

Bear Bibeault
Yehuda Katz

jQuery in Action



jQuery in Action

jQuery in Action

BEAR BİBEAULT
YEHUDA KATZ



MANNING

Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please go to www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
Sound View Court 3B
Greenwich, CT 06830

Fax: (609) 877-8256
Email: orders@manning.com

©2008 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15% recycled and processed without the use of elemental chlorine.



Manning Publications Co.
Sound View Court 3B
Greenwich, CT 06830

Copyeditor: Andrea Kaucher
Typesetter: Denis Dalinnik
Cover designer: Leslie Haimes

ISBN 1-933988-35-5

Second, corrected printing May 2008

Printed in the United States of America

2 3 4 5 6 7 8 9 10 – MAL – 12 11 10 09 08

contents

foreword xi
preface xiii
acknowledgments xvi
about this book xix
about the authors xxiv
about the title xxvi
about the cover illustration xxvii

1 **Introducing jQuery 1**

1.1 Why jQuery? 2

1.2 Unobtrusive JavaScript 3

1.3 jQuery fundamentals 5

The jQuery wrapper 6 ▪ *Utility functions 8* ▪ *The document ready handler 9* ▪ *Making DOM elements 11* ▪ *Extending jQuery 12* ▪ *Using jQuery with other libraries 14*

1.4 Summary 14

2 **Creating the wrapped element set 16**

2.1 Selecting elements for manipulation 17

Using basic CSS selectors 19 ▪ *Using child, container, and attribute selectors 20* ▪ *Selecting by position 24*
Using custom jQuery selectors 27

- 2.2 Generating new HTML 31
- 2.3 Managing the wrapped element set 32
 - Determining the size of the wrapped set* 34
 - *Obtaining elements from the wrapped set* 34
 - *Slicing and dicing the wrapped element set* 36
 - *Getting wrapped sets using relationships* 43
 - Even more ways to use a wrapped set* 44
 - *Managing jQuery chains* 45
- 2.4 Summary 47

3 ***Bringing pages to life with jQuery*** 48

- 3.1 Manipulating element properties and attributes 49
 - Manipulating element properties* 51
 - *Fetching attribute values* 52
 - *Setting attribute values* 54
 - *Removing attributes* 56
 - *Fun with attributes* 56
- 3.2 Changing element styling 58
 - Adding and removing class names* 58
 - *Getting and setting styles* 61
 - *More useful style-related commands* 67
- 3.3 Setting element content 68
 - Replacing HTML or text content* 68
 - *Moving and copying elements* 70
 - *Wrapping elements* 75
 - Removing elements* 76
 - *Cloning elements* 78
- 3.4 Dealing with form element values 79
- 3.5 Summary 81

4 ***Events are where it happens!*** 82

- 4.1 Understanding the browser event models 84
 - The DOM Level 0 Event Model* 85
 - *The DOM Level 2 Event Model* 91
 - *The Internet Explorer Event Model* 97
- 4.2 The jQuery Event Model 98
 - Binding event handlers using jQuery* 98
 - *Removing event handlers* 103
 - *Inspecting the Event instance* 104
 - Affecting the event propagation* 106
 - *Triggering event handlers* 106
 - *Other event-related commands* 107
- 4.3 Putting events (and more) to work 112
- 4.4 Summary 124

- 5 Sprucing up with animations and effects 126**
- 5.1 Showing and hiding elements 127
 - Implementing a collapsible list 128* ▪ *Toggling the display state of elements 134*
 - 5.2 Animating the display state of elements 135
 - Showing and hiding elements gradually 135* ▪ *Fading elements into and out of existence 140* ▪ *Sliding elements up and down 143* ▪ *Stopping animations 145*
 - 5.3 Creating custom animations 145
 - A custom scale animation 148* ▪ *A custom drop animation 148*
 - A custom puff animation 150*
 - 5.4 Summary 152
- 6 jQuery utility functions 153**
- 6.1 Using the jQuery flags 154
 - Detecting the user agent 155* ▪ *Determining the box model 161*
 - Detecting the correct float style to use 163*
 - 6.2 Using other libraries with jQuery 163
 - 6.3 Manipulating JavaScript objects and collections 167
 - Trimming strings 168* ▪ *Iterating through properties and collections 169* ▪ *Filtering arrays 170*
 - Translating arrays 172* ▪ *More fun with JavaScript arrays 175* ▪ *Extending objects 176*
 - 6.4 Dynamically loading scripts 180
 - 6.5 Summary 184
- 7 Extending jQuery with custom plugins 185**
- 7.1 Why extend? 186
 - 7.2 The jQuery plugin authoring guidelines 187
 - Naming files and functions 187* ▪ *Beware the \$ 189*
 - Taming complex parameter lists 190*
 - 7.3 Writing custom utility functions 192
 - Creating a data manipulation utility function 193*
 - Writing a date formatter 195*

- 7.4 Adding new wrapper methods 199
 - Applying multiple operations in a wrapper method* 201
 - Retaining state within a wrapper method* 206
- 7.5 Summary 216

8 **Talk to the server with Ajax** 217

- 8.1 Brushing up on Ajax 218
 - Creating an XHR instance* 219
 - Initiating the request* 221
 - Keeping track of progress* 222
 - Getting the response* 223
- 8.2 Loading content into elements 224
 - Loading content with jQuery* 226
 - Loading dynamic inventory data* 229
- 8.3 Making GET and POST requests 233
 - Getting data with jQuery* 234
 - Getting JSON data* 236
 - Making POST requests* 248
- 8.4 Taking full control of an Ajax request 249
 - Making Ajax requests with all the trimmings* 249
 - Setting request defaults* 252
 - Global functions* 253
- 8.5 Putting it all together 258
 - Implementing the flyout behavior* 259
 - Using The Termifier* 262
 - Room for improvement* 264
- 8.6 Summary 266

9 **Prominent, powerful, and practical plugins** 268

- 9.1 The Form Plugin 269
 - Getting form control values* 270
 - Clearing and resetting form controls* 274
 - Submitting forms through Ajax* 276
 - Uploading files* 284
- 9.2 The Dimensions Plugin 285
 - Extended width and height methods* 285
 - Getting scroll dimensions* 287
 - Of offsets and positions* 289
- 9.3 The Live Query Plugin 292
 - Establishing proactive event handlers* 292
 - Defining match and mismatch listeners* 294
 - Forcing Live Query evaluation* 294
 - Expiring Live Query listeners* 295

9.4	Introduction to the UI Plugin	299
	<i>Mouse interactions</i>	300
	▪ <i>UI widgets and visual effects</i>	316
9.5	Summary	316
9.6	The end?	317
<i>appendix</i>	<i>JavaScript that you need to know but might not!</i>	319
	<i>index</i>	339

foreword

It's all about simplicity. Why should web developers be forced to write long, complex, book-length pieces of code when they want to create simple pieces of interaction? There's nothing that says that complexity has to be a requirement for developing web applications.

When I first set out to create jQuery I decided that I wanted an emphasis on small, simple code that served all the practical applications that web developers deal with day to day. I was greatly pleased as I read through *jQuery in Action* to see in it an excellent manifestation of the principles of the jQuery library.

With an overwhelming emphasis on practical, real-world code presented in a terse, to-the-point format, *jQuery in Action* will serve as an ideal resource for those looking to familiarize themselves with the library.

What's pleased me the most about this book is the significant attention to detail that Bear and Yehuda have paid to the inner workings of the library. They were thorough in their investigation and dissemination of the jQuery API. It felt like nary a day went by in which I wasn't graced with an email or instant message from them asking for clarification, reporting newly discovered bugs, or recommending improvements to the library. You can be safe knowing that the resource that you have before you is one of the best thought-out and researched pieces of literature on the jQuery library.

One thing that surprised me about the contents of this book is the explicit inclusion of jQuery plugins and the tactics and theory behind jQuery plugin development. The reason why jQuery is able to stay so simple is through the

use of its plugin architecture. It provides a number of documented extension points upon which plugins can add functionality. Often that functionality, while useful, is not generic enough for inclusion in jQuery itself—which is what makes the plugin architecture necessary. A few of the plugins discussed in this book, like the Forms, Dimension, and LiveQuery plugins, have seen widespread adoption and the reason is obvious: They're expertly constructed, documented, and maintained. Be sure to pay special attention to how plugins are utilized and constructed as their use is fundamental to the jQuery experience.

With resources like this book the jQuery project is sure to continue to grow and succeed. I hope the book will end up serving you well as you begin your exploration and use of jQuery.

JOHN RESIG
CREATOR OF jQUERY

preface

One of your authors is a grizzled veteran whose involvement in programming dates back to when FORTRAN was the bomb, and the other is an enthusiastic domain expert, savvy beyond his years, who's barely ever known a world without an Internet. How did two people with such disparate backgrounds come together to work on a joint project?

The answer is, obviously, *jQuery*.

The paths by which we came together over our affection for this most useful of client-side tools are as different as night and day.

I (Bear) first heard of jQuery while I was working on *Ajax in Practice*. Near the end of the creation cycle of a book is a whirlwind phase known as the *copy-edit* when the chapters are reviewed for grammatical correctness and clarity (among other things) by the copyeditor and for technical correctness by the technical editor. At least for me, this is the most frenetic and stressful time in the writing of a book, and the *last* thing I want to hear is “you really should add a completely new section.”

One of the chapters I contributed to *Ajax in Practice* surveys a number of Ajax-enabling client-side libraries, one of which I was already quite familiar with (Prototype) and others (the Dojo Toolkit and DWR) on which I had to come up to speed pretty quickly.

While juggling what seemed like a zillion tasks (all the while holding down a day job, running a side business, and dealing with household issues),

the technical editor, Valentin Crettaz, casually drops this bomb: “So why don’t you have a section on jQuery?”

“J who?” I asked.

I was promptly treated to a detailed dissertation on how wonderful this fairly new library was and how it really should be part of any modern examination of Ajax-enabling client-side libraries. I asked around a bit. “Have any of you ever heard of this jQwerty library?”

I received a large number of positive responses, all enthusiastic and all agreeing that jQuery really was the cat’s pajamas. On a rainy Sunday afternoon, I spent about four hours at the jQuery site reading documentation and writing little test programs to get a feel for the jQuery way of doing things. Then I banged out the new section and sent it to the technical editor to see if I had really gotten it.

The section was given an enthusiastic thumb’s up, and we went on to finally complete the *Ajax in Practice* book. (That section on jQuery eventually went on to be published in the online version of *Dr. Dobb’s Journal*.)

When the dust had settled, my frenzied exposure to jQuery had planted relentless little seeds in the back of my mind. I’d liked what I’d seen during my headlong research into jQuery, and I set out to learn more. I started using jQuery in web projects. I still liked what I saw. I started replacing older code in previous projects to see how jQuery would simplify the pages. And I *really* liked what I saw.

Enthusiastic about this new discovery and wanting to share it with others, I took complete leave of my senses and submitted a proposal for *jQuery in Action* to Manning. Obviously, I must’ve been convincing. (As penance for causing such mayhem, I asked the technical editor who started all the trouble to also be the technical editor for *this* book. I’ll bet *that* taught him!)

It’s at that point that the editor, Mike Stephens, asked, “How would you like to work with Yehuda Katz on this project?”

“Yehenta who?” I asked...

Yehuda came to this project by a different route; his involvement with jQuery predates the days when it even had version numbers. After he stumbled on the Selectables Plugin, his interest in the jQuery core library was piqued. Somewhat disappointed by the (then) lack of online documentation, he scoured the wikis and established the Visual jQuery site (visualjquery.com).

