internet connection to run tests is bad practice because it means we cannot run tests while offline.

3.2.1.2 Running Tests

Running tests with YUI Test is as simple as loading up the HTML fixture in a browser (preferably several browsers) and watching the output in the console, as seen in Figure 3.1.

3.2.2 Other In-Browser Testing Frameworks

When choosing an in-browser testing framework, options are vast. YUI Test is among the most popular choices along with JsUnit and QUnit. As mentioned, JsUnit is long overdue for an upgrade, and I suggest you not start new projects with it at this point. QUnit is the testing framework developed and used by the jQuery team. Like YUI Test it is an in-browser test framework, but follows the traditional xUnit design less rigidly. The Dojo and Prototype.js libraries both have their test frameworks as well.

One might get the impression that there are almost as many testing frameworks out there as there are developers unit testing their scripts—there is no defacto standard way to test JavaScript. In fact, this is true for most programming tasks that are not directly related to browser scripting, because JavaScript has no general purpose standard library. CommonJS is an initiative to rectify this situation, originally motivated to standardize server-side JavaScript. CommonJS also includes a

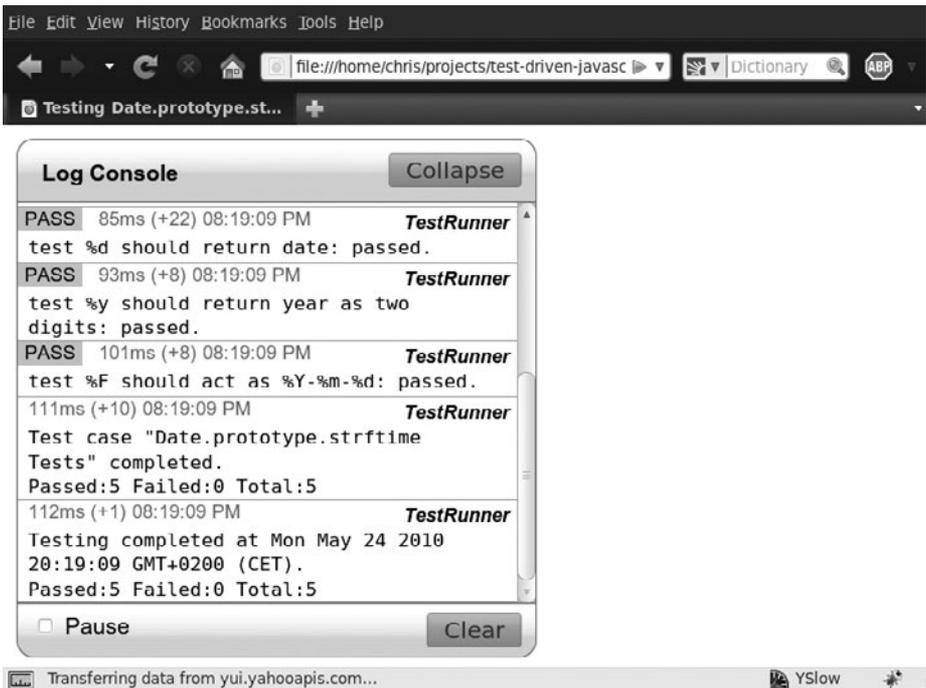


Figure 3.1 Running tests with YUI Test.

unit testing spec, which we will look into when testing a `Node.js` application in Chapter 14, *Server-Side JavaScript with Node.js*.

3.3 Headless Testing Frameworks

In-browser testing frameworks are unfit to support a test-driven development process where we need to run tests frequently and integrated into the workflow. An alternative to these frameworks is headless testing frameworks. These typically run from the command line, and can be interacted with in the same way testing frameworks for any other server-side programming language can.

There are a few solutions available for running headless JavaScript unit tests, most originating from either the Java or Ruby worlds. Both the Java and Ruby communities have strong testing cultures, and testing only half the code base (the server-side part) can only make sense for so long, probably explaining why it is these two communities in particular that have stood out in the area of headless testing solutions for JavaScript.

3.3.1 Crosscheck

Crosscheck is one of the early headless testing frameworks. It provides a Java backed emulation of Internet Explorer 6 and Firefox versions 1.0 and 1.5. Needless to say, Crosscheck is lagging behind, and its choice of browsers are unlikely to help develop applications for 2010. Crosscheck offers JavaScript unit tests much like that of YUI Test, the difference being that they can be run on the command line with the Crosscheck jar file rather than in a browser.

3.3.2 Rhino and env.js

`env.js` is a library originally developed by John Resig, creator of the jQuery JavaScript framework. It offers an implementation of the browser (i.e., BOM) and DOM APIs on top of Rhino, Mozilla's Java implementation of JavaScript. Using the `env.js` library together with Rhino means we can load and run in-browser tests on the command line.

3.3.3 The Issue with Headless Test Runners

Although the idea of running tests on the command line is exciting, I fail to recognize the power of running tests in an environment where production code will never run. Not only are the browser environment and DOM emulations, but the JavaScript engine (usually Rhino) is an altogether different one as well.

Relying on a testing framework that simply emulates the browser is bad for a few reasons. For one, it means tests can only be run in browsers that are emulated by the testing framework, or, as is the case for solutions using Rhino and `env.js`, in an alternate browser and DOM implementation altogether. Limiting the available testing targets is not an ideal feature of a testing framework and is unlikely to help write cross-browser JavaScript. Second, an emulation will never match whatever it is emulating perfectly. Microsoft probably proved this best by providing an Internet Explorer 7 emulation mode in IE8, which is in fact not an exact match of IE7. Luckily, we can get the best from both worlds, as we will see next, in Section 3.4, *One Test Runner to Rule Them All*.

3.4 One Test Runner to Rule Them All

The problem with in-browser testing frameworks is that they can be cumbersome to work with, especially in a test-driven development setting where we need to run tests continuously and integrated into the workflow. Additionally, testing on a wide array of platform/browser combinations can entail quite a bit of manual work. Headless

frameworks are easier to work with, but fail at testing in the actual environment the code will be running in, reducing their usefulness as testing tools. A fairly new player on the field of xUnit testing frameworks is `JsTestDriver`, originating from Google. In contrast to the traditional frameworks, `JsTestDriver` is first and foremost a test runner, and a clever one at that. `JsTestDriver` solves the aforementioned problems by making it easy both to run tests and to test widely in real browsers.

3.4.1 How `JsTestDriver` Works

`JsTestDriver` uses a small server to run tests. Browsers are captured by the test runner and tests are scheduled by issuing a request to the server. As each browser runs the tests, results are sent back to the client and presented to the developer. This means that as browsers are idly awaiting tests, we can schedule runs from either the command line, the IDE, or wherever we may feel most comfortable running them from. This approach has numerous advantages:

- Tests can be run in browsers without requiring manual interaction with the browser.
- Tests can be run in browsers on multiple machines, including mobile devices, allowing for arbitrary complex testing grids.
- Tests run **fast**, due to the fact that results need not be added to the DOM and rendered, they can be run in any number of browsers simultaneously, and the browser doesn't need to reload scripts that haven't changed since the tests were last run.
- Tests can use the full DOM because no portion of the document is reserved for the test runner to display results.
- No need for an HTML fixture, simply provide one or more scripts and test scripts, an empty document is created on the fly by the test runner.

`JsTestDriver` tests are **fast**. The test runner can run complex test suites of several hundred tests in under a single second. Because tests are run simultaneously, tests will still run in about a second even when testing 15 browsers at the same time. Granted, some time is spent communicating with the server and optionally refreshing the browser cache, but a full run still completes in a matter of a few seconds. Single test case runs usually complete in the blink of an eye.

As if faster tests, simpler setup, and full DOM flexibility weren't enough, `JsTestDriver` also offers a plugin that calculates test coverage, XML test report output compatible with JUnit's reports, meaning we can immediately use existing continuous

integration servers, and it can use alternative assertion frameworks. Through plugins, any other JavaScript testing framework can take advantage of the `JsTestDriver` test runner, and at the time of writing, adapters for QUnit and YUI Test already exist. This means tests can be written using YUI Test's assertions and syntax, but run using `JsTestDriver`.

3.4.2 `JsTestDriver` Disadvantages

At the time of writing, `JsTestDriver` does not support any form of asynchronous testing. As we will see in Chapter 12, *Abstracting Browser Differences: Ajax*, this isn't necessarily a problem from a unit testing perspective, but it may limit the options for integration tests, in which we want to fake as little as possible. It is possible that asynchronous test support will be added to future versions of `JsTestDriver`.

Another disadvantage of `JsTestDriver` is that the JavaScript required to run tests is slightly more advanced, and may cause a problem in old browsers. For instance, by design, a browser that is to run `JsTestDriver` needs to support the `XMLHttpRequest` object or similar (i.e., Internet Explorer's corresponding `ActiveXObject`) in order to communicate with the server. This means that browsers that don't support this object (older browsers, Internet Explorer before version 7 with ActiveX disabled) cannot be tested with the `JsTestDriver` test runner. This problem can be effectively circumvented, however, by using YUI Test to write tests, leaving the option of running them manually with the default test runner in any uncooperative browser.

3.4.3 Setup

Installing and setting up `JsTestDriver` is slightly more involved than the average in-browser testing framework; still, it will only take a few minutes. Also, the setup is only required once. Any projects started after the fact are dirt simple to get running. `JsTestDriver` requires Java to run both the server component and start test runs. I won't give instructions on installing Java here, but most systems have Java installed already. You can check if Java is installed by opening a shell and issue the `java -version` command. If you don't have Java installed, you will find instructions on java.com.

3.4.3.1 Download the Jar File

Once Java is set up, download the most recent `JsTestDriver` jar file from <http://code.google.com/p/js-test-driver/downloads/list>. All the examples in this book use version 1.2.1, be sure to use that version when following along with the

examples. The jar file can be placed anywhere on the system, I suggest `~/bin`. To make it easier to run, set up an environment variable to point to this directory, as shown in Listing 3.3.

Listing 3.3 Setting the `$JSTESTDRIVER_HOME` environment variable

```
export JSTESTDRIVER_HOME=~/bin
```

Set the environment variable in a login script, such as `.bashrc` or `.zshrc` (depends on the shell—most systems use Bash, i.e., `~/ .bashrc`, by default).

3.4.3.2 Windows Users

Windows users can set an environment variable in the `cmd` command line by issuing the `set JSTESTDRIVER_HOME=C:\bin` command. To set it permanently, right-click *My Computer* (*Computer* in Windows 7) and select *Properties*. In the *System window*, select *Advanced system properties*, then the *Advanced tab*, and then click the *Environment Variables . . .* button. Decide if you need to set the environment variable for yourself only or for all users. Click *New*, enter the name (`JSTESTDRIVER_HOME`) in the top box, and then the path where you saved the jar file in the bottom one.

3.4.3.3 Start the Server

To run tests through `JsTestDriver`, we need a running server to capture browsers with. The server can run anywhere reachable from your machine—locally, on a machine on the local network, or a public facing machine. Beware that running the server on a public machine will make it available to anyone unless the machine restricts access by IP address or similar. To get started, I recommend running the service locally; this way you can test while being offline as well. Open a shell and issue the command in either Listing 3.4 or Listing 3.5 (current directory is not important for this command).

Listing 3.4 Starting the `JsTestDriver` server on Linux and OSX

```
java -jar $JSTESTDRIVER_HOME/JsTestDriver-1.2.1.jar --port  
4224
```

Listing 3.5 Starting the `JsTestDriver` server on Windows

```
java -jar %JSTESTDRIVER_HOME%\JsTestDriver-1.2.1.jar --port  
4224
```

Port 4224 is the defacto standard JsTestDriver port, but it is arbitrarily picked and you can run it on any port you want. Once the server is running, the shell running it must stay open for as long as you need it.

3.4.3.4 Capturing Browsers

Open any browser and point it to `http://localhost:4224` (make sure you change the port number if you used another port when starting the server). The resulting page will display two links: *Capture browser* and *Capture in strict mode*. JsTestDriver runs tests inside an HTML 4.01 document, and the two links allow us to decide if we want to run tests with a transitional or strict doctype. Click the appropriate link, and leave the browser open. Repeat in as many browsers as desired. You can even try hooking up your phone or browsers on other platforms using virtual instances.

3.4.3.5 Running Tests

Tests can be run from the command line, providing feedback in much the same way a unit testing framework for any server-side language would. As tests are run, a dot will appear for every passing test, an F for a failing test, and an E for a test with errors. An error is any test error that is not a failing assertion, i.e., an unexpected exception. To run the tests, we need a small configuration file that tells JsTestDriver which source and test files to load (and in what order), and which server to run tests against. The configuration file, `jsTestDriver.conf` by default, uses YAML syntax, and at its simplest, it loads every source file and every test file, and runs tests at `http://localhost:4224`, as seen in Listing 3.6.

Listing 3.6 A barebone `jsTestDriver.conf` file

```
server: http://localhost:4224

load:
- src/*.js
- test/*.js
```

Load paths are relative to the location of the configuration file. When it's required to load certain files before others, we can specify them first and still use the `*.js` notation, JsTestDriver will only load each file once, even when it is referenced more than once. Listing 3.7 shows an example where `src/mylib.js` always need to load first.

Listing 3.7 Making sure certain files load first

```
server: http://localhost:4224
```

```
load:
```

- src/mylib.js
 - src/*.js
 - test/*.js
-

In order to test the configuration we need a sample project. We will revisit the `strftime` example once again, so start by copying the `strftime.js` file into the `src` directory. Then add the test case from Listing 3.8 in `test/strftime-test.js`.

Listing 3.8 `Date.prototype.strftime` test with `JsTestDriver`

```
TestCase("strftimeTest", {
  setUp: function () {
    this.date = new Date(2009, 9, 2, 22, 14, 45);
  },

  tearDown: function () {
    delete this.date;
  },

  "test %Y should return full year": function () {
    var year = Date.formats.Y(this.date);

    assertNumber(year);
    assertEquals(2009, year);
  },

  "test %m should return month": function () {
    var month = Date.formats.m(this.date);

    assertString(month);
    assertEquals("10", month);
  },

  "test %d should return date": function () {
    assertEquals("02", Date.formats.d(this.date));
  },

  "test %y should return year as two digits": function () {
    assertEquals("09", Date.formats.y(this.date));
  },
});
```

```

    "test %F should act as %Y-%m-%d": function () {
      assertEquals("2009-10-02", this.date.strftime("%F"));
    }
  });

```

The test methods are almost syntactically identical to the YUI Test example, but note how this test case has less scaffolding code to support the test runner. Now create the configuration file as shown in Listing 3.9.

Listing 3.9 JsTestDriver configuration

```

server: http://localhost:4224

load:
  - src/*.js
  - test/*.js

```

We can now schedule tests to run by issuing the command in Listing 3.10 or Listing 3.11, depending on your operating system.

Listing 3.10 Running tests with JsTestDriver on Linux and OSX

```

java -jar $JSTESTDRIVER_HOME/JsTestDriver-1.2.1.jar --tests
all

```

Listing 3.11 Running tests with JsTestDriver on Windows

```

java -jar %JSTESTDRIVER_HOME%\JsTestDriver-1.2.1.jar--tests
all

```

The default configuration file name is `jsTestDriver.conf`, and as long as this is used we don't need to specify it. When using another name, add the `--config path/to/file.conf` option.

When running tests, JsTestDriver forces the browser to refresh the test files. Source files, however, aren't reloaded between test runs, which may cause errors due to stale files. We can tell JsTestDriver to reload everything by adding the `--reset` option.

3.4.3.6 JsTestDriver and TDD

When TDD-ing, tests will fail frequently, and it is vital that we are able to quickly verify that we get the failures we expect in order to avoid buggy tests. A browser such as Internet Explorer is not suitable for this process for a few reasons. First, its error

messages are less than helpful; you have probably seen “Object does not support this property or method” more times than you care for. The second reason is that IE, at least in older versions, handles script errors badly. Running a TDD session in IE will cause it to frequently choke, requiring you to manually refresh it. Not to mention the lack of performance in IE, which is quite noticeable compared to, e.g., Google Chrome.

Disregarding Internet Explorer, I would still advise against keeping too many browsers in your primary TDD process, because doing so clutters up the test runner’s report, repeating errors and log messages once for every captured browser. My advice is to develop against one server that only captures your browser of choice, and frequently run tests against a second server that captures many browsers. You can run against this second server as often as needed—after each passed test, completed method, or if you are feeling bold, even more. Keep in mind that the more code you add between each run, the harder it will be to spot any bugs that creep up in those secondary browsers.

To ease this sort of development, it’s best to remove the `server` line from the configuration file and use the `--server` command line option. Personally I do this kind of development against Firefox, which is reasonably fast, has good error messages, and always runs on my computer anyway. As soon as I pass a test, I issue a run on a remote server that captures a wider variety of browsers, new and old.

3.4.4 Using JsTestDriver From an IDE

JsTestDriver also ships plugins for popular integrated development environments (IDEs), Eclipse and IntelliJ IDEA. In this section I will walk through setting up the Eclipse plugin and using it to support a test-driven development process. If you are not interested in developing in Eclipse (or Aptana), feel free to skip to Section 3.4.5, *Improved Command Line Productivity*.

3.4.4.1 Installing JsTestDriver in Eclipse

To get started you need to have Eclipse (or Aptana Studio, an IDE based on Eclipse aimed at web developers) installed. Eclipse is a free open source IDE and can be downloaded from <http://eclipse.org>. Once Eclipse is running, go to the *Help* menu and select *Install new software*. In the window that opens, enter the following URL as a new *update site*: <http://js-test-driver.googlecode.com/svn/update/>

“JS Test Driver Eclipse Plugin” should now be displayed with a checkbox next to it. Check it and click *Next*. The next screen is a confirmation that sums up the plugins to be installed. Click *Next* once again and Eclipse asks you to accept the

terms of use. Check the appropriate radio button and click *Next* if you accept. This should finish the installation.

Once the plugin is installed we need to configure it. Find the Preferences pane under the Window menu (Eclipse menu on OS X). There should be a new entry for Js Test Driver; select it. As a bare minimum we need to enter the port where Eclipse should run the server. Use 4224 to follow along with the example. You can also enter the paths to browsers installed locally to ease browser capturing, but it's not really necessary.

3.4.4.2 Running JsTestDriver in Eclipse

Next up, we need a project. Create a new project and enter the directory for the command line example as location. Now start the server. Locate the JsTestDriver panel in Eclipse and click the green play button. Once the server is running, click the browser icons to capture browsers (given that their path was configured during setup). Now right-click a file in the project, and select *Run As* and then *Run Configurations . . .* Select *Js Test Driver Test* and click the sheet of paper icon indicating “new configuration.” Give the configuration a name and select the project's configuration file. Now click run and the tests run right inside Eclipse, as seen in Figure 3.2.