Before too long, he was spearheading the push for better online documents, contributing to jQuery, and overseeing the plugin architecture and ecosystem, all while evangelizing jQuery to the Ruby community.

Then came the day when he received a call from Manning (his name having been dropped to the publisher by a friend), asking if he'd be interested in working with this Bear guy on a jQuery book...

Despite the differences in our backgrounds, experiences, and strengths, and the manner in which we came together on this project, we've formed a great team and have had a lot of fun working together. Even geographic distance (I'm in the heart of Texas, and Yehuda is on the California coast) proved no barrier. Thank goodness for email and instant messaging!

We think that the combination of our knowledge and talents brings you a strong and informative book. We hope you have as much fun reading this book as we had working on it.

We just advise you to keep saner hours.

acknowledgments

Have you ever been surprised, or even bemused, by the seemingly endless list of names that scrolls up the screen during the ending credits of a motion picture? Do you ever wonder if it really takes that many people to make a movie?

Similarly, the number of people involved in the writing of book would probably be a big surprise to most people. It takes a large collaborative effort on the part of many contributors with a variety of talents to bring the volume you are holding (or ebook that you are reading onscreen) to fruition.

The staff at Manning worked tirelessly with us to make sure that this book attained the level of quality that we hoped for, and we thank them for their efforts. Without them, this book would not have been possible. The “end credits” for this book include not only our publisher, Marjan Bace, and editor Mike Stephens, but also the following contributors: Douglas Pudnick, Andrea Kaucher, Karen Tegtmayer, Katie Tenant, Megan Yockey, Dottie Marsico, Mary Piergies, Tiffany Taylor, Denis Dalinnik, Gabriel Dobrescu, and Ron Tomich.

Enough cannot be said to thank our peer reviewers who helped mold the final form of the book, from catching simple typos to correcting errors in terminology and code and even helping to organize the chapters within the book. Each pass through a review cycle ended up vastly improving the final product. For taking the time to help review the book, we’d like to thank Jonathan Bloomer, Valentin Crettaz, Denis Kurilenko, Rama Krishna Vavilala, Philip Hallstrom, Jay Blanchard, Jeff Cunningham, Eric Pascarello, Josh Heyer, Gregg Bolinger, Andrew Siemer, John Tyler, and Ted Goddard.

Very special thanks go to Valentin Crettaz who served as the book’s technical editor. In addition to checking each and every sample of example code in multiple environments, he also offered invaluable contributions to the technical accuracy of the text.

BEAR BIBEAULT

For this, my third published tome, the cast of characters I’d like to thank has a long list of “usual suspects,” including, once again, the membership and staff at javaranch.com. Without my involvement in JavaRanch, I’d never have gotten the opportunity to start writing in the first place, and so I sincerely thank Paul Wheaton and Kathy Sierra for starting the whole thing, as well as fellow staffers who gave me encouragement and support, including (but probably not limited to) Eric Pascarello, Ben Souther, Ernest Friedman Hill, Mark Herschberg, and Max Habbibi.

Thanks go out to Valentin Crettaz—not only for serving as technical editor but also for introducing me to jQuery in the first place—and to my coworker Daniel Hedrick who volunteered the PHP examples for the latter part of the book.

My partner Jay, and dogs, Little Bear (well, we couldn’t have named him just *Bear*, now could we?) and Cozmo, get the usual warm thanks for putting up with the shadowy presence who shared their home but who rarely looked up from the MacBook Pro keyboard for all the months it took to write this book.

And finally I’d like to thank my coauthor, Yehuda Katz, without whom this project would not have been possible.

YEHUDA KATZ

To start, I’d like to thank Bear Bibeault, my coauthor, for the benefit of his extensive writing experience. His talented writing and impressive abilities to navigate the hurdles of professional publishing were a tremendous part of what made this book possible.

While speaking of making things possible, it’s necessary that I thank my lovely wife Leah, who put up with the long nights and working weekends for far longer than I would have felt comfortable asking. Her dedication to completing this book rivaled even my own; and, as in all things, she made the most difficult part of this project bearable. I love you, Leah.

Obviously, there would be no *jQuery in Action* without the jQuery library itself. I’d like to thank John Resig, the creator of jQuery, for changing the face of client-side development and easing the burden of web developers across the

globe (believe it or not, we have sizable user groups in China, Japan, France, and many other countries). I also count him as a friend who, as a talented author himself, helped me to prepare for this tremendous undertaking.

There would be no jQuery without the incredible community of users and core team members, including Brandon Aaron and Jörn Zaefferer on the development team; Rey Bango and Karl Swedberg on the evangelism team; Paul Bakaus, who heads up jQuery UI; and Klaus Hartl and Mike Alsup, who work on the plugins team with me. This great group of programmers helped propel the jQuery framework from a tight, simple base of core operations to a world-class JavaScript library, complete with user-contributed (and modular) support for virtually any need you could have. I'm probably missing a great number of jQuery contributors; there are a lot of you guys. Suffice it to say that I would not be here without the unique community that has come up around this library, and I can't thank you enough.

Lastly, I want to thank my family whom I don't see nearly enough since my recent move across the country. Growing up, my siblings and I shared a tight sense of camaraderie, and the faith my family members have in me has always made me imagine I can do just about anything. Mommy, Nikki, Abie, and Yaakov: thank you, and I love you.

about this book

Do more with less.

Plain and simple, that is the purpose of this book: to help you learn how to do more on your web application pages with less script. Your authors, one a jQuery contributor and evangelist and the other an avid and enthusiastic user, believe that jQuery is the best library available today to help you do just that.

This book is aimed at getting you up and running with jQuery quickly and effectively and, hopefully, having some fun along the way. The entire core jQuery API is discussed, and each API method is presented in an easy-to-digest syntax block that describes the parameters and return values of the method. Small examples of using the APIs effectively are included; and, for those *big concepts*, we provide what we call *lab pages*. These comprehensive and fun pages are an excellent way for you to see the nuances of the jQuery methods in action without the need to write a slew of code yourself.

All example code and lab pages are available for download at <http://www.manning.com/bibeault>.

We could go on and on with some marketing jargon telling you how great this book is, but you don't want to waste time reading that, do you? What you really want is to get your arms into the bits and bytes up to your elbows, isn't it?

What's holding you back? Read on!

Audience

This book is aimed at novice to advanced web developers who want to take control of the JavaScript on their pages and produce great, interactive Rich Internet Applications without the need to write all the client-side code necessary to achieve such applications from scratch.

All web developers who yearn to create usable web applications that delight, rather than annoy, their users by leveraging the power that jQuery brings to them will benefit from this book.

Although novice web developers may find some sections a tad involved, this should not deter them from diving into this book. We've included an appendix on essential JavaScript concepts that help in using jQuery to its fullest potential, and such readers will find that the jQuery library itself is novice-friendly once they understand a few key concepts—all without sacrificing the power available to the more advanced web developers.

Whether novices or veterans of web development, client-side programmers will benefit greatly from adding jQuery to their repertoire of development tools. We know that the lessons within this book will help add this knowledge to your toolbox quickly.

Roadmap

This book is organized to help you wrap your head around jQuery in the quickest and most efficient manner possible. It starts with an introduction to the design philosophies on which jQuery was founded and quickly progresses to fundamental concepts that govern the jQuery API. We then take you through the various areas in which jQuery can help you write fabulous client-side code, from the handling of events all the way to making Ajax requests to the server. To top it all off, we take a survey of some of the most popular jQuery extensions.

In chapter 1, we'll learn about the philosophy behind jQuery and how it adheres to modern principles such as Unobtrusive JavaScript. We examine why we might want to adopt jQuery and run through an overview of how it works, as well as the major concepts such as document-ready handlers, utility functions, Document Object Model (DOM) element creation, and how jQuery extensions are created.

Chapter 2 introduces us to the concept of the jQuery wrapped set—the core concept around which jQuery operates. We'll learn how this wrapped set—a collection of DOM elements that's to be operated upon—can be created by selecting elements from the page document using the rich and powerful collection of

jQuery *selectors*. We'll see how these selectors, while powerful, leverage knowledge that we already possess by using standard CSS notation.

In chapter 3, we'll learn how to use the jQuery wrapped set to manipulate the page DOM. We cover changing the styling and attributes of elements, setting element content, moving elements around, and dealing with form elements.

Chapter 4 shows us how we can use jQuery to vastly simplify the handling of events on our pages. After all, handling user events is what makes Rich Internet Applications possible, and anyone who's had to deal with the intricacies of event handler across the differing browser implementations will certainly appreciate the simplicity that jQuery brings to this particular area.

The world of animations and effects is the subject of chapter 5. We'll see how jQuery makes creating animated effects not only painless but also efficient and fun.

In chapter 6, we'll learn about the utility functions and flags that jQuery provides, not only for page authors, but also for those who will write extensions and plugins for jQuery.

We present writing such extensions and plugins in chapter 7. We'll see how jQuery makes it extraordinarily easy for anyone to write such extensions without intricate JavaScript or jQuery knowledge and why it makes sense to write any reusable code as a jQuery extension.

Chapter 8 concerns itself with one of the most important areas in the development of Rich Internet Applications: making Ajax requests. We'll see how jQuery makes it almost brain-dead simple to use Ajax on our pages and how it shields us from all the pitfalls that can accompany the introduction of Ajax to our pages, while vastly simplifying the most common types of Ajax interactions (such as returning JSON constructs).

Finally, in chapter 9 we'll take a survey of the most popular and powerful of the vast multitude of jQuery plugins and make sure that we know where we can find information on even more such plugins. We examine plugins that enable us to deal with forms and Ajax submissions with even more power than core jQuery and those that let us employ drag-and-drop on our pages.

We provide an appendix highlighting key JavaScript concepts such as *function contexts* and *closures*—essential to making the most effective use of jQuery on our pages—for those who would like a refresher on these concepts.

Code conventions

All source code in listings or in the text is in a fixed-width font like this to separate it from ordinary text. Method and function names, properties, XML elements, and attributes in the text are also presented in this same font.

In some cases, the original source code has been reformatted to fit on the pages. In general, the original code was written with page-width limitations in mind, but sometimes you may find a slight formatting difference between the code in the listings and that provided in the source download. In a few rare cases, where long lines could not be reformatted without changing their meaning, the book listings will contain line-continuation markers.

Code annotations accompany many of the listings, highlighting important concepts. In many cases, numbered bullets link to explanations that follow in the text.

Code downloads

Source code for all the working examples in this book (along with some extras that never made it into the text) is available for download from <http://www.manning.com/jqueryinAction> or <http://www.manning.com/bibeault>.

The code examples for each chapter are organized to be easily served by a local web server. Unzip the downloaded code into a folder of your choice, and make that folder the document root of the application. A launch page is set up at the application root in the file `index.html`.

With the exception of the examples for chapter 8 and a handful from chapter 9, most of the examples don't require the presence of a web server and can be loaded directly into a browser for execution. Instructions for easily setting up Tomcat to use as the web server for these examples is provided in file `chapter8/tomcat.pdf`.

All examples were tested in a variety of browsers that include Internet Explorer 7, Firefox 2, Opera 9, Safari 2, and Camino 1.5. The examples will also generally run in Internet Explorer 6 although some layout issues might be encountered. Note that all jQuery code works flawlessly in IE6—it's the CSS of the examples that cause any layout anomalies. Because the target audience of this book is professional web developers, it's assumed that all readers will have a variety of browsers available in which to execute the example code.

Author Online

The purchase of *jQuery in Action* includes free access to a private forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and other users. To access and subscribe to the forum, point your browser to <http://www.manning.com/jqueryinAction> or <http://www.manning.com/bibeault>. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct in the forum. (Play nice!)