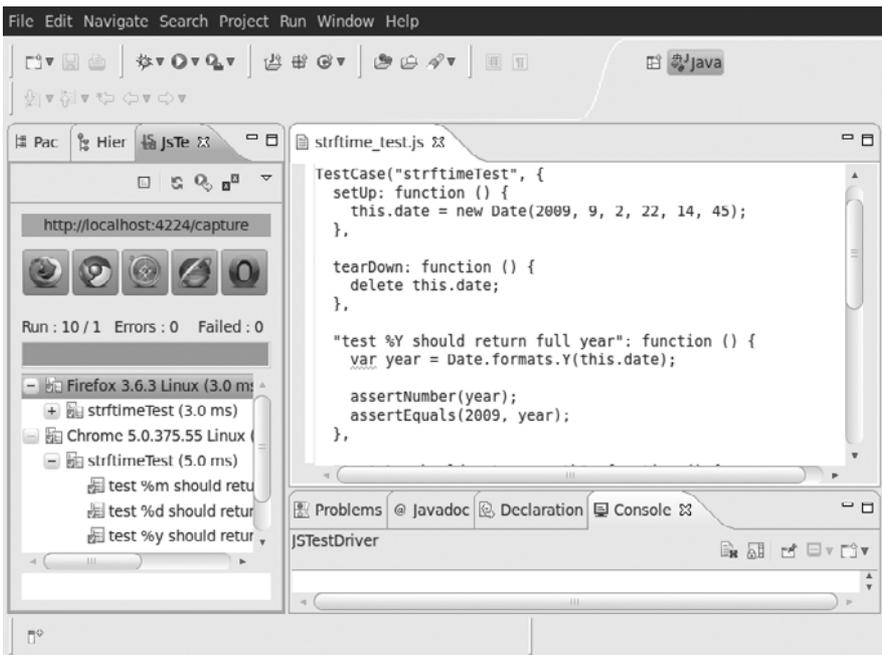


Figure 3.2 Running JsTestDriver tests inside Eclipse.

On subsequent runs, simply select *Run As* and then *Name of configuration*. Even better, check the *Run on every save* checkbox in the configuration prompt. This way, tests are run anytime a file in the project is saved, perfect for the test-driven development process.

3.4.5 Improved Command Line Productivity

If the command line is your environment of choice, the Java command to run tests quickly becomes a bit tiresome to type out. Also, it would be nice to be able to have tests run automatically whenever files in the project change, just like the Eclipse and IDEA plugins do. Jstdutil is a Ruby project that adds a thin command line interface to JsTestDriver. It provides a leaner command to run tests as well as an `jsautotest` command that runs related tests whenever files in the project change.

Jstdutil requires Ruby, which comes pre-installed on Mac OS X. For other systems, installation instructions can be found on ruby-lang.org. With Ruby installed, install Jstdutil by running `gem install jstdutil` in a shell. Jstdutil uses the previously mentioned `$JSTESTDRIVER_HOME` environment variable to locate the JsTestDriver jar file. This means that running tests is a simple matter of `jstestdriver --tests all`, or for autotest, simply `jsautotest`. If the configuration file is not automatically picked up, specify it using `jstestdriver --config path/to/file.conf --tests all`. The `jstestdriver` and `jsautotest` commands also add coloring to the test report, giving us that nice red/green visual feedback.

3.4.6 Assertions

JsTestDriver supports a rich set of assertions. These assertions allow for highly expressive tests and detailed feedback on failures, even when a custom assertion message isn't specified. The full list of supported assertions in JsTestDriver is:

- `assert(msg, value)`
- `assertTrue(msg, value)`
- `assertFalse(msg, value)`
- `assertEquals(msg, expected, actual)`
- `assertNotEquals(msg, expected, actual)`
- `assertSame(msg, expected, actual)`
- `assertNotSame(msg, expected, actual)`
- `assertNull(msg, value)`

- `assertNotNull(msg, value)`
- `assertUndefined(msg, value)`
- `assertNotUndefined(msg, value)`
- `assertNaN(msg, number)`
- `assertNotNaN(msg, number)`
- `assertException(msg, callback, type)`
- `assertNoException(msg, callback)`
- `assertArray(msg, arrayLike)`
- `assertTypeOf(msg, type, object)`
- `assertBoolean(msg, value)`
- `assertFunction(msg, value)`
- `assertNumber(msg, value)`
- `assertObject(msg, value)`
- `assertString(msg, value)`
- `assertMatch(msg, pattern, string)`
- `assertNoMatch(msg, pattern, string)`
- `assertTagName(msg, tagName, element)`
- `assertClassName(msg, className, element)`
- `assertElementId(msg, id, element)`
- `assertInstanceOf(msg, constructor, object)`
- `assertNotInstanceOf(msg, constructor, object)`

We will be using `JsTestDriver` for most examples throughout this book.

3.5 Summary

In this chapter we have taken a look at what tools can be helpful to support the test-driven development process, as well as a few available tools. Getting a good test-driven development rhythm requires adequate tools, and for the remaining examples of this book, `JsTestDriver` was selected to run tests. It offers both a highly efficient workflow as well as thorough testing on a wide array of platform and browser combinations.

This chapter also touched briefly on BDD and “specs” and how test-driven development, as practiced in this book, shares a lot in common with it.

Although we visited the topics of test coverage reports and continuous integration in this chapter, no setup or examples were given for such tools. On the book's website¹ you will find a guide to running the Coverage plugin for JsTestDriver as well as a guide on how to run JsTestDriver tests in the open source continuous integration server Hudson.

In the next chapter we will have a look at some other ways to utilize unit tests before we move on to Part II, *JavaScript for Programmers*.

1. <http://tddjs.com>

This page intentionally left blank

Index

A

acceptance test-driven development, 34

access tokens, 381–382, 385–386

embeds for, 385–386

updates for, 385

ActionScript, 159

activateTab method, 190–192

event delegation in, 190

implementation of, 192

tag name in, 190

testing for, 190–191

activation object, 82

Active X objects, 252

identifiers for, 252

addEventListener method, 206

custom event handlers, 212–213

addMessage, 361–363

as asynchronous, 365–366

callbacks with, 361–362

event emitters with, 374

promise refactoring with, 367–371

testing implementation for, 366

UIDs for, 362–363

updating of, 369

ad hoc scopes, 101–103

avoiding the global scope, 101–102

lightboxes and, 101–102

with nested closures, 103

simulation of, 102–103

AJAX. *See* **Asynchronous JavaScript and XML**

ajax.cometClient, 323–338

data delegation with, 324–325

data dispatching with, 323–327

error handling with, 325–327

event data looping with, 327

expectations for, 323

notifications with, 324–325

observers with, 325–329

public objects with, 326

server connections with, 329–338

setup for, 323

ajax.loadFragment method, 94

ajax.poll, 453

Ajax Push, 314

anonymous closures, 146

anonymous function expression, 74, 101–107.

See also **namespaces**

ad hoc scopes, 101–103

immediately called, 101–107

namespaces, 103–107

anonymous mocks, 454

anonymous proxy function, 95

APIs. *See* **application programming interfaces**

application programming interfaces (APIs),

247–249, 269–277. *See also* **DOM**

manipulation

AJAX, 269–277

integration test for, 269–271

local requests for, 273–274

send method and, 271

status testing for, 274–277

TDD and, 247

testing results for, 270–271

apply method, 75, 77

summing numbers with, 91

this keyword and, 90

arbitrary events, 241–246

- arbitrary events** (*Continued*)
 - notify method, 243–245
 - observe method, 241–242
- arbitrary objects**, 235–241
 - inheritance motivations, 235
 - observable behavior for, 235–236
 - renaming methods** and, 240–241
- arguments, 97–99
 - binding functions with, 97–99
 - bind method and, 97–99
 - formal parameters v., 173
 - in memoization, 114
 - passing**, 231–232
 - setTimeout method and, 97
- arguments object, 77–80, 153
 - accessing properties in, 79
 - array methods and, 78–79
 - dynamic mapping of, 79–80
 - formal parameters, 78–80
 - modifications of, 79
 - structure of, 78
- array literals, 118
- Array.prototype, 121–122
 - Enumerable module in, 157
 - method addition to, 122
 - native object extension for, 122
- Array.prototype.splice method, 56–58
 - arrays, 56
 - browser consoles, 57
 - programming, 58
 - removed item returns, 57
 - traditional testing, 56
- arrays
 - addObserver method and, 229
 - arguments object and, 78–79
 - Array.prototype.splice method, 56
 - in ECMAScript 5, 175–176
 - enumerable properties, 123
 - hard-coding, 225
 - in observer patterns, 224–225
 - with obsolete constructors, 238
 - for refactoring, 226
 - spliced, 57
 - for **this** keyword, 88
- assert function, 74
- assertions**, 9–10, 36
 - for controllers, 347
 - functions of, 9
 - JsTestDriver, 51–52
 - in POST requests, 283
 - testing for, 10
 - in unit tests, 465–466
- Asynchronous JavaScript and XML (AJAX)**, 247–292. *See also* **GET requests**; **POST Requests**
 - Ajax Push, 314
 - APIs**, 247–249, 269–277
 - baseline interfaces for, 290
 - browser inconsistencies with, 248
 - development strategy for, 248
 - directions for, 291
 - directory layout for, 249
 - duplication with, 292
 - GET requests**, 255–268
 - goals of, 248–249
 - implementation review for, 290
 - JsTestDriver and, 249–250
 - namespaces for, 256, 290
 - onreadystatechange handler and, 266–267
 - POST requests**, 277–287
 - refactoring with, 292
 - request APIs and, 247–249, 288–292
 - request interfaces and, 249–250
 - restoring of, 258
 - Reverse Ajax, 314
 - source files for, 256
 - stubbing and, 248–249
 - TDD and, 292
 - tddjs.ajax.create method and, 253–254
 - test cases for, 292
 - XMLHttpRequest object and, 247–249
- asynchronous tests**, 35
 - sleep function in, 35
 - unit tests and, 35
- automated stubbing**, 258–260, 262–263
 - helper method extraction with, 258–259
 - open method, 259–260
 - stub helper and, 259
- automated testing**, 3–19. *See also* **unit tests**
 - assertions**, 9–10
 - debugging with, 3
 - development of, 3–4
 - functions**, 11–12
 - green**, as symbol for success, 10
 - integration tests**, 14–16
 - JUnit, 4
 - red**, as symbol for failure in, 10
 - setUp method, 13–14
 - TDD, 30

tearDown method, 13–14
unit tests, 4–10, 16–18

B

BDD. *See* **behavior-driven development**

Beck, Kent, 21

behavior-driven development (BDD), 33–34
 TDD, 34

user stories in, 34
 xUnits and, 33

behavior verification, of test doubles, 442–443
 inspection of, 443

isolation of behavior from, 470–472

by **mocks**, 457, 470–472

stubbing and, 451–452, 470–472

tailored asserts for, 451

unit tests as, 465–466, 468–472

benchmarks, 60–69

binding functions and, 98

definition of, 60

DOM manipulation in, 68

`Function.prototype` in, 65–66

functions for, 65

highlighting in, 67–68

integration of, 64

loops for, 61–63, 66

measuring of, 67–68

reformatting of, 66

runners for, 61

setup for, 64

tools for, 64–65

use of, 66–67

in Windows Vista, 61

binding functions, 93–100

anonymous proxy functions and, 95

with arguments, 97–99

benchmarks and, 97

`bind` method, 95–97

currying and, 99–100

`Function.prototype.bind`, 95–96

lightbox examples of, 93–95

`setTimeout` method, 97

`this` keyword and, 93–96

`bind` method, 95–97

arguments and, 97–99

closure and, 96

implementation of, 96

optimized, 98–99

use of, 96

bogus headers, 308

bogus observers, 232–233

exceptions for, 233

non-callable arguments and, 232

preconditions for, 233

bootstrap scripts

in **chat client model**, 430

message lists and, 421

static files and, 410–411

bottlenecks, in performance tests, 68–69

DOM implementation in, 69

Firebug, 68

locating, 68–69

profiling, 68–69

`box-shadow` property, 209

browser sniffing, 199–207. *See also* **object**

detection, in browser sniffing

event listening fixes in, 198–199

libraries and, 200

object detection in, 199–206

problems with, 200

state of, 200

testing in, 207

updating of, 200

user agent sniffing and, 198–199

C

cache issues, with long polling,
 319–320

buster additions, 319–320

URLs and, 319–320

callable host objects, 203–204

callbacks, 308–311

with `addMessage`, 361–362

`complete`, 300–302, 311

defaults, 310

in domain models, for Node.js, 358

failure, 310–311

nested, 367

for `onreadystatechange` handler,
 266–268

polling, for data, 308–311

with server connections, 333

static files and, 409

success, 309–310

`tddjs.ajax.poller` and, 300–302

calling, of functions, 77–80

`arguments` object, 77–79

direct, 77–80

`call` method, 75, 77

`this` keyword and, 89

call order documentation, 234–235

as feature, 234

- cascading style sheets (CSS)**, 208–210
 - box-shadow property in, 209
 - feature testing of, 208–210
 - static files and, 410
 - style properties in, 209–210
 - support detection of, 209
- chat client model**, for DOM manipulation, 429–434
 - application styling of, 430–431
 - bootstrapping script in, 430
 - compression in, 434
 - deployment notes** in, 433–434
 - design of, 430–431
 - input field clearance in, 432–433
 - message forms in, 429
 - scrolling** in, 431–432
 - user testing of, 430–433
- chatRoom, 372–375
 - property descriptors for, 373
- Circle hybrid, 168–169
- circle objects, 88, 152
- Circle.prototype, 132–134, 136–137, 143
 - assignments for, 133
 - failing assertions for, 133–134
 - Sphere.prototype and, 138
 - super method, 143
 - testing for, 133
- circular references
 - assertions of, 272
 - breaking of, 272–273
 - with XMLHttpRequest object, 271–272
- clean code, in TDD, 28
- closure
 - ad hoc scopes, 103
 - anonymous, 146
 - in anonymous proxy function, 95
 - bind method and, 96
 - functions and, 84
 - for onreadystatechange handler, 267
 - private methods and, 145
- code paths**, from stubbing, 444–445
- Comet**, 314–315, 321–338. *See also*
 - ajax.cometClient; **server connections**
 - ajax.cometClient, 323–338
 - browsers with, 321
 - client interface with, 322
 - data publishing** with, 338
 - drawbacks to, 314
 - feature tests** for, 338
 - forever frames** and, 314–315
 - format messaging** with, 321–322
 - HTML5** streaming, 315
 - JSON response with, 322
 - limitations of, 314, 321
 - with observable objects, 321
 - server connections** with, 329–338
 - XMLHttpRequest streaming, 315
- command line productivity**, 51
- CommonJs modules**, 341, 345
- CommonJs testing frameworks, 40–41
- complete callbacks, 300–302, 311
 - scheduling of, 302
 - specifications of, 301–302
- console.log method, 76
- constructors**, 130–136. *See also* prototypes
 - broken properties of, 134
 - circle object, 139
 - ECMA-262 and, 136
 - instanceof operators, 136
 - missing properties for, 134–135
 - misuse of, 135–136
 - objects** from, 130–132, 239–240
 - in observer patterns, 223
 - private methods for, 146–147
 - problems with, 135–136
 - prototypes, 130–135
- continuous integration**, 34–35
 - for JavaScript, 34–35
 - minifying code with, 35
- controllers**, 345–357, 378–386
 - access tokens and, 381–382, 385–386
 - application testing, 356–357
 - application testing for, 386
 - assertions for, 347
 - body** of, 386
 - closing connections for, 355–356
 - closing responses for, 356
 - CommonJs modules**, 345
 - creation of, 346–347
 - done method, 346
 - duplications with, 350, 353
 - event handlers and, 352
 - expectations for, 345
 - formatting messages with, 383–385
 - GET requests** and, 380–386
 - JSON and, 347–350
 - malicious data with, 354
 - message extraction with, 351–354
 - message filters, 381–382
 - with message lists, 411–412
 - module definition**, 345–346

- MVC, 391
 - with **Node.js**, 345–357, 378–386
 - POST** messages, 347–354
 - post method completion with, 378–380
 - request bodies with, 348–351
 - request responses with, 354–356
 - respond method, 382–383, 386
 - response codes for, 355
 - response headers**, 386
 - servers, 356–357
 - setup for, 351
 - status codes for, 354–355
 - stubbing with, 348–349, 353
 - tabController object, 187–190
 - tab controllers**, 192–196
 - testing for, 346
 - in user forms, 392–393
- Crockford, Douglas, 148, 175, 333
- cross-browser event handling**, 210–213
 - addEventListener method in, 212–213
 - custom events in, 211–213
 - feature detection for, 210–211
 - normalization in, 211
- cross-browsers**
 - event handlers**, 210–213
 - IDE, 17
- crosscheck**, 42
- cross site scripting (XSS) protection, 418
- CSS. *See* **cascading style sheets**
- currying**, 99–100
 - binding v., 99
 - implementation of, 100
- D**
- Dahl, Ryan, 341
- data publishing**, with Comet, 338
- data streaming**, 293–339. *See also* **Comet**; **polling**, for data; **server connections**; `tddjs.ajax.poller`
 - with **Comet**, 314–315, 321–338
 - long polling** for, 315–320
 - polling** for, 294–313
 - server connections** and, 329–338
 - with Server Push, 293
 - TDD and, 293
- Date.formats.j method, 14
- Date.prototype.strftime, 7
- JsTestDriver, 47–48
- day of year calculations, 15
- debugging
 - assertions** and, 9
 - with automated testing, 3
- decoupled code, 22
- decrementing functions, 84
- dedicated respond method, 383
- dependencies**, 37
- Dojo libraries, 40
- domain models**, for Node.js, 358–366
 - addMessage in, 361–363
 - asynchronous interfaces, 358
 - bad data in, 359–361
 - callbacks in, 358
 - chart room creation, 358
 - getMessageSince method, 363–365
 - I/O interface**, 358
 - messages in, 359–366
 - usernames in, 359–361
- DOM events, 42, 207–208
 - in benchmarks, 68
 - in bottlenecks, 69
 - feature detection in, 207–208
 - feature testing** in, 207–208
 - in IE, 207
 - in lightbox objects, 94
 - observer patterns and, 220
- DOM manipulation**, 389–434. *See also* **chat client model**, for DOM manipulation; **message forms**; **message lists**, with DOM manipulation; **user forms**
 - approaches to, 390–391
 - chat client model** with, 429–434
 - client display, 391
 - directory structure for, 390
 - JsTestDriver configuration in, 390
 - message forms** with, 422–429
 - message lists** with, 411–421
 - MVC and, 391
 - MVP and, 391
 - passive view and, 391
 - static files** in, 408–411
 - TDD and, 389–434
 - user forms** and, 392–408
- done method, 346
- DontDelete attribute, 126
- DontEnum attribute, 126–128
- IE, 127
 - overriding properties for, 127
- dot notation, 118
- dummy objects**, 441

duplication

- with AJAX, 292
- with controllers, 350, 353
- status testing, for APIs, 274–275
- in **TDD**, 28
- test removal, 229–230
- with unit test, 467–468
- for XMLHttpRequest object, 253

E

Eclipse, 49–51

- JsTestDriver installation, 49–50
- running tests, 50–51

ECMA-262, 58, 118

- constructors and, 136
- properties and, 126
- prototypal inheritance and, 138
- in prototype chains, 119

ECMAScript 5, 25, 58, 159–176. *See also* **strict mode**, in ECMAScript 5

- ActionScript and, 159
- additions** to, 174–176
- arrays** in, 175–176
- backwards compatibility in, 159–160
- browser extensions in, 160
- Circle hybrid in, 168–169
- empowered properties, 162
- Enumerable module and, 161
- in execution contexts, 81
- Firefox and, 160
- Function.prototype and, 95
- Function.prototype.bind method in, 175
- get function, 161
- getters in, 166–167
- in global object, 82
- Google Chrome and, 160
- improvements** to, 174–176
- JScript.Net and, 159
- JSON** in, 175
- name/value assignment in, 161–162
- Object.create method in, 165–168
- object models** and, 161–171
- Object.seal implementation in, 163
- property attributes, 161–163, 167–170
- property descriptor changes in, 162
- prototypal inheritance** in, 164–166
- reserved keywords** in, 170–171