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It's not a commitment to any specific amount of participation on the part of the authors, whose contribution to the book's forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions, lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the authors



BEAR BİBEAULT has been writing software for over three decades, starting with a Tic-Tac-Toe program written on a Control Data Cyber supercomputer via a 100-baud teletype. Because he has two degrees in Electrical Engineering, Bear should be designing antennas or something; but, since his first real job with Digital Equipment Corporation, he has always been much more fascinated with programming.

Bear has also served stints with companies such as Lightbridge Inc., BMC Software, Dragon Systems, and even served in the U. S. Military teaching infantry soldiers how to blow up tanks. (Care to guess which job was the most fun?) Bear is currently a Software Architect and Technical Manager for a company that builds and maintains a large financial web application used by the accountants that many of the Fortune 500 companies keep in their dungeons.

In addition to his day job, Bear also writes books (duh!), runs a small business that creates web applications and offers other media services (but not wedding videography, never wedding videography), and helps to moderate JavaRanch.com as a “sheriff.” When not planted in front of a computer, Bear likes to cook *big* food (which accounts for his jeans size), dabble in photography and video editing, ride his Yamaha V-Star, and wear tropical print shirts.

He works and resides in Austin, Texas, a city he dearly loves except for the completely insane drivers.



YEHUDA KATZ has been involved in a number of open-source projects over the past several years. In addition to being a core team member of the jQuery project, he is also a contributor to Merb, an alternative to Ruby on Rails (also written in Ruby).

Yehuda was born in Minnesota, grew up in New York, and now lives in sunny Santa Barbara, California. He has worked on websites for the *New York Times*, *Allure Magazine*, *Architectural Digest*, *Yoga Journal*, and other similarly high-profile clients. He has programmed professionally in a number of languages including Java, Ruby, PHP, and JavaScript.

In his copious spare time, he maintains VisualjQuery.com and helps answer questions from new jQuery users in the IRC channel and on the official jQuery mailing list.

about the title

By combining introductions, overviews, and how-to examples, the *In Action* books are designed to help learning *and* remembering. According to research in cognitive science, the things people remember are things they discover during self-motivated exploration.

Although no one at Manning is a cognitive scientist, we are convinced that for learning to become permanent it must pass through stages of exploration, play, and, interestingly, re-telling of what is being learned. People understand and remember new things, which is to say they master them, only after actively exploring them. Humans learn *in action*. An essential part of an *In Action* book is that it is example-driven. It encourages the reader to try things out, to play with new code, and explore new ideas.

There is another, more mundane, reason for the title of this book: Our readers are busy. They use books to do a job or solve a problem. They need books that allow them to jump in and jump out easily and learn just what they want just when they want it. They need books that aid them *in action*. The books in this series are designed for such readers.

about the cover illustration

The figure on the cover of *jQuery in Action* is called “The Watchman.” The illustration is taken from a French travel book, *Encyclopedie des Voyages* by J. G. St. Saveur, published in 1796. Travel for pleasure was a relatively new phenomenon at the time and travel guides such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other regions of the world, as well as to the regional costumes and uniforms of French soldiers, civil servants, tradesmen, merchants, and peasants.

The diversity of the drawings in the *Encyclopedie des Voyages* speaks vividly of the uniqueness and individuality of the world’s towns and provinces just 200 years ago. This was a time when the dress codes of two regions separated by a few dozen miles identified people uniquely as belonging to one or the other. The travel guide brings to life a sense of isolation and distance of that period and of every other historic period except our own hyperkinetic present.

Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life. Or a more varied and interesting intellectual and technical life.

We at Manning celebrate the inventiveness, the initiative, and the fun of the computer business with book covers based on the rich diversity of regional life two centuries ago brought back to life by the pictures from this travel guide.

Introducing jQuery

1

This chapter covers

- Why you should use jQuery
- What *Unobtrusive JavaScript* means
- The fundamental elements and concepts of jQuery
- Using jQuery in conjunction with other JavaScript libraries

Considered a “toy” language by serious web developers for most of its lifetime, JavaScript has regained its prestige in the past few years as a result of the renewed interest in Rich Internet Applications and Ajax technologies. The language has been forced to grow up quickly as client-side developers have tossed aside cut-and-paste JavaScript for the convenience of full-featured JavaScript libraries that solve difficult cross-browser problems once and for all and provide new and improved paradigms for web development.

A relative latecomer to this world of JavaScript libraries, jQuery has taken the web development community by storm, quickly winning the support of major websites such as MSNBC, and well-regarded open source projects including SourceForge, Trac, and Drupal.

Compared with other toolkits that focus heavily on clever JavaScript techniques, jQuery aims to change the way that web developers think about creating rich functionality in their pages. Rather than spending time juggling the complexities of advanced JavaScript, designers can leverage their existing knowledge of Cascading Style Sheets (CSS), Extensible Hypertext Markup Language (XHTML), and good old straightforward JavaScript to manipulate page elements directly, making more rapid development a reality.

In this book, we’re going to take an in-depth look at what jQuery has to offer us as page authors of Rich Internet Applications. Let’s start by finding out what exactly jQuery brings to the page-development party.

1.1 Why jQuery?

If you’ve spent any time at all trying to add dynamic functionality to your pages, you’ve found that you’re constantly following a pattern of selecting an element or group of elements and operating upon those elements in some fashion. You could be hiding or revealing the elements, adding a CSS class to them, animating them, or modifying their attributes.

Using raw JavaScript can result in dozens of lines of code for each of these tasks. The creators of jQuery specifically created the library to make common tasks trivial. For example, designers will use JavaScript to “zebra-stripe” tables—highlighting every other row in a table with a contrasting color—taking up to 10 lines of code or more. Here’s how we accomplish it using jQuery:

```
$("#table tr:nth-child(even)").addClass("striped");
```

Don’t worry if that looks a bit cryptic to you right now. In short order, you’ll understand how it works, and you’ll be whipping out your own terse—but powerful—



Year	Make	Model
1965	Ford	Mustang
1970	Toyota	Corolla
1979	AMC	Jeep CJ-5
1983	Ford	EXP
1985	Dodge	Daytona
1990	Chrysler	Jeep Wrangler Sahara
1995	Ford	Ranger
1997	Chrysler	Jeep Wrangler Sahara
2000	Chrysler	Jeep Wrangler Sahara
2005	Chrysler	Jeep Wrangler Unlimited
2007	Dodge	Caliber R/T

Figure 1.1
Adding “zebra stripes” to a table is easy to accomplish in one statement with jQuery!

jQuery statements to make your pages come alive. Let’s briefly examine how this code snippet works.

We identify every even row (`<tr>` element) in all of the tables on the page and add the CSS class `striped` to each of those rows. By applying the desired background color to these rows via a CSS rule for class `striped`, a single line of JavaScript can improve the aesthetics of the entire page.

When applied to a sample table, the effect could be as shown in figure 1.1.

The real power in this jQuery statement comes from the *selector*, an expression for identifying target elements on a page that allows us to easily identify and grab the elements we need; in this case, every even `<tr>` element in all tables. You’ll find the full source for this page in the downloadable source code for this book in file `chapter1/zebra.stripes.html`.

We’ll study how to easily create these selectors; but first, let’s examine how the inventors of jQuery think JavaScript can be most effectively used in our pages.

1.2 Unobtrusive JavaScript

Remember the bad old days before CSS when we were forced to mix stylistic markup with the document structure markup in our HTML pages?

Anyone who’s been authoring pages for any amount of time surely does and, perhaps, with more than a little shudder. The addition of CSS to our web development toolkits allows us to separate stylistic information from the document structure and give travesties like the `` tag the well-deserved boot. Not only does the separation of style from structure make our documents easier to

manage, but it also gives us the versatility to completely change the stylistic rendering of a page by swapping out different stylesheets.

Few of us would voluntarily regress back to the days of applying style with HTML elements; yet markup such as the following is still all too common:

```
<button
  type="button"
  onclick="document.getElementById('xyz').style.color='red';">
  Click Me
</button>
```

We can easily see that the style of this button element, including the font of its caption, is not applied via the use of the `` tag and other deprecated style-oriented markup, but is determined by CSS rules in effect on the page. But although this declaration doesn't mix style markup with structure, it does mix *behavior* with structure by including the JavaScript to be executed when the button is clicked as part of the markup of the button element (which in this case turns some Document Object Model [DOM] element named `xyz` red upon a click of the button).

For all the same reasons that it's desirable to segregate style from structure within an HTML document, it's as beneficial (if not more so) to separate the *behavior* from the structure.

This movement is known as *Unobtrusive JavaScript*, and the inventors of jQuery have focused that library on helping page authors easily achieve this separation in their pages. Unobtrusive JavaScript, along with the legions of the jQuery-savvy, considers any JavaScript expressions or statements embedded in the `<body>` of HTML pages, either as attributes of HTML elements (such as `onclick`) or in script blocks placed within the body of the page, to be incorrect.

"But how would I instrument the button without the `onclick` attribute?" you might ask. Consider the following change to the button element:

```
<button type="button" id="testButton">Click Me</button>
```

Much simpler! But now, you'll note, the button doesn't *do* anything.

Rather than embedding the button's behavior in its markup, we'll move it to a script block in the `<head>` section of the page, outside the scope of the document body, as follows:

```
<script type="text/javascript">
  window.onload = function() {
    document.getElementById('testButton').onclick = makeItRed;
  };

  function makeItRed() {
    document.getElementById('xyz').style.color = 'red';
  }
</script>
```

```
}  
</script>
```

We place the script in the `onload` handler for the page to assign a function, `makeItRed()`, to the `onclick` attribute of the button element. We add this script in the `onload` handler (as opposed to inline) because we need to make sure that the button element exists *before* we attempt to manipulate it. (In section 1.3.3 we'll see how jQuery provides a better place for us to put such code.)

If any of the code in this example looks odd to you, fear not! Appendix A provides a look at the JavaScript concepts that you'll need to use jQuery effectively. We'll also be examining, in the remainder of this chapter, how jQuery makes writing the previous code easier, shorter, and more versatile all at the same time.

Unobtrusive JavaScript, though a powerful technique to further add to the clear separation of responsibilities within a web application, doesn't come without its price. You might already have noticed that it took a few more lines of script to accomplish our goal than when we placed it into the button markup. Unobtrusive JavaScript not only may increase the amount of script that needs to be written, but also requires some discipline and the application of good coding patterns to the client-side script.

None of that is bad; anything that persuades us to write our client-side code with the same level of care and respect usually allotted to server-side code is a good thing! But it *is* extra work—without jQuery.

As mentioned earlier, the jQuery team has specifically focused jQuery on the task of making it easy and delightful for us to code our pages using Unobtrusive JavaScript techniques, without paying a hefty price in terms of effort or code bulk in order to do so. We'll find that making effective use of jQuery will enable us to accomplish much more on our pages by writing less code.

Without further ado, let's start taking a look at just how jQuery makes it so easy for us to add rich functionality to our pages without the expected pain.

1.3 jQuery fundamentals

At its core, jQuery focuses on retrieving elements from our HTML pages and performing operations upon them. If you're familiar with CSS, you're already well aware of the power of selectors, which describe groups of elements by their attributes or placement within the document. With jQuery, you'll be able to leverage your knowledge and that degree of power to vastly simplify your JavaScript.

jQuery places a high priority on ensuring our code will work in a consistent manner across all major browsers; many of the more difficult JavaScript problems,

such as waiting until the page is loaded before performing page operations, have been silently solved for us.