- server connections and, 333
- set function, 161
- setters in, 166–167
- shortcuts in, 164
- standard codification for, 160
- strict mode** in, 160, 171–174
- tddjs.extend method and, 156
- this** keyword and, 90–91
- writable function, 161
- encapsulation**, 145–150
 - private members** and, 147–148
 - private methods** and, 145–147
 - privileged methods** and, 147–148
 - radius property in, 148
- Enumerable module, 157–158
 - Array.prototype in, 157
 - in ECMAScript 5 object models, 161
- enumerable properties**, 122–126
 - looping arrays, 123
 - Object.prototype.hasOwnProperty, 124–126
 - running tests with, 123
- env.js** library, 42
- errback conventions, in Node.js, 358
- error handling**, 232–235
 - with ajax.cometClient, 325–327
 - bogus observer additions** and, 232–233
 - call order documentation** and, 234–235
 - forever frames and, 314
 - misbehaving observers** and, 233–234
- event emitters**, 372–378
 - addMessage with, 374
 - chatRoom with, 372–375
 - getMessageSince method, 376
 - waitForMessagesSince method, 375–378
- event handlers, 102–103
 - controllers and, 352
 - cross-browsers**, 210–213
 - handleSubmit method, 397–398
 - in object detection, 201
 - tabController object in, 187–188
 - unit tests and, 466
 - in unobtrusive JavaScript, 179
 - in user forms, 394–395
- event listeners, 394–398
 - application code for, 394–395
- events. *See* **arbitrary events**; **cross-browser event handling**; event handlers
- execution context**, 80–81

- ECMAScript specification, 81
 - this** keyword and, 88
 - variable object** in, 81–82
- expression**, functions, 74–75, 84–87
 - anonymous, 74
 - conditional declarations in, 85
 - conditional definitions in, 85
 - feature detection and, 85
 - hoisting in, 85
 - named, 75, 86–87
 - punctuation for, 75
 - `String.prototype.trim` method and, 85
- F**
- Facebook, 294
- failure callbacks, 310–311
- fake objects**, 440–441
- feature detection**, 85, 197–215
 - Browser sniffing**, 199–207
 - for **Comet**, 338
 - for **cross-browser event handling**, 210–213
 - in **DOM events**, 207–208
 - IE browsers and, 213
 - for **long polling**, 320
 - for **message forms**, 428–429
 - for **message lists**, 420
 - script production in, 215
 - self-testing code, 215
 - in `strftime`, 214
 - stubbing and, 263
 - undetectable features, 214
 - uses of, 213–214
 - for `XMLHttpRequest` object, 254
- Fibonacci sequence, 112–114
 - alternative versions of, 113
- Firebug, 68–69
 - `console.log` method in, 76
 - profiler for, 69
- Firefox
 - ECMAScript 5 and, 160
 - integration tests with, 270–271
- for**, as enumerable property, 123
- forever frames**, 314–315
 - error handling and, 314
- for-in**, as enumerable property, 123–124
- format specifiers, 15–16
- Fowler, Martin, 17, 391
- functional inheritance**, 148–150
 - definition of, 148
 - durable objects and, 149
 - implementation of, 148–149
 - object extension in, 149–150
 - patterns, 149
 - private variables with, 150
 - `Sphere.prototype` and, 150
- `Function.prototype`, 65–66, 75–78
 - `apply` method, 75, 77
 - binding functions and, 95–96
 - `call` method, 75, 77
 - ECMAScript 5 and, 95
 - function creation, 77
 - `Function.prototype.bind` method, 175
 - `Function.prototype.inherit` functions, 152–153
- functions**, 73–91. *See also* **anonymous function**
 - expression; arguments object; **binding functions**; **expression**, of functions; **stateful functions**; **this** keyword
 - activation object** and, 82
 - anonymous proxy, 95
 - arguments object and, 77–80
 - `assert`, 74
 - binding**, 93–100
 - calling of, 77–80
 - closure and, 84
 - declarations of, 73–74
 - decrementing, 84
 - definitions of, 73–77
 - execution contexts**, 80–81
 - expression of**, 74–75, 84–87
 - formal parameters of, 74
 - free variables, 84
 - `Function.prototype`, 75–78
 - global object** and, 82–83
 - hoisting of, 82, 85
 - incrementing, 84
 - length property, 76
 - `Object.prototype`, 75
 - scope**, 80–84
 - stateful**, 107–112
 - this** keyword, 87–91
- function scope, 80
- G**
- Geisendörfer, Felix, 408
- `getMessageSince` method, 363–365
 - addition of, 364
 - message retrieval testing with, 363–365
 - with promises, 372
 - proxy for, 376
- `getPanel` function, 193–195
 - toggles in, 193–194

- GET requests**, 255–268
 - automated stubbing** and, 258–260, 262–263
 - controllers** and, 380–386
 - formatting messages with, 383–385
 - improved stubbing** and, 261–263
 - manual stubbing** and, 257–258
 - `onreadystatechangehandler`, 263–268
 - POST requests** and, 285–287
 - `respond` method, 382–384
 - stubbing**, 257–263
 - `tdajs.ajax.create` object and, 255
 - URL requirement for, 255–256
- `getters`, 166–167
- Giammarchi, Andrea, 208
- global object**, 82–83
 - `Array.prototype` and, 122
 - ECMAScript in, 82
 - property assignment in, 83
 - this** keyword in, 88
 - window and, 83
- global scope**, 80, 101–102
- Gmail, unobtrusive JavaScript in, 184
- Gnome Shell, 160
- Google Chrome, 160
- green**, as symbol for success in unit testing, 10
- GTalk, 294
- H**
- `handleSubmit` method, 397–398, 401–402, 404
 - message forms and, 425
- hard-coding, 27, 225–226
 - in `addObserver` method, 227
 - for arrays, 225
 - for inputs, 27
 - for outputs, 27
- headers**, in data polling, 308–311
 - bogus, 308
 - passing on, 309
- headless testing frameworks**, 41–42
 - crosscheck**, 42
 - DOM implementation, 42
 - `env.js` library, 42
 - issues with, 42
 - Rhino**, 42
- Heilmann, Chris, 178
- hoisting, of functions, 82, 85
- host objects**, 202–204
 - callable, 203–204
 - ECMAScript specification in, 202
 - feature detection in, 204
 - in IE, 202
 - unfriendly, 203
- HTML5** streaming, 315
- Hypertext Markup Language (HTML), 269–271
 - in Comet, streaming for, 315
 - integration testing, 269–271
 - in `JsTestDriver`, 400
 - in static files, 409–410
 - in unobtrusive JavaScript, 177
 - user form embedding with, 400–401
- I**
- IDE. *See* integrated development environment
- IE. *See* Internet Explorer
- immediately called anonymous functions**, 101–107
 - ad hoc scopes** and, 101–103
 - punctuation and, 101
- improved stubbing**, 261–263
- in-browser test frameworks**, 37–43. *See also*
 - YUI test**
 - disadvantages of, 42–43
 - Dojo, 40
 - headless**, 41–42
 - JsTestDriver**, 43–51
 - `JsUnit`, 37, 40
 - `Prototype.js`, 40
 - `QUnit`, 40
 - URL query string, 37
 - YUI test**, 38–40
- incrementing functions, 84
- inheritance models, 119–120
 - `Object.create` method, 151
- inputs
 - for hard-coding, 27
 - in TDD, 24–25
- `instanceof` operators, 136
- integrated development environment (IDE), 17, 49–51. *See also* Eclipse
 - Eclipse, 49–51
 - IntelliJ IDEA, 49
 - `JsTestDriver`, 49–51
- integration tests**, 14–16
 - for APIs, 269–271
 - `Date.formats.j` method, 14
 - for day of year calculations, 15
 - with Firefox, 270–271
 - format specifiers in, 15–16
 - high-level, 14
 - HTML document testing, 269
 - script for, 269–270

- IntelliJ IDEA, 49
- Internet Explorer (IE), 127–128
 - addObserver method, 228
 - DOM events in, 207
 - DOMEnum attribute in, 127
 - feature detection and, 213
 - host objects in, 202
 - named function expressions in, 86–87
 - Object.defineProperty in, 166
 - XMLHttpRequest object and, 252
- I/O interfaces**, 358
- iterators**, 109–112
 - closures, 109
 - functional approach to, 111–112
 - looping with, 112
 - tddjs.iterator method, 109–111
- J**
- Jar file**, 44–45
 - on Linux, 45
 - starting servers, 45–46
 - for Windows users, 45
- JavaScript. *See also* **Asynchronous JavaScript and XML**; **Node.js**; **unobtrusive JavaScript**
 - ECMAScript 5 in, 25
 - JsLint**, 474
 - Mozilla, 58
 - observer pattern in, 220–221
 - programming of, 58–59
 - unit tests, 55–60
 - unobtrusive**, 177–196
 - writing cross-browser code in, 197
- JavaScriptCore, 58
- JavaScript dates**, 5–9
 - strftime for, 5–9
- jQuery
 - performance tests, 69
 - tabbed panels, 196
 - in unobtrusive JavaScript, 195–196
- JScript.Net, 58, 159
- JsLint**, 474
- JSON**, support for, 175
 - in **Comet**, 322
 - controllers, in Node.js, 347–350
 - server connections and, 331, 333–334
- JsTestDriver**, 43–52. *See also* **Jar file**
 - AJAX and, 249–250
 - assertions**, 51–52
 - browser capture for, 46
 - in browsers, 43
 - command line productivity**, 51
 - configuration files for, 249–250
 - configuration for, 48
 - Date.prototype.strftime, 47–48
 - disadvantages of, 44
 - in DOM manipulation, 390
 - functions of, 43–44
 - HTML in, 400
 - IDE, 49–51
 - Jar file**, 44–45
 - Linux testing, 48
 - load paths, 46
 - observer patterns and, 221
 - OSX testing, 48
 - plug-ins, 43
 - polling data and, 295
 - project layout for, 249–250
 - running tests for, 46–48
 - server connections and, 333
 - setup, 44–49
 - starting servers for, 45–46
 - TDD and, 48–49
 - timer testing, 303–308
 - uid's and, 108
 - updating of, 262
 - user form configurations, 404
 - Windows testing for, 48
- JUnit
 - in In-Browser test frameworks, 37
 - testing frameworks, 4, 37, 40
 - timer testing, 303–304
- L**
- learning tests, 56, 59–60
 - bugs and, 59
 - frameworks, 60
 - new browsers, 59
 - wisdom from, 59
- lightbox objects, 93–95
 - ad hoc scopes** and, 101–102
 - ajax.loadFragment method, 94
 - pseudo code for, 94
- Linux
 - ECMAScript 5 and, 160
 - Jar file on, 45
 - JsTestDriver testing, 48
 - load paths, 46–47
- local requests**, 273–274
 - success handler for, 273–274
 - URLs and, 274
- long polling**, 315–320
 - cache issues** with, 319–320

long polling (*Continued*)

- feature tests** for, 320
 - implementation of, 316–319
 - low latency from, 316
 - stubbing dates with, 316–319
- looping properties, 128–130
- `ajax.cometClient`, 327

M**manual stubbing**, 257–258**memoization**, 112–115

- argument serialization in, 114
- definition of, 112
- Fibonacci sequence in, 112–114
- general methods, 113–114
- limiting of, 114

`messageFormController`, 424

message forms, 422–429

- acquisition of, 428
- in chat client model, 429
- for current users, 426–428
- empty function additions in, 426
- extraction of, 423
- feature tests** for, 428–429
- `handleSubmit` method and, 425
- message clearance in, 433
- message form controllers and, 422
- `messageFormController` with, 424
- publishing of, 425–428
- refactoring of, 423–425
- `setModel` moving in, 425
- TDD and, 428
- test setup with, 422
- `userFormController` with, 423–424
- view setting with, 422–425

`messageListController`, 412

message lists, with DOM manipulation, 411–421

- `addMessage` with, 413–414
- bootstrap scripts and, 421
- controller definition with, 411–412
- feature tests** for, 420
- initialization of, 420–421
- message addition to, 416–418
- `messageListController`, 412
- model setting, 411–414
- node lists and, 419
- `observe` method with, 413
- reference storage with, 417
- repeated messages in, 418–420
- scrolling** of, 432

- `setModel` in, 413

- `setView` method and, 393, 414–416

- subscription to, 412–414

- user additions, 416

- user tracking in, 419

- view settings, 414–416

- XSS protection in, 418

Meszaros, Gerard, 440

misbehaving observers, 233–234

- exceptions, 234

mixins, 157–158

- definition of, 157

- `Enumerable` module and, 157–158

mocks, 453–458

- `ajax.poll`, 453

- anonymous**, 454

- automatic verification of, 454

- behavior verification** with, 457, 470–472

- definition of, 453

- dependency silencing by, 457

- method restoration of, 453–454

- multiple expectations of, 455–456

- `notify` method and, 454

- in POST requests, 284

- `stubs.v.`, 457–458

- for `tdajs.ajax.poller`, 298–299

- `this` value, 456

Model-View-Controller (MVC), 391

Model-View-Presenter (MVP), 391

- axis for, 391

- components for, 391

- passive view in, 391

module patterns, 107

mouseover events, 184

Mozilla, 58

MVC. *See* Model-View-Controller

MVP. *See* Model-View-Presenter

N

named function expressions, 75

- in Internet Explorer, 86–87

namespace method, 187

namespaces, 103–107

- for AJAX, 256, 290

- custom creation of, 106

- definition of, 105–106

- functions of, 104–105

- implementation of, 104–106

- importing, 106–107

- in libraries, 104

- native, 103
 - objects as, 103–104
 - for XMLHttpRequest object, 251
 - name tabbed panels, 182
 - name tests**, 462
 - native objects**, 202–204
 - ECMAScript specification in, 202
 - nested callbacks, 367
 - new operators, 131–132
 - Node.js**, 341–387. *See also* **controllers**; **domain models**, for Node.js; **promises**, with Node.js
 - access tokens in, 381–382, 385–386
 - assertions for, 347
 - controllers** with, 345–357, 378–386
 - directory structure for, 342–343
 - domain models**, 358–366
 - environments for, setting up, 342–343
 - event emitters**, 372–378
 - events with, 342
 - framework testing for, 343
 - HTTP server, 344
 - message filters, 381–382
 - nested callbacks and, 367
 - node-paperboy, 408–409
 - promises** with, 367–372
 - respond method with, 382–383
 - runtime**, 341–344
 - servers with, 343–344
 - starting point** for, 343–344
 - startup scripts for, 344
 - static files**, 408–411
 - storage** for, 358–366
 - stubbing** and, 452
 - test scripts for, 343
 - node lists, 419
 - node-paperboy, 408–409
 - notify method, 243–245
 - arguments for, 243
 - implementation of, 245
 - mocks and, 454
 - relevant observers for, 243–244
 - storage of, 244–245
 - testing for, 244
 - updating of, 245
- O**
- object(s)**, 117–136, 150–157. *See also* **arbitrary objects**; **private methods**, for objects
 - arbitrary**, 235–241
 - arguments, 153
 - circle, 152
 - composition, 150–157
 - from **constructors**, 130–132, 239–240
 - direct inheritance in, 151
 - ECMA-262, 118
 - encapsulation** of, 145–150
 - in functional inheritance, 149–150
 - information hiding** and, 145–150
 - inspection of, 131
 - mixins**, 157–158
 - new operators, 131–132
 - Object.create method, 151–153
 - object literals, 117–118
 - Object.prototype.hasOwnProperty, 125
 - observable, 239–240
 - private methods** for, 145–147
 - prototype chains**, 119–122
 - prototypes, 130–135
 - radius property, 131
 - sphere, 151–152
 - in strict mode, 174
 - tddjs.extend method, 153–157
 - Object.create method, 151–153, 168–169
 - direct inheritance in, 151
 - ECMAScript 5 and, 165–166, 167–168
 - for function creation, 169–170
 - Function.prototype.inherit function, 152–153
 - implementation of, 152, 165–166
 - inheritance models, 151
 - with properties, 165
 - Object.defineProperty, 166
 - object detection**, in browser sniffing, 199–206
 - addEventListener method and, 206
 - event handling in, 201
 - host objects** and, 202–204
 - individual features of, 200
 - native objects** and, 202–204
 - premise of, 200
 - purposes of, 200–206
 - sample use testing** in, 204–206
 - strftime and, 204–206
 - testing of, 201
 - type checking** in, 201–202
 - object literals, 117–118
 - object model**, **ECMAScript 5** and, 161–171
 - Circle hybrid in, 168–169
 - empowered properties, 162
 - Enumerable module and, 161

object model, ECMAScript 5 and (*Continued*)

- get function, 161
 - getters in, 166–167
 - name/value assignment in, 161–162
 - Object.create method in, 165–168
 - Object.seal implementation in, 163
 - property attributes, 161–163, 167–170
 - property descriptor changes in, 162
 - prototypal inheritance** in, 164–166
 - reserved keywords** in, 170–171
 - set function, 161
 - setters in, 166–167
 - shortcuts in, 164
 - writable, 161
- Object.prototype, 75, 120–121
- Object.prototype.hasOwnProperty, 124–126
- browsers in, 125
 - loop qualification, 124
 - objects in, 125
- Object.seal method, 163
- observable objects, 239–240
- with Comet, 321
- observe method, 241–242
- call updating, 241–242
 - formal parameters for, 242
 - message lists, 413
- observer notification**, 230–232
- calls, 230–231
 - passing arguments** in, 231–232
- observer pattern**, 219–246. *See also* **arbitrary objects; bogus observers; error handling; observer notification**
- adding constructors in, 223
 - adding observers to, 222–225
 - addObserver method with, 224–230
 - for **arbitrary events**, 241–246
 - for **arbitrary objects**, 235–241
 - arrays in, 224–225
 - code writing for, 220
 - configuration files with, 221
 - definition of, 219
 - directory layouts in, 221
 - DOM events and, 220
 - environment setting for, 221
 - error handling** in, 232–235
 - in JavaScript, 220–221
 - JsTestDriver and, 221
 - Observable constructors with, 222
 - observe method, 241–242
 - observer notification**, 230–232

- refactoring** with, 17, 225–226, 229–230
 - roles within, 219–220
 - search results for, 220
 - stubbing and, 445
 - testing, 222–225
- observers, with ajax.cometClient, 325–329
- addition of, 327–329
 - saving of, 328
 - testing of, 328
 - type checking of, 329
- obsolete constructors**, 236–238
- addObserver method and, 237
 - array definition with, 238
 - emptying of, 238
- one-liners**, 311–313
- poller interfaces, 311
 - start method and, 312–313
 - URLs and, 313
- onreadystatechangehandler, 263–268
- AJAX and, 266–267
 - anonymous closure of, 267
 - assignment verification for, 264
 - callbacks for, 266–268
 - empty, 264
 - handling of, 265–268
 - send method, 264–265
 - testing of, 265–268
- open method, 259–260
- OSX, JsTestDriver testing for, 48
- outputs
- in hard-coding, 27
 - in TDD, 24–25