Should we find that the library needs a bit more juice, its developers have built in a simple but powerful method for extending its functionality. Many new jQuery programmers find themselves putting this versatility into practice by extending jQuery on their first day.

But first, let's look at how we can leverage our CSS knowledge to produce powerful, yet terse, code.

1.3.1 The jQuery wrapper

When CSS was introduced to web technologies in order to separate design from content, a way was needed to refer to groups of page elements from external style sheets. The method developed was through the use of *selectors*, which concisely represent elements based upon their attributes or position within the HTML document.

For example, the selector

```
p a
```

refers to the group of all links (<a> elements) that are nested inside a <p> element. jQuery makes use of the same selectors, supporting not only the common selectors currently used in CSS, but also the more powerful ones not yet fully implemented by most browsers. The `nth-child` selector from the zebra-stripping code we examined earlier is a good example of a more powerful selector defined in CSS3.

To collect a group of elements, we use the simple syntax

```
$(selector)
```

or

```
jQuery(selector)
```

Although you may find the `$()` notation strange at first, most jQuery users quickly become fond of its brevity.

For example, to retrieve the group of links nested inside a <p> element, we use the following

```
$("p a")
```

The `$()` function (an alias for the `jQuery()` function) returns a special JavaScript object containing an array of the DOM elements that match the selector. This object possesses a large number of useful predefined methods that can act on the group of elements.

In programming parlance, this type of construct is termed a *wrapper* because it wraps the matching element(s) with extended functionality. We'll use the term *jQuery wrapper* or *wrapped set* to refer to this set of matched elements that can be operated on with the methods defined by jQuery.

Let's say that we want to fade out all `<div>` elements with the CSS class `notLongForThisWorld`. The jQuery statement is as follows:

```
$("#div.notLongForThisWorld").fadeOut();
```

A special feature of a large number of these methods, which we often refer to as *jQuery commands*, is that when they're done with their action (like a fading-out operation), they return the same group of elements, ready for another action. For example, say that we want to add a new CSS class, `removed`, to each of the elements in addition to fading them out. We write

```
$("#div.notLongForThisWorld").fadeOut().addClass("removed");
```

These *jQuery chains* can continue indefinitely. It's not uncommon to find examples in the wild of jQuery chains dozens of commands long. And because each function works on all of the elements matched by the original selector, there's no need to loop over the array of elements. It's all done for us behind the scenes!

Even though the selected group of objects is represented as a highly sophisticated JavaScript object, we can pretend it's a typical array of elements, if necessary. As a result, the following two statements produce identical results:

```
$("#someElement").html("I have added some text to an element");
```

or

```
$("#someElement")[0].innerHTML =  
  "I have added some text to an element";
```

Because we've used an ID selector, only one element will match the selector. The first example uses the jQuery method `html()`, which replaces the contents of a DOM element with some HTML markup. The second example uses jQuery to retrieve an array of elements, select the first one using an array index of 0, and replace the contents using an ordinary JavaScript means.

If we want to achieve the same results with a selector that resulted in multiple matched elements, the following two fragments would produce identical results:

```
$("#div.fillMeIn")  
  .html("I have added some text to a group of nodes");
```

or

```
var elements = $("div.fillMeIn");
for(i=0;i<elements.length;i++)
  elements[i].innerHTML =
    "I have added some text to a group of nodes";
```

As things get progressively more complicated, leveraging jQuery's chainability will continue to reduce the lines of code necessary to produce the results that you want. Additionally, jQuery supports not only the selectors that you have already come to know and love, but also more advanced selectors—defined as part of the CSS Specification—and even some custom selectors.

Here are a few examples.

```
$("p:even");
```

This selector selects all even <p> elements.

```
$("tr:nth-child(1)");
```

This selector selects the first row of each table.

```
$("body > div");
```

This selector selects direct <div> children of <body>.

```
$("a[href$=pdf]");
```

This selector selects links to PDF files.

```
$("body > div:has(a)");
```

This selector selects direct <div> children of <body>-containing links.

Powerful stuff!

You'll be able to leverage your existing knowledge of CSS to get up and running fast and then learn about the more advanced selectors jQuery supports. We'll be covering jQuery selectors in great detail in section 2.1, and you can find a full list at <http://docs.jquery.com>Selectors>.

Selecting DOM elements for manipulation is a common need in our pages, but some things that we also need to do don't involve DOM elements at all. Let's take a brief look at more that jQuery offers beyond element manipulation.

1.3.2 Utility functions

Even though wrapping elements to be operated upon is one of the most frequent uses of jQuery's `$()` function, that's not the only duty to which it's assigned. One of its additional duties is to serve as the *namespace prefix* for a handful of general-purpose utility functions. Because so much power is given to page authors by the jQuery wrapper created as a result of a call to `$()` with a selector, it's somewhat rare

for most page authors to need the services provided by some of these functions; we won't be looking at the majority of these functions in detail until chapter 6 as a preparation for writing jQuery plug-ins. But you *will* see a few of these functions put to use in the upcoming sections, so we're introducing their concept here.

The notation for these functions may look odd at first. Let's take, for example, the utility function for trimming strings. A call to it could be

```
$.trim(someString);
```

If the `$.` prefix looks weird to you, remember that `$` is an identifier like any other in JavaScript. Writing a call to the same function using the `jQuery` identifier, rather than the `$` alias, looks a bit more familiar:

```
jQuery.trim(someString);
```

Here it becomes clear that the `trim()` function is merely namespaced by `jQuery` or its `$` alias.

NOTE Even though these elements are called the utility *functions* in jQuery documentation, it's clear that they are actually *methods* of the `$()` function. We'll put aside this technical distinction and use the term *utility function* to describe these methods so as not to introduce conflicting terminology with the online documentation.

We'll explore one of these utility functions that helps us to extend jQuery in section 1.3.5, and one that helps jQuery peacefully coexist with other client-side libraries in section 1.3.6. But first, let's look at another important duty that jQuery's `$` function performs.

1.3.3 The document ready handler

When embracing Unobtrusive JavaScript, behavior is separated from structure, so we'll be performing operations on the page elements outside of the document markup that creates them. In order to achieve this, we need a way to wait until the DOM elements of the page are fully loaded before those operations execute. In the zebra-stripping example, the entire table must load before striping can be applied.

Traditionally, the `onload` handler for the `window` instance is used for this purpose, executing statements after the entire page is fully loaded. The syntax is typically something like

```
window.onload = function() {  
    $("table tr:nth-child(even)").addClass("even");  
};
```

This causes the zebra-stripping code to execute after the document is fully loaded. Unfortunately, the browser not only delays executing the `onload` code until after the DOM tree is created but also waits until after all images and other external resources are fully loaded and the page is displayed in the browser window. As a result, visitors can experience a delay between the time that they first see the page and the time that the `onload` script is executed.

Even worse, if an image or other resource takes a significant time to load, visitors would have to wait for the image loading to complete before the rich behaviors become available. This could make the whole Unobtrusive JavaScript movement a non-starter for many real-life cases.

A much better approach would be to wait *only* until the document structure is fully parsed and the browser has converted the HTML into its DOM tree form before executing the script to apply the rich behaviors. Accomplishing this in a cross-browser manner is somewhat difficult, but jQuery provides a simple means to trigger the execution of code once the DOM tree, but not external image resources, has loaded. The formal syntax to define such code (using our striping example) is as follows:

```
$(document).ready(function() {  
    $("table tr:nth-child(even)").addClass("even");  
});
```

First, we wrap the document instance with the `jQuery()` function, and then we apply the `ready()` method, passing a function to be executed when the document is ready to be manipulated.

We called that the *formal syntax* for a reason; a shorthand form used much more frequently is as follows:

```
$(function() {  
    $("table tr:nth-child(even)").addClass("even");  
});
```

By passing a function to `$()`, we instruct the browser to wait until the DOM has fully loaded (but only the DOM) before executing the code. Even better, we can use this technique multiple times within the same HTML document, and the browser will execute all of the functions we specify in the order that they are declared within the page. In contrast, the window's `onload` technique allows for only a single function. This limitation can also result in hard-to-find bugs if any third-party code we might be using already uses the `onload` mechanism for its own purpose (not a best-practice approach).

We've seen another use of the `$()` function; now let's see yet something else that it can do for us.

1.3.4 Making DOM elements

It's become apparent by this point that the authors of jQuery avoided introducing a bunch of global names into the JavaScript namespace by making the `$()` function (which you'll recall is merely an alias for the `jQuery()` function) versatile enough to perform many duties. Well, there's one more duty that we need to examine.

We can create DOM elements on the fly by passing the `$()` function a string that contains the HTML markup for those elements. For example, we can create a new paragraph element as follows:

```
$("#<p>Hi there!</p>")
```

But creating a disembodied DOM element (or hierarchy of elements) isn't all that useful; usually the element hierarchy created by such a call is then operated on using one of jQuery's DOM manipulation functions.

Let's examine the code of listing 1.1 as an example.

Listing 1.1 Creating HTML elements on the fly

```
<html>
  <head>
    <title>Follow me!</title>
    <script type="text/javascript" src="../scripts/jquery-1.2.js">
    </script>
    <script type="text/javascript">
      $(function(){
        $("#<p>Hi there!</p>").insertAfter("#followMe");
      });
    </script>
  </head>

  <body>
    <p id="followMe">Follow me!</p>
  </body>
</html>
```

1 Ready handler that creates HTML element

2 Existing element to be followed

This example establishes an existing HTML paragraph element named `followMe` **2** in the document body. In the script element within the `<head>` section, we establish a ready handler **1** that uses the following statement to insert a newly created paragraph into the DOM tree after the existing element:

```
$("#<p>Hi there!</p>").insertAfter("#followMe");
```

The result is as shown in figure 1.2.



Figure 1.2
A dynamically created
and inserted element

We'll be investigating the full set of DOM manipulation functions in chapter 2, where you'll see that jQuery provides many means to manipulate the DOM to achieve about any structure that we may desire.

Now that you've seen the basic syntax of jQuery, let's take a look at one of the most powerful features of the library.

1.3.5 Extending jQuery

The jQuery wrapper function provides a large number of useful functions we'll find ourselves using again and again in our pages. But no library can anticipate everyone's needs. It could be argued that no library should even try to anticipate every possible need; doing so could result in a large, clunky mass of code that contains little-used features that merely serve to gum up the works!

The authors of the jQuery library recognized this concept and worked hard to identify the features that most page authors would need and included only those needs in the core library. Recognizing also that page authors would each have their own unique needs, jQuery was designed to be easily extended with additional functionality.

But why extend jQuery versus writing standalone functions to fill in any gaps?

That's an easy one! By extending jQuery, we can use the powerful features it provides, particularly in the area of element selection.

Let's look at a particular example: jQuery doesn't come with a predefined function to disable a group of form elements. And if we're using forms throughout our application, we might find it convenient to be able to use the following syntax:

```
$("#form#myForm input.special").disable();
```

Fortunately, and by design, jQuery makes it easy to extend its set of functions by extending the wrapper returned when we call `$()`. Let's take a look at the basic idiom for how that is accomplished:

```
$.fn.disable = function() {
  return this.each(function() {
    if (typeof this.disabled != "undefined") this.disabled = true;
  });
}
```

A lot of new syntax is introduced here, but don't worry about it too much yet. It'll be old hat by the time you make your way through the next few chapters; it's a basic idiom that you'll use over and over again.

First, `$.fn.disable` means that we're extending the `$` wrapper with a function called `disable`. Inside that function, `this` is the collection of wrapped DOM elements that are to be operated upon.