P

- passing arguments**, 231–232
 - test confirmation and, 231
- performance tests**, 60–69
- benchmarks**, 60–69
 - bottlenecks**, 68–69
 - closures, 60
 - footprints for, 63
 - jQuery, 69
 - relative performance** of, 60–69
 - setTimeout calls, 63
 - YUI, 63
- Plug-ins, for JsTestDriver, 43
- polling**, for data, 294–313. *See also*
- tddjs.ajax.poller
 - callbacks** and, 308–311
 - directory layout in, 294

- in Facebook, 294
- final version of, 313
- in GTalk, 294
- headers** and, 308–311
- jsTestDriver and, 295
- load order with, 295
- one-liners** and, 311–313
- project layout in, 294–295
- with server connections, 330, 334
- with `tdajs.ajax.poller`, 295–302
- timer testing**, 303–308
- post method, 378–380
 - closing connections with, 379
 - response times with, 380
 - verification delay with, 379
- POST requests**, 277–287
 - assertions in, 283
 - configuration methods with, 278–279
 - copy-pasting for, 278
 - cropping, 280
 - data additions, 286–287
 - data handling functions in, 284–285
 - data transport for, 282–287
 - delegation to, 281
 - encoding data in, 283–285
 - expectation of, 281
 - extraction of data in, 278, 285
 - GET requests** and, 285–287
 - implementation of, 277–281
 - introductions for, 281
 - method call changes for, 280
 - mocking in, 284
 - Node.js messages, 347–354
 - ReadyStateHandlerTest, 280
 - setting headers for, 287
 - string encoding with, 282
 - stubbing in, 284
 - in TDD, 279
 - test cases for, 279–280
 - updating of, 280
 - URLs and, 282, 285
- private methods**, for objects, 145–147
 - closures and, 145
 - definition of, 145–146
 - function object creations in, 147
 - inside constructors, 146–147
- promises**, with Node.js, 367–372
 - addMessage refactoring, 367–371
 - consumption of, 371–372
 - definition of, 367
 - getMessageSince method
 - with, 372
 - grouping of, 371–372
 - nested callbacks, 367
 - rejection of, 369–370
 - resolution of, 370–371
 - resolve method with, 367
 - returning, 368–369
 - test conversion with, 371
 - then method with, 369
- properties**, prototypal inheritance and, 117–130
 - access, 118–119
 - attributes, 126–130
 - DontDelete attribute for, 126
 - DontEnum attribute for, 126–128
 - dot notation in, 118
 - ECMA-262 and, 126
 - enumerable, 122–126
 - inheritance, 120–121
 - looping, 128–130
 - names, with spaces, 119
 - ReadOnly attribute for, 126
 - shadowing, 120–121
 - square bracket notation in, 118
 - test methods for, 119
 - toString method and, 119
 - values for, 120
 - whitespace and, 118
- property identifiers**, reserved keywords and, 170–171
- prototypal inheritance**, 117–130, 136–144, 158.
 - See also functional inheritance*; *—super method*
 - access in, 138–139
 - Circle.prototype, 136–137
 - ECMA-262 and, 138
 - in **ECMAScript 5**, 164–166
 - functional**, 148–150, 158
 - functions, 137–138
 - implementation of, 138
 - properties** and, 117–130
 - specifications for, 137
 - Sphere.prototype, 136–137
 - super, as concept, 139–144
 - super method*, 140–143
 - surface area calculations for, 139–140
- prototype chains**, 119–122
 - Array.prototype, 121–122
 - ECMA-262 specification in, 119
 - inheritance models, 119–120

prototype chains, (*Continued*)

- object extension through, 121–122
- Object.prototype, 120–121
- Prototype.js library, 40
- prototypes, 130–135
 - Circle.prototype, 132–134, 136–137, 143
 - constructors, 130, 132
 - property additions to, 132–135

Q

QUnit testing frameworks, 40

R

- radius property, 131
 - in encapsulation, 148
- ReadOnly attribute, 126
- ReadyStateHandlerTest, 280
- red**, as symbol for failure in unit testing, 10
- refactoring**, 17, 225–226, 229–230
 - with addMessage, 367–371
 - with addObserver method, 225
 - with AJAX, 292
 - arrays for, 226
 - duplicated test removal, 229–230
 - hard-coding and, 225–226
 - of **message forms**, 423–425
 - method renaming and, 17
 - in notify method, 245
 - with **observer pattern**, 17
 - TDD** and, 28
 - test failure and, 17
 - unit tests**, 17
- regression testing**, 16
- renaming methods**, 240–241
- reserved keywords**, 170–171
 - property identifiers and, 170–171
- Resig, John, 42
- resolve method, 367
- respond method, 382–384, 386
 - dedicated, 383
 - initial tests for, 382–384
- response codes, 355
- Reverse Ajax, 314
- Rhino**, 42
- S**
- saboteurs, 445
- scope**, 80–84
 - Ad Hoc**, 101–103
 - blocking of, 80
 - chains in, 83–84
 - function, 80, 82
 - global, 80, 101–102
 - scope chain**, 83–84
 - decrementing functions, 84
 - incrementing functions, 84
 - scrolling**, 431–432
 - of message lists, 432
 - stubbing in, 432
 - send method
 - onreadystatechangehandler and, 264–265
 - server connections**, 329–338
 - callbacks with, 333
 - concerns with, 334–338
 - custom headers with, 336
 - data dispatching with, 332–334
 - ECMAScript5 and, 333
 - exceptions to, 331
 - JSON data and, 331, 333–334
 - JsTestDriver and, 333
 - missing URLs and, 331
 - obtaining of, 329
 - polling for, 330, 334
 - request headers with, 337
 - response data in, 332
 - tokens with, 336
 - Server Push, 293
 - setModel additions, 402
 - with message forms, 425
 - with message lists, 413
 - setters, 166–167
 - setTimeout calls, 63
 - setTimeout method, binding
 - arguments, 97
 - setUp function, xUnits and, 35
 - setUp method, 13–14
 - setView method, 393, 414–416
 - compliant, 415
 - single responsibility principle**, 30–31
 - sleep function, 35
 - slice function, 153
 - sphere objects, 151–152
 - Sphere.prototype, 136–137
 - Circle.prototype and, 138
 - functional inheritance and, 150
 - implementation of, 143
 - super method, 143
 - testing for, 137
 - spliced arrays, 57
 - square bracket notation, 118
 - start method, 296–298

- additions of, 297
 - definition for, 297
 - one-liners and, 312–313
 - polling for data and, 312–313
- stateful functions**, 107–112. *See also* **iterators**
 - generating uid's, 107–109
 - iterators**, 109–112
 - memoization**, 112–115
 - module patterns, 107
- state verification**, of test doubles, 442
- static files**, 408–411
 - bootstrap scripts and, 410–411
 - callbacks and, 409
 - chapp's servers and, 409
 - CSS files, 410
 - HTML in, 409–410
- status codes, 354–355
- status testing**, for APIs, 274–277
 - coding in, 276–277
 - duplication reduction and, 274–275
 - fake requests and, 275
 - request helpers for, 275–276
 - success/failure callbacks and, 277
 - TDD and, 276
- storage**
 - with **message lists**, 417
 - for **Node.js**, 358–366
 - of notify method, 244–245
 - for uid's, variable states, 109
 - unit tests and, 4
 - in user forms, 403
- strftime, 5–9
 - Date.prototype.strftime, 7
 - defining of, 205
 - feature detection in, 214
 - Firebug session and, 7
 - implementation of, 205–206
 - object detection and, 204–206
 - restructuring of, 12
 - starting point for, 5–6
 - test cases with, 12
 - test pages with, 8
 - use of, 205–206
 - YUI test and, 38–40
- strict mode**, in ECMAScript 5, 160, 171–174
 - changes, 172–174
 - enabling of, 171–172
 - formal parameters in, 172–173
 - functions in, 172–174
 - global object, 171
 - implicit globals in, 172
 - local, 171–172
 - objects** in, 174
 - properties** in, 174
 - restrictions** in, 174
 - variables** in, 174
- String.prototype.trim method, 24–25
 - function expression and, 85
 - successful testing of, 27, 29
 - test failure and, 25
- stubbing**, 257–263, 443–445, 447–452
 - AJAX and, 248–249
 - automated**, 258–260, 262–263
 - behavior verification** with, 451–452, 470–472
 - code paths** from, 444–445
 - with controllers, 348–349, 353
 - Date, 316–319
 - DOM and, 444
 - feature detection** and, 263
 - global methods and, 448
 - improved**, 261–263
 - inconvenient interfaces and, 444
 - libraries**, 447–452
 - with long polling, 316–320
 - manual**, 257–258
 - mocks v., 457–458
 - Node.js** and, 452
 - Observer pattern and, 445
 - in POST requests, 284
 - saboteurs, 445
 - in scrolling, 432
 - for `tdajs.ajax.poller`, 298–299
 - test doubles** and, 443–445, 447–452
 - testing timers and, 303, 305
 - test spies** with, 445–446
 - throwaway, 448
 - with user forms, 397, 403
 - with `waitForMessagesSince` method, 375–376
 - of XMLHttpRequest object, 248–249, 257–263
- stubbing Date, 316–319
 - fixed output constructors, 316
 - intervals between, 318
 - requests with, 317
 - testing with, 317–319
 - timers and, 318–319
- stub helper, 259
- stub libraries**, 447–452
 - automatic management with, 449–450
 - automatic restoring of, 450

stub libraries, (*Continued*)

- functions of, 448
- manual spying with, 448–449
- methods, 448–450
- Observer patterns and, 447

success callbacks, 309–310

- passing of, 310

SUnit, 5

super, as concept, 139–144

—super method, 140–143

- Circle.prototype, 143
- helper functions, 143–144
- implementation of, 142, 144
- performance of, 143
- Sphere.prototype, 143
- testing of, 141
- try-catch and, 143