Then, the `each()` method of this wrapper is called to iterate over each element in the wrapped collection. We'll be exploring this and similar methods in greater detail in chapter 2. Inside of the iterator function passed to `each()`, `this` is a pointer to the specific DOM element for the current iteration. Don't be confused by the fact that `this` resolves to different objects within the nested functions. After writing a few extended functions, it becomes natural to remember.

For each element, we check whether the element has a `disabled` attribute, and if it does, set it to `true`. We return the results of the `each()` method (the wrapper) so that our brand new `disable()` method will support chaining like many of the native jQuery methods. We'll be able to write

```
$("form#myForm input.special").disable().addClass("moreSpecial");
```

From the point of view of our page code, it's as though our new `disable()` method was built into the library itself! This technique is so powerful that most new jQuery users find themselves building small extensions to jQuery almost as soon as they start to use the library.

Moreover, enterprising jQuery users have extended jQuery with sets of useful functions that are known as *plugins*. We'll be talking more about extending jQuery in this way, as well as introducing the official plugins that are freely available in chapter 9.

Before we dive into using jQuery to bring life to our pages, you may be wondering if we're going to be able to use jQuery with Prototype or other libraries that also use the `$` shortcut. The next section reveals the answer to this question.

1.3.6 Using jQuery with other libraries

Even though jQuery provides a set of powerful tools that will meet the majority of the needs for most page authors, there may be times when a page requires that multiple JavaScript libraries be employed. This situation could come about because we're in the process of transitioning an application from a previously employed library to jQuery, or we might want to use both jQuery and another library on our pages.

The jQuery team, clearly revealing their focus on meeting the needs of their user community rather than any desire to lock out other libraries, have made provisions for allowing such cohabitation of other libraries with jQuery on our pages.

First, they've followed best-practice guidelines and have avoided polluting the global namespace with a slew of identifiers that might interfere with not only other libraries, but also names that you might want to use on the page. The identifiers `jQuery` and its alias `$` are the limit of jQuery's incursion into the global namespace. Defining the utility functions that we referred to in section 1.3.2 as part of the `jQuery` namespace is a good example of the care taken in this regard.

Although it's unlikely that any other library would have a good reason to define a global identifier named `jQuery`, there's that convenient but, in this particular case, pesky `$` alias. Other JavaScript libraries, most notably the popular Prototype library, use the `$` name for their own purposes. And because the usage of the `$` name in that library is key to its operation, this creates a serious conflict.

The thoughtful jQuery authors have provided a means to remove this conflict with a utility function appropriately named `noConflict()`. Anytime after the conflicting libraries have been loaded, a call to

```
jQuery.noConflict();
```

will revert the meaning of `$` to that defined by the non-jQuery library.

We'll further cover the nuances of using this utility function in section 7.2.

1.4 Summary

In this whirlwind introduction to jQuery we've covered a great deal of material in preparation for diving into using jQuery to quickly and easily enable Rich Internet Application development.

jQuery is generally useful for any page that needs to perform anything but the most trivial of JavaScript operations, but is also strongly focused on enabling page authors to employ the concept of Unobtrusive JavaScript within their pages. With this approach, behavior is separated from structure in the same way that CSS

separates style from structure, achieving better page organization and increased code versatility.

Despite the fact that jQuery introduces only two new names in the JavaScript namespace—the self-named `jQuery` function and its `$` alias—the library provides a great deal of functionality by making that function highly versatile; adjusting the operation that it performs based upon its parameters.

As we’ve seen, the `jQuery()` function can be used to do the following:

- Select and wrap DOM elements to operate upon
- Serve as a namespace for global utility functions
- Create DOM elements from HTML markup
- Establish code to be executed when the DOM is ready for manipulation

jQuery behaves like a good on-page citizen not only by minimizing its incursion into the global JavaScript namespace, but also by providing an official means to reduce that minimal incursion in circumstances when a name collision might still occur, namely when another library such as Prototype requires use of the `$` name. How’s *that* for being user friendly?

You can obtain the latest version of jQuery from the jQuery site at <http://jquery.com/>. The version of jQuery that the code in this book was tested against (version 1.2.1) is included as part of the downloadable code.

In the chapters that follow, we’ll explore all that jQuery has to offer us as page authors of Rich Internet Applications. We’ll begin our tour in the next chapter as we bring our pages to life via DOM manipulation.



Creating the wrapped element set

This chapter covers

- Selecting elements to be wrapped by jQuery using *selectors*
- Creating and placing new HTML elements in the DOM
- Manipulating the *wrapped element set*

In the previous chapter, we discussed the many ways that the jQuery `$()` function can be used. Its capabilities range from the selection of DOM elements to defining functions to be executed when the DOM is loaded.

In this chapter, we examine in great detail how the DOM elements to be operated upon are identified by looking at two of the most powerful and frequently used capabilities of `$()`: the selection of DOM elements via *selectors* and the creation of new DOM elements.

A good number of the capabilities required by Rich Internet Applications are achieved by manipulating the DOM elements that make up the pages. But before they can be manipulated, they need to be identified and selected. Let's begin our detailed tour of the many ways that jQuery lets us specify what elements are to be targeted for manipulation.

2.1 Selecting elements for manipulation

The first thing we need to do when using virtually any jQuery method (frequently referred to as jQuery *commands*) is to select some page elements to operate upon. Sometimes, the set of elements we want to select will be easy to describe, such as “all paragraph elements on the page.” Other times, they'll require a more complex description like “all list elements that have the class `listElement` and contain a link.”

Fortunately, jQuery provides a robust selector syntax; we'll be able to easily specify virtually any set of elements elegantly and concisely. You probably already know a big chunk of the syntax: jQuery uses the CSS syntax you already know and love, and extends it with some custom methods to select elements that help you perform tasks both common and complex.

To help you learn about element selection, we've put together a Selectors Lab page that's available with the downloadable code examples for this book. If you haven't yet downloaded the example code, now would be a great time to do so. Please see the book's front section for details on how to find and download this code.

This Selectors Lab allows you to enter a jQuery selector string and see (in real time!) which DOM elements get selected. The Selectors Lab can be found at `chapter2/lab.selectors.html` in the example code.

When displayed, the Lab should look as shown in figure 2.1 (if the panes don't appear correctly lined up, you may need to widen your browser window).

The Selector pane at top left contains a text box and a button. To run a Lab experiment, type a selector into the text box and click the Apply button. Go ahead and type the string `li` into the box, and click Apply.

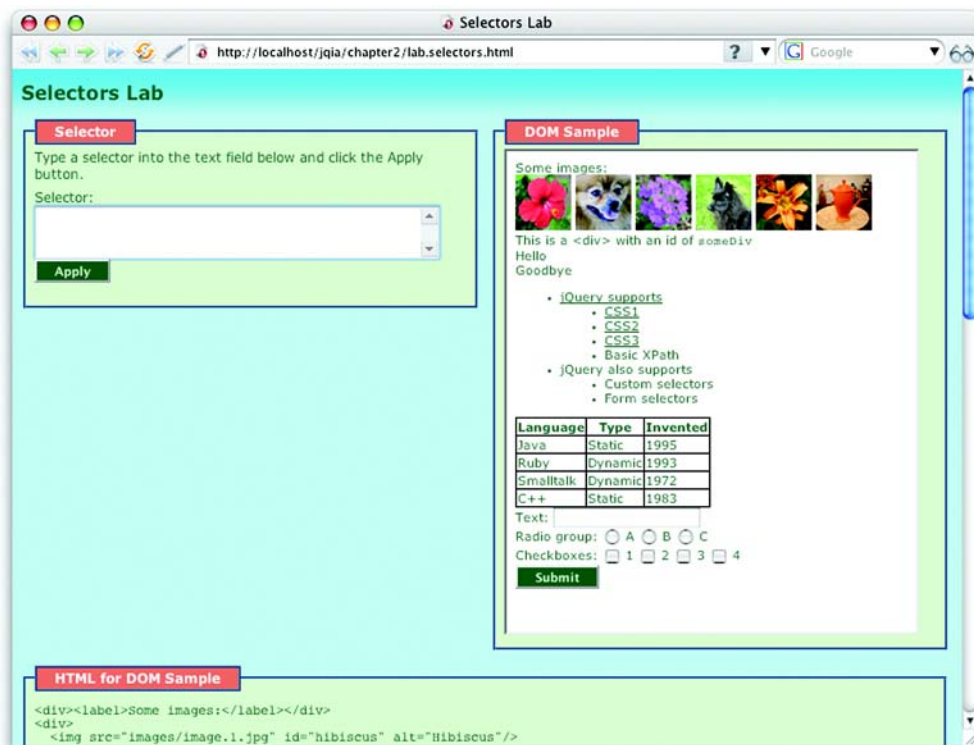


Figure 2.1 The Selectors Lab page allows us to observe the behavior of any selector we choose in real time.

The selector that you type (in this case `li`) is applied to the HTML page loaded into an `<iframe>` in the DOM Sample pane at upper right. The jQuery code on the sample page causes all matched elements to be highlighted with a red border. After clicking Apply, you should see the display shown in figure 2.2 in which all `` elements in the page are highlighted.

Note that the `` elements in the sample page have been outlined and that the executed jQuery statement, along with the tag names of the selected elements, has been displayed below the Selector text box.

The HTML markup used to render the DOM Sample page is displayed in the lower pane labeled HTML for DOM Sample to help you experiment with writing selectors.

We'll talk more about using this Lab as we progress through the chapter. But first, let's wade into familiar territory: traditional CSS selectors.

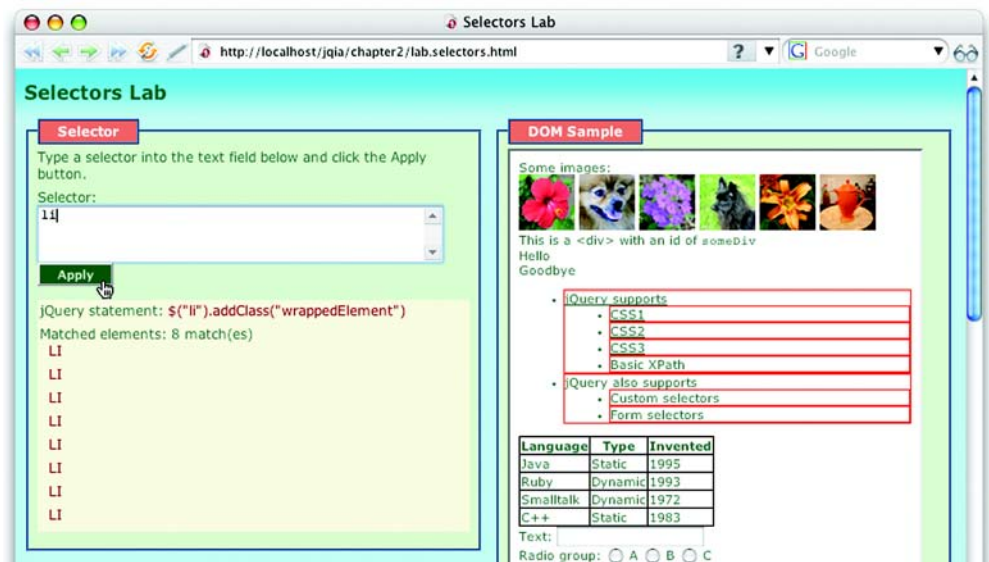


Figure 2.2 A selector value of `li` matches all `` elements when applied as shown by the display results.

2.1.1 Using basic CSS selectors

For applying styles to page elements, web developers have become familiar with a small, but powerful and useful, group of selection methods that work across all browsers. Those methods include selection by an element's ID, CSS class name, tag name, and the DOM hierarchy of the page elements.