T

tabbed panels, 179–182, 185–196

- activateTab method, 190–192
- class names in, 186–187
- clean markup for, 181–182
- getPanel function in, 194–195
- jQuery, 196
- name, 182
- namespace method and, 187
- shared setUp, 186
- styles for, 182

tabController object, 187–190

tab controllers in, 192–196

- tddjs.extend method and, 187
- in TDD projects, 185
- testing for, 186–187

tabController object, 187–190

- behaviors of, 189
- DOM event listener, 188–189
- event handlers in, 187–188
- implementation of, 188, 190
- test cases for, 188

tab controllers, 192–196

- getPanel function in, 193–195

TDD. *See* test-driven development

tddjs.ajax.create object, 253–254

- Get requests and, 255

tddjs.ajax.poller, 295–302

- callbacks and, 300–302
- definition of, 296
- exceptions for, 297
- expectations of, 296
- object definition with, 296

requests for, 299–300

running tests in, 300

start method for, 296–298

stubbing strategy for, 298–299

URLs, 297–299

tddjs.extend method, 153–157

arrays, 153

Boolean strings in, 156

dummy objects, 155

ECMAScript 3 and, 156

ECMAScript 5 and, 156

explicit borrowing in, 154

implementation of, 155

initial testing in, 154–155

method collection in, 154

null method, 155–156

single arguments in, 156

slice functions, 153

sources in, 156

tabbed panels and, 187

tddjs.iterator method, 109–111

implementation of, 110–111

tearDown function, in xUnits, 35

tearDown method, 13–14, 307

testability, of unit tests, 18

testCase function, 11–12

test coverage report, 36

test doubles, 439–459. *See also* stubbing

definition of, 439

dummy objects and, 441

fake objects and, 440–441

mocks and, 453–458

overview of, 439–441

real-life comparisons to, 440

stubbing and, 443–445, 447–452

verification of, 441–443

test-driven development (TDD), 21–31

acceptance of, 34

AJAX and, 292

APIs and, 247

autotesting in, 30

BDD and, 34

benefits of, 30–31

clean code in, 28

conscious development in, 31

data streaming and, 293

decoupled code in, 22

design, 22–23

development cycle changes for, 22

DOM manipulation and, 389–434

duplication, 28

- ECMAScript 5 in, 25
 - facilitation** of, 29–30
 - goals of, 21
 - hard-coding in, 27
 - inputs for, 24–25
 - JsTestDriver and, 48–49
 - message forms and, 428
 - outputs for, 24–25
 - POST requests and, 279
 - process** of, 23–29
 - productivity boosts from, 31
 - purpose of, 21
 - refactoring**, 28
 - sample code in, 22
 - single responsibility principle**, 30–31
 - status testing and, 276
 - `String.prototype.trim` method and, 24–25
 - successful testing of, 26–27
 - in tabbed panels, 185
 - test failure for, 25
 - test-writing for, 24–25
 - unobtrusive JavaScript and, 182
 - workable code from, 30
 - YAGNI methodology for, 26
 - test functions**, 11–12
 - testing. *See* automated testing
 - testing timers**, with polling, 303–308
 - configurable intervals, 306–308
 - extraction with, 306
 - helper methods and, 304
 - JsUnit and, 303–304
 - new request scheduling, 304–306
 - required waits with, 306
 - running tests, 306
 - scheduling with, 305
 - stubbing and, 303, 305
 - `tearDown` methods, 307
 - test reports, 36
 - test runner**, 35–36
 - test coverage reports for, 36
 - test reports for, 36
 - test spies**, 445–447
 - detail inspection with, 446–447
 - indirect input testing with, 446
 - `then` method, 369
 - this** keyword, 87–91
 - anonymous proxy function, 95
 - `apply` method and, 90
 - array methods for, 88
 - behaviors of, 87–88
 - binding functions and, 93–96
 - Boolean strings and, 89–90
 - calling functions and, 89
 - `call` method and, 89
 - `circle` object, 88
 - ECMAScript 5 mode, 90–91
 - execution contexts and, 88
 - explicit setting** for, 89
 - in global objects, 88
 - implicit setting** for, 88–89
 - mocks and, 456
 - primitives as, 89–91
 - summing numbers with, 90–91
 - values for, 88
 - throwaway stubs, 448
 - `toString` method, 119
 - try-catch, 143
 - Twitter, search feature for, 69
 - type checking**, 201–202
 - features of, 201–202
- ## U
- `uid`'s. *See* unique IDs
 - unfriendly host objects, 203
 - uniform resource locators (URLs)
 - cache issues, 319–320
 - for Get requests, 255–256
 - local requests and, 274
 - one-liners and, 313
 - POST requests and, 282, 285
 - query string, 37
 - server connections and, 331
 - `tddjs.ajax.poller` and, 297–299
 - unique IDs (`uid`'s), 107–109
 - for `addMessage`, 362–363
 - free variable storage states and, 109
 - implementation of, 108–109
 - JsTestDriver and, 108
 - specification of, 107–108
 - unit tests**, 4–10, 16–18, 461–475
 - `Array.prototype.splice` method, 56–58
 - assertions in, 465–466
 - asynchronous tests, 35
 - behavior verification**, 465–466, 468–472
 - benefits** of, 16–18
 - bugs** in, 473–475
 - code breaking in, 474
 - Cross-Browser testing**, 17
 - definition of, 4
 - disk storage and, 4

unit tests (*Continued*)

- domain specific test helpers and, 466
- duplication with, 467–468
- event handlers and, 466
- exercise structure for, 464–465
- formatting of, 464–465
- functionality testing for, 57
- green**, as symbol for success in, 10
- high-level abstractions in, 465–466
- JavaScript, 55–60
- JavaScript dates**, 5–9
- JsLint** and, 474
- learning tests and, 56
- name tests** for, 462
- pitfalls of, 18
- readability** of, 462–468
- red**, as symbol for failure in, 10
- refactoring**, 17
- regression**, 16
- scannability of, 462–463
- setup structure for, 464–465
- SUnit, 5
- technical limitations of, 463–464
- testability of, 18
- test case functions, 463–464
- verify structure for, 464–465
- whitespace matching, 58–59
- writing of, 57
- writing** of, 461–475
- xUnits and, 5

unobtrusive JavaScript, 177–196

- accessibility of, 178
- assumptions** in, 183–184
- clean code in, 177
- code decoupling in, 179
- definition of, 177
- event delegation in, 179
- event handlers in, 179
- extensibility of, 178
- fallback solutions in, 183–184
- flexibility of, 178
- global footprint of, 183
- in Gmail, 184
- goals of, 177–178
- isolation within, 183
- jQuery in, 195–196
- mouseover events in, 184
- performance of, 178
- progressive enhancement in, 182

- robustness in, 178
- rules** of, 178–182, 184–185
- semantic HTML in, 177
- tabbed panels** in, 179–182, 185–196
- tabcontroller object in, 187–190
- TDD and, 182
- WCAG for, 184

URLs. *See* uniform resource locators

user agent sniffing, 198–199

userFormController, 423–424

user forms, 392–408

- class additions to, 393–394
- class removals to, 406
- controller definitions in, 392–393
- default action changes, 398–400
- event handlers in, 394–395
- event listener additions to, 394–398
- handleSubmit method with, 397–398, 401–402, 404
- HTML embeds with, 400–401
- JsTestDriver configuration, 404
- namespace method in, 187, 395
- observer notifications with, 403–406
- reference storage in, 403
- setModel additions, 402
- setUp code extraction, 396
- setup sharing for, 399–400
- setView method with, 393
- stubbing with, 397, 403
- submit events with, 398–407
- test cases for, 392–393
- test setup with, 405
- usernames in, 401–403
- view setting with, 392–398

usernames, 406–408

- in domain models, 359–361
- feature tests for, 407–408
- rejection of, 406–407
- in user forms, 401–403

user stories, 34

V

variable object, in execution context, 81–82

verification, of test doubles, 441–443

- behavior**, 442–443
- implications of, 443
- of mocks, 454
- stages of, 441
- state**, 442

W

`waitForMessagesSince` method, 375–378

- listener additions in, 376–377
- message listener implementation with, 377
- resolution with, 375
- stubbing with, 375–376

WCAG. *See* web content accessibility guidelines

web content accessibility guidelines (WCAG), 184

whitespace

- matching, 58–59
- properties and, 118

Windows

- Jar file for, 45
- JsTestDriver tests, 48

Windows Vista, benchmarks in, 61

X

`XMLHttpRequest` object, 247–254, 263–268.

See also **long polling**

- Active X objects and, 252
- background for, 251–253
- browser inconsistencies with, 248
- circular references with, 271–272
- code duplication for, 253
- in Comet, 315
- creation of, 250–254
- development strategy for, 248
- extraction of, 262
- feature detection for, 254
- goals of, 248–249
- IE and, 252
- instantiation of, 252
- interface style for, 250
- long polling**, 315–320
- namespace creation for, 251

`onreadystatechange` handler, 263–268

- running tests for, 253
- standards for, 251–252
- stubbing of, 248–249, 257–263
- support for, 254
- testing of, 251

XSS protection. *See* cross site scripting protection

xUnits, 5, 33, 35–37

- assertions**, 36
- BDD, 33
- dependencies**, 37
- `setUp` function in, 35
- special values for, 36
- `tearDown` function in, 35
- test frameworks for, 35–36
- test reports for, 36
- test runner for, 35–36

Y

YAGNI methodology. *See* “You ain’t gonna need it” methodology

“You ain’t gonna need it” (YAGNI) methodology, 26

YUI test, 38–40

- HTML fixture file, 38
- as performance test, 63
- for production code, 40
- running tests, 40, 41
- setup of, 38–40
- `strftime` file, 38–40

Z

Zaytsev, Juriy, 207

Zyp, Kris, 367