Here are some examples to give you a quick refresher.

- `a`—This selector matches all link (`<a>`) elements.
- `#specialID`—This selector matches elements that have an id of `specialID`.
- `.specialClass`—This selector matches elements that have the class of `specialClass`.
- `a#specialID.specialClass`—This selector matches links with an id of `specialID` and a class of `specialClass`.
- `p a.specialClass`—This selector matches links with a class of `specialClass` declared within `<p>` elements.

We can mix and match the basic selector types to select fairly fine-grained sets of elements. In fact, the most fancy and creative websites use some combination of these basic options to create their dazzling displays.

We can use jQuery out of the box with the CSS selectors that we're already accustomed to using. To select elements using jQuery, we wrap the selector in `$()`, as in

```
$("#p a.specialClass")
```

With a few exceptions, jQuery is fully CSS3 compliant, so selecting elements this way will come with no surprises; the same elements that would be selected in a style sheet by a standards-compliant browser will be selected by jQuery's selector engine. Note that jQuery doesn't depend upon the CSS implementation of the browser it's running within. Even if the browser doesn't implement a standard CSS selector correctly, jQuery will correctly select elements according to the rules of the World Wide Web Consortium (W3C) standard.

For some exercise, go play with the Selectors Lab and run some experiments with the various basic CSS selectors.

These basic selectors are powerful, but sometimes we need even finer-grained control over which elements we want to match. jQuery meets this challenge and steps up to the plate with even more advanced selectors.

2.1.2 Using child, container, and attribute selectors

For more advanced selectors, jQuery uses the next generation of CSS supported by Mozilla Firefox, Internet Explorer 7, Safari, and other modern browsers. These advanced selectors include selecting the direct children of some elements, elements that occur after other elements in the DOM, and elements with attributes matching certain conditions.

Sometimes, we'll want to select only the direct children of a certain element. For example, we might want to select list elements directly under some list, but not list elements belonging to a sublist. Consider the following HTML fragment from the sample DOM of the Selectors Lab:

```
<ul class="myList">
  <li><a href="http://jquery.com">jQuery supports</a>
    <ul>
      <li><a href="css1">CSS1</a></li>
      <li><a href="css2">CSS2</a></li>
      <li><a href="css3">CSS3</a></li>
      <li>Basic XPath</li>
    </ul>
  </li>
  <li>jQuery also supports
    <ul>
```

```

    <li>Custom selectors</li>
    <li>Form selectors</li>
  </ul>
</li>
</ul>

```

Suppose we want to select the link to the remote jQuery site, but not the links to various local pages describing the different CSS specifications. Using basic CSS selectors, we might try something like `ul.myList li a`. Unfortunately, that selector would grab all links because they all descend from a list element.

You can verify this by entering the selector `ul.myList li a` into the Selectors Lab and clicking Apply. The results will be as shown in figure 2.3.

A more advanced approach is to use *child selectors*, in which a parent and its direct child are separated by the right angle bracket character (`>`), as in

```
p > a
```

This selector matches only links that are *direct* children of a `<p>` element. If a link were further embedded, say within a `` within the `<p>`, that link would not be selected.

Going back to our example, consider a selector such as

```
ul.myList > li > a
```

This selector selects only links that are direct children of list elements, which are in turn direct children of `` elements that have the class `myList`. The links contained in the sublists are excluded because the `` elements serving as the

The screenshot shows the Selectors Lab interface. On the left, the 'Selector' field contains `ul.myList li a` and the 'Apply' button is clicked. Below, the jQuery statement is shown as `$("#ul.myList li a").addClass("wrappedElement")` and the result shows 4 matched elements, all highlighted in red. On the right, the 'DOM Sample' shows a list of images and a list of links. The selected elements are highlighted in red.

Some images:

This is a `<div>` with an id of `someDiv`

Hello

Goodbye

- jQuery supports
 - CSS1
 - CSS2
 - CSS3
 - Basic XPath
- jQuery also supports
 - Custom selectors
 - Form selectors

Language	Type	Invented
Java	Static	1995
Ruby	Dynamic	1993
Smalltalk	Dynamic	1972

Figure 2.3 All anchor tags that are descendants, at any depth, of an `` element are selected by `ul.myList li a`.

parent of the sublists `` elements don't have the class `myList`, as shown in the Lab results of figure 2.4.

Attribute selectors are also extremely powerful. Say we want to attach a special behavior only to links that point to locations outside our sites. Let's take another look at a portion of the Lab example that we previously examined:

```
<li><a href="http://jquery.com">jQuery supports</a>
  <ul>
    <li><a href="css1">CSS1</a></li>
    <li><a href="css2">CSS2</a></li>
    <li><a href="css3">CSS3</a></li>
    <li>Basic XPath</li>
  </ul>
</li>
```

What makes the link pointing to an external site unique is the presence of the string `http://` at the beginning of the value of the link's `href` attribute. We could select links with an `href` value starting with `http://` with the following selector:

```
a[href^=http://]
```

This matches all links with an `href` value beginning with exactly `http://`. The caret character (`^`) is used to specify that the match is to occur at the beginning of a value. This is the same character used by most regular expression processors to signify matching at the beginning of a candidate string; it should be easy to remember.

Visit the Lab page, from which the example HTML fragment was lifted, type `a[href^=http://]` into the text box, and click Apply. Note how only the jQuery link is highlighted.

There are other ways to use attribute selectors. To match an element that possesses a specific attribute, regardless of its value, we can use

```
form[method]
```



Figure 2.4 With the selector `ul.myList > li > a`, only the direct children of parent nodes are matched.

This matches any `<form>` element that has an explicit `method` attribute.

To match a specific attribute value, we use something like

```
input[type=text]
```

This selector matches all `input` elements with a `type` of `text`.

We've already seen the "match attribute at beginning" selector in action. Here's another:

```
div[title^=my]
```

This selects all `<div>` elements with `title` attributes whose value begins with `my`.

What about an "attribute ends with" selector? Coming right up:

```
a[href$=.pdf]
```

This is a useful selector for locating all links that reference PDF files.

And there's a selector for locating elements whose attributes contain arbitrary strings anywhere in the attribute value:

```
a[href*=jquery.com]
```

As we would expect, this selector matches all `<a>` elements that reference the jQuery site.

Beyond attributes, we'll sometimes want to select an element only if it contains some other element. In the previous list example, suppose we want to apply some behavior to list elements containing links. jQuery supports this kind of selection with the *container selector*:

```
li:has(a)
```

This selector matches all `` elements that contain an `<a>` element. Note that this is *not* the same as a selector of `li a`, which matches all `<a>` elements contained within `` elements. Use the Selectors Lab page to convince yourself of the difference between these two forms.

Table 2.1 shows the CSS selectors that we can use with jQuery.

Be aware that only a single level of nesting is supported. Although it's possible to nest *one* level, such as

```
foo:not(bar:has(baz))
```

additional levels of nesting, such as

```
foo:not(bar:has(baz:eq(2)))
```

aren't supported.

Table 2.1 The basic CSS Selectors supported by jQuery

Selector	Description
*	Matches any element.
E	Matches all element with tag name E.
E F	Matches all elements with tag name F that are descendents of E.
E>F	Matches all elements with tag name F that are direct children of E.
E+F	Matches all elements F immediately preceded by sibling E.
E~F	Matches all elements F preceded by any sibling E.
E:has(F)	Matches all elements with tag name E that have at least one descendent with tag name F.
E.C	Matches all elements E with class name C. Omitting E is the same as *.C.
E#I	Matches element E with id of I. Omitting E is the same as *#I.
E[A]	Matches all elements E with attribute A of any value.
E[A=V]	Matches all elements E with attribute A whose value is exactly V.
E[A^=V]	Matches all elements E with attribute A whose value begins with V.
E[A\$=V]	Matches all elements E with attribute A whose value ends with V.
E[A*=V]	Matches all elements E with attribute A whose value contains V.

With all this knowledge in hand, head over to the [Selectors Lab](#) page, and spend some more time running experiments using selectors of various types from table 2.1. Try to make some targeted selections like the `` elements containing the text *Hello* and *Goodbye* (hint: you'll need to use a combination of selectors to get the job done).

As if the power of the selectors that we've discussed so far isn't enough, there are some more options that give us an even finer ability to slice and dice the page.

2.1.3 Selecting by position

Sometimes, we'll need to select elements by their position on the page or in relation to other elements. We might want to select the first link on the page, or every other paragraph, or the last list item of each list. jQuery supports mechanisms for achieving these specific selections.

For example, consider

```
a:first
```

This format of selector matches the first `<a>` element on the page.

What about picking every other element?

```
p:odd
```

This selector matches every odd paragraph element. As we might expect, we can also specify that evenly ordered elements be selected with

```
p:even
```

Another form

```
li:last-child
```

chooses the last child of parent elements. In this example, the last `` child of each `` element is matched.

There are a whole slew of these selectors, and they can provide surprisingly elegant solutions to sometimes tough problems. See table 2.2 for a list of these positional selectors.

Table 2.2 The more advanced positional selectors supported by jQuery: selecting elements based on their position in the DOM

Selector	Description
<code>:first</code>	The first match of the page. <code>li a:first</code> returns the first link also under a list item.
<code>:last</code>	The last match of the page. <code>li a:last</code> returns the last link also under a list item.
<code>:first-child</code>	The first child element. <code>li:first-child</code> returns the first item of each list.
<code>:last-child</code>	The last child element. <code>li:last-child</code> returns the last item of each list.
<code>:only-child</code>	Returns all elements that have no siblings.
<code>:nth-child(<i>n</i>)</code>	The <i>n</i> th child element. <code>li:nth-child(2)</code> returns the second list item of each list.
<code>:nth-child(even odd)</code>	Even or odd children. <code>li:nth-child(even)</code> returns the even children of each list.

continued on next page

Table 2.2 The more advanced positional selectors supported by jQuery: selecting elements based on their position in the DOM (continued)

Selector	Description
<code>:nth-child(Xn+Y)</code>	The <i>n</i> th child element computed by the supplied formula. If <i>Y</i> is 0, it may be omitted. <code>li:nth-child(3n)</code> returns every third item, whereas <code>li:nth-child(5n+1)</code> returns the item after every fifth element.
<code>:even</code> and <code>:odd</code>	Even and odd matching elements page-wide. <code>li:even</code> returns every even list item.
<code>:eq(<i>n</i>)</code>	The <i>n</i> th matching element.
<code>:gt(<i>n</i>)</code>	Matching elements after (and excluding) the <i>n</i> th matching element.
<code>:lt(<i>n</i>)</code>	Matching elements before (and excluding) the <i>n</i> th matching element.

There is one quick gotcha (isn't there always?). The `nth-child` selector starts counting from 1, whereas the other selectors start counting from 0. For CSS compatibility, `nth-child` starts with 1, but the jQuery custom selectors follow the more common programming convention of starting at 0. With some use, it becomes second nature to remember which is which, but it may be a bit confusing at first.

Let's dig in some more.

Consider the following table, containing a list of some programming languages and some basic information regarding them:

```
<table id="languages">
  <thead>
    <tr>
      <th>Language</th>
      <th>Type</th>
      <th>Invented</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Java</td>
      <td>Static</td>
      <td>1995</td>
    </tr>
    <tr>
      <td>Ruby</td>
      <td>Dynamic</td>
      <td>1993</td>
    </tr>
    <tr>
      <td>Smalltalk</td>
```

```
        <td>Dynamic</td>
        <td>1972</td>
    </tr>
    <tr>
        <td>C++</td>
        <td>Static</td>
        <td>1983</td>
    </tr>
</tbody>
</table>
```

Let's say we want to get all of the table cells that contained the names of programming languages. Because they are all the first cells in their row, we can use

```
table#languages tbody td:first-child
```

We can also easily use

```
table#languages tbody td:nth-child(1)
```

But the first syntax would be considered pithier and more elegant.

To grab the language type cells, we change the selector to use `:nth-child(2)`, and for the year they were invented, we use `:nth-child(3)` or `:last-child`. If we want the absolute last table cell (the one containing the text *1983*), we'd use `td:last`. Also, whereas `td:eq(2)` returns the cell containing the text *1995*, `td:nth-child(2)` returns all of the cells giving programming language types. Again, remember that `:eq` is 0-based, but `:nth-child` is 1-based.

Before we move on, head back over to the Selectors Lab, and try selecting entries two and four from the list. Then, try to find three different ways to select the cell containing the text *1972* in the table. Also, try and get a feel for the difference between the `nth-child` selectors and the absolute position selectors.

Even though the CSS selectors we've examined so far are incredibly powerful, let's discuss ways of squeezing even more power out of jQuery's selectors.

2.1.4 Using custom jQuery selectors

The CSS selectors give us a great deal of power and flexibility to match the desired DOM elements, but sometimes we'll want to select elements based on a characteristic that the CSS specification did not anticipate.

For example, we might want to select all check boxes that have been checked by the user. Because trying to match by attribute will only check the initial state of the control as specified in the HTML markup, jQuery offers a custom selector, `:checked`, that filters the set of matched elements to those that are in checked state. For example, whereas the `input` selector selects all `<input>` elements, the

`input:checked` narrows the search to only `<input>` elements that are checked. The custom `:checked` selector works like a CSS attribute selector (such as `[foo=bar]`) in that both filter the matching set of elements by some criteria. Combining these custom selectors can be powerful; consider `:radio:checked` and `:checkbox:checked`.

As we discussed earlier, jQuery supports all of the CSS filter selectors and also a number of custom selectors defined by jQuery. They are described in table 2.3.

Table 2.3 The jQuery custom filter selectors that give immense power to identify target elements

Selector	Description
<code>:animated</code>	Selects elements that are currently under animated control. Chapter 5 will cover animations and effects.
<code>:button</code>	Selects any button (<code>input[type=submit]</code> , <code>input[type=reset]</code> , <code>input[type=button]</code> , or <code>button</code>).
<code>:checkbox</code>	Selects only check box elements (<code>input[type=checkbox]</code>).
<code>:checked</code>	Selects only check boxes or radio buttons that are checked (supported by CSS).
<code>:contains(foo)</code>	Selects only elements containing the text <i>foo</i> .
<code>:disabled</code>	Selects only form elements that are disabled in the interface (supported by CSS).
<code>:enabled</code>	Selects only form elements that are enabled in the interface (supported by CSS).
<code>:file</code>	Selects all file elements (<code>input[type=file]</code>).
<code>:header</code>	Selects only elements that are headers; for example: <code><h1></code> through <code><h6></code> elements.
<code>:hidden</code>	Selects only elements that are hidden.
<code>:image</code>	Selects form images (<code>input[type=image]</code>).
<code>:input</code>	Selects only form elements (<code>input</code> , <code>select</code> , <code>textarea</code> , <code>button</code>).
<code>:not(filter)</code>	Negates the specified filter.
<code>:parent</code>	Selects only elements that have children (including text), but not empty elements.
<code>:password</code>	Selects only password elements (<code>input[type=password]</code>).
<code>:radio</code>	Selects only radio elements (<code>input[type=radio]</code>).
<code>:reset</code>	Selects reset buttons (<code>input[type=reset]</code> or <code>button[type=reset]</code>).

continued on next page

Table 2.3 The jQuery custom filter selectors that give immense power to identify target elements (*continued*)

Selector	Description
<code>:selected</code>	Selects option elements that are selected.
<code>:submit</code>	Selects submit buttons (<code>button[type=submit]</code> or <code>input[type=submit]</code>).
<code>:text</code>	Selects only text elements (<code>input[type=text]</code>).
<code>:visible</code>	Selects only elements that are visible.

Many of the custom jQuery selectors are form-related, allowing us to specify, rather elegantly, a specific element type or state. We can combine selector filters too. For example, if we want to select only enabled and checked check boxes, we could use

```
:checkbox:checked:enabled
```

Try out as many of these filters as you like in the Selectors Lab until you feel that you have a good grasp of their operation.

These filters are an immensely useful addition to the set of selectors at our disposal, but what about the *inverse* of these filters?

Using the `:not` filter

If we want to negate a filter, let's say to match any input element that's *not* a check box, we use the `:not` filter, which is supported for CSS filters and works with custom jQuery selector filters too.

To select non-check box `<input>` elements, we use

```
input:not(:checkbox)
```

It's important to recognize the distinction between *filter* selectors, which attenuate a matching set of elements by applying a further selection criteria to them (like the ones shown previously), and *find* selectors. Find selectors, such as the descendent selector (space character), the child selector (`>`), and the sibling selector (`+`), find *other* elements that bear some relationship to the ones already selected, rather than limiting the scope of the match with criteria applied to the matched elements.

We can apply the `:not` filter to filter selectors, but not to find selectors. The selector

```
div p:not(:hidden)
```

is a perfectly valid selector, but `div :not(p:hidden)` isn't.

In the first case, all `<p>` elements descending from a `<div>` element that aren't hidden are selected. The second selector is illegal because it attempts to apply `:not` to a selector that isn't a filter (the `p` in `p:hidden` isn't a filter).

To make things simpler, filter selectors are easily identified because they all begin with a colon character (`:`) or a square bracket character (`[]`). Any other selector can't be used inside the `:not()` filter.

As we've seen, jQuery gives us a large toolset with which to select existing elements on a page for manipulation via the jQuery methods, which we'll examine in chapter 3. But before we look at the manipulation methods, let's see how to use the `$()` function to create new HTML elements to include in matched sets.

“But wait!” as they say, “there’s more!”

We've emphasized, and will continue to emphasize, that part of jQuery's strength is the ease with which it allows extensions via plugins. If you're familiar with using XML Path Language (XPath) to select elements within an Extensible Markup Language (XML) document, you're in luck. A jQuery plugin provides some basic XPath support that can be used together with jQuery's excellent CSS and custom selectors. Look for this plugin at <http://jquery.com/plugins/project/xpath>.

Keep in mind that the support for XPath is basic, but it should be enough (in combination with everything else we can do with jQuery) to make some powerful selections possible.

First, the plugin supports the typical `/` and `//` selectors. For example, `/html//form/fieldset` selects all `<fieldset>` elements that are directly under a `<form>` element on the page.

We can also use the `*` selector to represent any element, as in `/html//form/*/input`, which selects all `<input>` elements directly under exactly one element that's under a `<form>` element.

The XPath plugin also supports the parent selector `..`, which selects parents of previous element selectors. For example: `//div/..` matches all elements that are directly parent to a `<div>` element.

Also supported are XPath attribute selectors (`//div[@foo=bar]`), as well as container selectors (`//div[@p]`, which selects `<div>` elements containing at least one `<p>` element). The plugin also supports `position()` via the jQuery position selectors described earlier. For instance, `position()=0` becomes `:first`, and `position()>n` becomes `:gt(n)`.

2.2 Generating new HTML

Sometimes, we'll want to generate new fragments of HTML to insert into the page. With jQuery, it's a simple matter because, as we saw in chapter 1, the `$` function can create HTML in addition to selecting existing page elements. Consider

```
$("<div>Hello</div>")
```

This expression creates a new `<div>` element ready to be added to the page. We can run any jQuery commands that we could run on wrapped element sets of existing elements on the newly created fragment. This may not seem impressive on first glance, but when we throw event handlers, Ajax, and effects into the mix (as we will in the upcoming chapters), we'll see how it can come in handy.

Note that if we want to create an empty `<div>` element, we can get away with a shortcut:

```
$("<div>") ← Identical to $("<div> </div>")  
and $("<div/>")
```

As with many things in life, there is a small caveat: we won't be able to use this technique to reliably create `<script>` elements. But there are plenty of techniques for handling the situations that would normally cause us to want to build `<script>` elements in the first place.

To give you a taste of what you'll be able to do later (don't worry if some of it goes over your head at this point), take a look at this:

```
$("<div class='foo'>I have foo!</div><div>I don't</div>")
  .filter(".foo").click(function() {
    alert("I'm foo!");
  }).end().appendTo("#someParentDiv");
```

In this snippet, we first create two `<div>` elements, one with class `foo` and one without. We then narrow down the selection to only the `<div>` with class `foo` and bind an event handler to it that will fire an alert dialog box when clicked. Finally, we use the `end()` method (see section 2.3.6) to revert back to the full set of both `<div>` elements and attach them to the DOM tree by appending them to the element with the id of `someParentDiv`.

That's a lot going on in a single statement, but it certainly shows how we can get a lot accomplished without writing a lot of script.

An HTML page that runs this example is provided in the downloaded code as `chapter2/new.divs.html`. Loading this file into a browser results in the displays shown in figure 2.5.

On initial load, as seen in the upper portion of figure 2.5, the new `<div>` elements are created and added to the DOM tree (because we placed the example

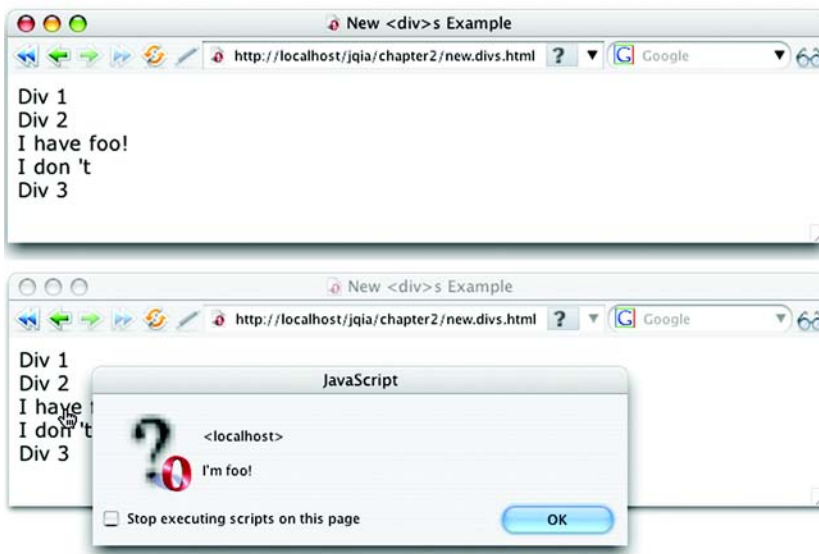


Figure 2.5 New HTML elements can be created under script control and given advanced attributes, such as event handlers, all in a single jQuery statement.

snippet into the page's ready handler) right after the element containing text *Div 2* (which has the id of *someParentDiv*). The lower portion of the figure shows that the defined event handler is triggered when the first newly-created `<div>` is clicked.

Don't be too worried that we haven't covered much of what you may need to fully understand the previous example; we'll get to all of it soon enough. In fact, let's get right to manipulating the wrapped set, including the `filter()` command we used in the example.

2.3 Managing the wrapped element set

Once we've got the set of wrapped elements that we either identified by using a selector to match existing DOM elements or created as new elements using HTML snippets (or a combination of both), we're ready to manipulate those elements using the powerful set of jQuery commands. We'll start looking at those commands in the next chapter; but what if we're not quite ready yet? What if we want to *further* refine the set of elements wrapped by the jQuery function?

In this section, we'll explore the many ways that we can refine, extend, or subset the set of wrapped elements that we wish to operate upon.

In order to visually help you in this endeavor, another lab page has been set up and included in the downloadable example code for this chapter: the Wrapped Set Lab, which you will find in `chapter2/lab.wrapped.set.html`. This page, which looks a lot like the Selectors Lab we examined earlier in this chapter, is shown in figure 2.6.

This new lab page not only looks like the Selectors Lab, it also operates in a similar fashion. Except in this Lab, rather than typing a selector, we can type in a complete jQuery wrapped set *operation*. The operation is applied to the DOM Sample, and, as with the Selectors Lab, the results are displayed.

In this sense, the Wrapped Set Lab is a more general case of the Selectors Lab. Whereas the latter only allowed us to enter a single selector, the Wrapped Set Lab allows us to enter any expression that results in a jQuery wrapped set. Because of the way jQuery chaining works, this expression can also include commands, making this a powerful Lab for examining the operations of jQuery. Be

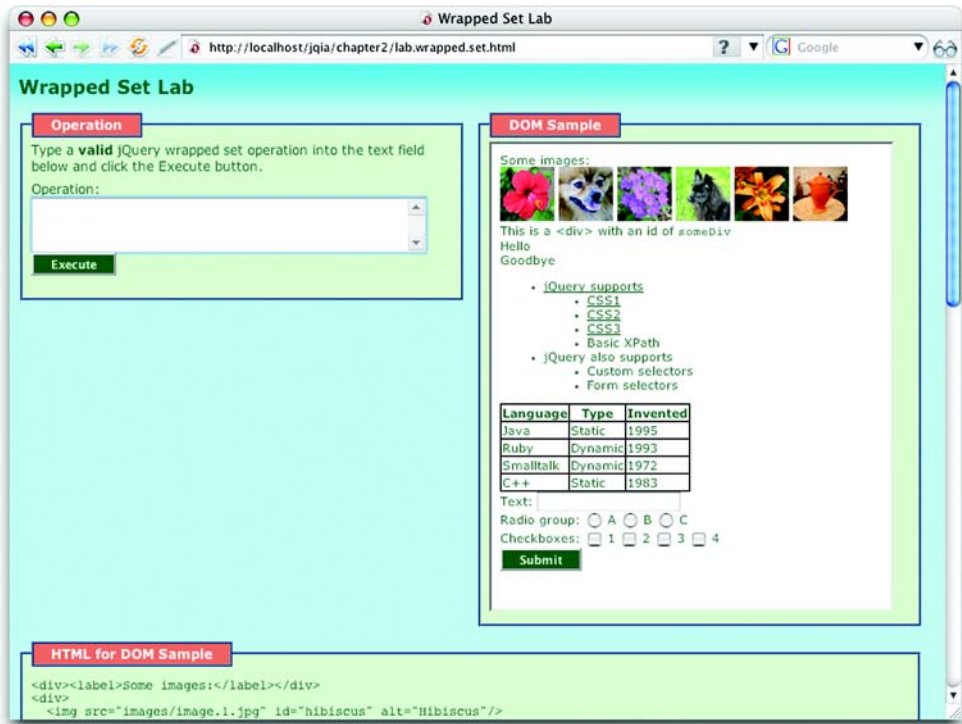


Figure 2.6 The Wrapped Set Lab helps us see how wrapped sets can be created and managed.

aware that you need to enter valid syntax, as well as expressions that result in a jQuery wrapped set. Otherwise, you're going to be faced with a handful of unhelpful JavaScript errors.

We'll see this new Lab in action as we work our way through the sections that follow.

2.3.1 Determining the size of the wrapped set

We mentioned before that the set of jQuery wrapped elements acts a lot like an array. This mimicry includes a `length` property, like JavaScript arrays, that contains the number of wrapped elements.

Should we wish to use a method rather than a property, jQuery also defines the `size()` method, which returns the same information.

Consider the following statement:

```
$('#someDiv')  
  .html('There are '+$('a').size()+' link(s) on this page.');
```

The inner jQuery wrapper matches all elements of type `<a>` and returns the number of matched elements using the `size()` method. This is used to construct a text string, which is set as the content of an element with `id` of `someDiv` using the `html()` method (which we'll see in the next chapter).

The formal syntax of the `size()` command is as follows:

Command syntax: `size`

size()

Returns the count of elements in the wrapped set

Parameters

none

Returns

The element count

OK, so now you know how many elements you have. What if you want to access them directly?

2.3.2 Obtaining elements from the wrapped set

Usually, once we have a wrapped set of elements, we can use jQuery commands to perform some sort of operation upon them; for example, hiding them all with the `hide()` method. But there may be times when we want to get our hands on a

direct reference to an element or elements to perform raw JavaScript operations upon them.

Because jQuery allows us to treat the wrapped set as a JavaScript array, we can use simple array indexing to obtain any element in the wrapped list by position. For example, to obtain the first element in the set of all `` elements with an `alt` attribute on the page, we can write

```
$('#img[alt]')[0]
```

If we prefer to use a method rather than array indexing, jQuery defines the `get()` method for that purpose.

Command syntax: `get`

`get(index)`

Obtains one or all of the matched elements in the wrapped set. If no parameter is specified, all elements in the wrapped set are returned in a JavaScript array. If an `index` parameter is provided, the indexed element is returned.

Parameters

`index` (Number) The index of the single element to return. If omitted, the entire set is returned in an array.

Returns

A DOM element or an array of DOM elements.

The fragment

```
$('#img[alt]').get(0)
```

is equivalent to the previous example that used array indexing.

The `get()` method can also be used to obtain a plain JavaScript array of all the wrapped elements. Consider:

```
var allLabeledButtons = $('#label+button').get();
```

This statement wraps all the `<button>` elements on a page that are immediately preceded by `<label>` elements in a jQuery wrapper and then creates a JavaScript array of those elements to assign to the `allLabeledButtons` variable.

We can use an inverse operation to find the index of a particular element in the wrapped set. Let's say for some reason we want to know the ordinal index of an image with the `id` of `findMe` within the entire set of images in a page. We can obtain this value with

```
var n = $('#img').index($('#img#findMe')[0]);
```


The syntax of the `index()` command is as follows:

Command syntax: `index`

`index(element)`

Finds the passed element in the wrapped set and returns its ordinal index within the set. If the element isn't resident in the set, the value -1 is returned.

Parameters

`element` (Element) A reference to the element whose ordinal value is to be determined.

Returns

The ordinal value of the passed element within the wrapped set or -1 if not found.

Now, rather than obtaining elements, how would you go about adjusting the set of elements that are wrapped?

2.3.3 *Slicing and dicing the wrapped element set*

Once we have a wrapped element set, we may want to augment that set by adding to it or by reducing the set to a subset of the originally matched elements. jQuery gives us a large collection of methods to manage the set of wrapped elements. First, let's look at adding elements to a wrapped set.

Adding more elements to the wrapped set

Often, we may find ourselves in a situation where we want to add more elements to an existing wrapped set. This capability is most useful when we want to add more elements after applying some command to the original set. Remember, jQuery chaining makes it possible to perform an enormous amount of work in a single statement.

But first, let's examine a simple situation. Let's say that we want to match all `` elements that have either an `alt` or a `title` attribute. The powerful jQuery selectors allow us to express this as a single selector, such as

```
$('.img[alt],img[title]')
```

But to illustrate the operation of the `add()` method, we could match the same set of elements with

```
$('.img[alt]').add('img[title]')
```

Using the `add()` method in this fashion allows us to chain a bunch of selectors together into an *or* relationship, creating the union of the elements that satisfy both of the selectors. Methods such as `add()` can also be useful in place of selectors

in that the `end()` method (which we'll examine in section 2.3.6) can be used to back out of the elements added via `add()`.

Command syntax: add

add(expression)

Adds elements, specified by the `expression` parameter, to the wrapped set. The expression can be a selector, an HTML fragment, a DOM element, or an array of DOM elements.

Parameters

`expression` (String|Element|Array) Specifies what is to be added to the matched set. This parameter can be a jQuery selector, in which case any matched elements are added to the set. If an HTML fragment, the appropriate elements are created and added to the set. If a DOM element or an array of DOM elements, they are added to the set.

Returns

The wrapped set.

Bring up the Wrapped Set Lab page in your browser, enter the previous example (exactly as shown), and click the Execute button. This should execute the jQuery operation and result in the selection of all images with either an `alt` or `title` attribute.

Inspecting the HTML source for the DOM Sample reveals that all the images depicting flowers have `alt` attributes, the puppy images have `title` attributes, and the coffee pot image has neither. Therefore, we should expect that all images but the coffee pot would become part of the wrapped set. Figure 2.7 shows a screen capture of the relevant page portions of the results.

The screenshot shows two panels. The left panel, titled 'Operation', contains a text area with the jQuery selector `$('img[alt]').add('img[title]')` and an 'Execute' button. Below the button, it displays 'Wrapped set elements: 5 element(s)' followed by a list of image IDs: `IMG#hibiscus`, `IMG#verbena`, `IMG#tigerLily`, `IMG#littleBear`, and `IMG#cozmo`. The right panel, titled 'DOM Sample', shows a row of six images. The first five images (hibiscus, puppy, verbena, puppy, tiger lily) are highlighted in red, indicating they were selected by the jQuery operation. Below the images is text: 'Some images:', 'This is a <div> with an id of someDiv', 'Hello', and 'Goodbye'. There is also a list of items: 'jQuery supports' (CSS1, CSS2, CSS3, Basic XPath) and 'jQuery also supports' (Custom selectors, Form selectors). At the bottom, there is a table:

Language	Type	Invented
Java	Static	1995
Perl	Interpreted	1987

Figure 2.7 The expected image elements, those with an `alt` or `title` attribute, have been matched by the jQuery expression.

We can see that five of the six images (all but the coffee pot) were added to the wrapped set. (The red outline may be a bit hard to see in the print version of this book with grayscale figures.)

Now let's take a look at a more realistic use of the `add()` method. Let's say that we want to apply a thick border to all `` elements with `alt` attributes, and then apply a level of transparency to all `` elements with either `alt` or `title` attributes. The comma operator (`,`) of CSS selectors won't help us with this one because we want to apply an operation to a wrapped set and *then* add more elements to it. We could easily accomplish this with multiple statements, but it would be more efficient and elegant to use the power of jQuery chaining to accomplish the task in a single statement, such as

```
$('.img[alt]').addClass('thickBorder').add('.img[title]')
  ➔ .addClass('seeThrough')
```

In this statement, we create a wrapped set of all `` elements with `alt` attributes, apply a predefined class that applies a thick border, add the `` elements that have `title` attributes, and finally apply a class that applies transparency to the newly augmented set.

Enter this statement into the Wrapped Set Lab page (which has predefined the named classes), and view the results as shown in figure 2.8.

In these results, we can see that the flower images (those with `alt`) have thick borders, and all images but the coffee pot (the only one with neither an `alt` nor a `title`) are faded as a result of applying an opacity rule.

The `add()` method can also be used to add elements to an existing wrapped set given references to those elements. Passing an element reference, or an array of element references, to the `add()` method adds the elements to the wrapped set. If we assume that we have an element reference in a variable named `someElement`, it could be added to the set of all images containing an `alt` property with

```
$('.img[alt]').add(someElement)
```



Figure 2.8
jQuery chaining allows us to perform complex operations in a single statement, as seen by these results.

As if that weren't flexible enough, the `add()` method not only allows us to add existing elements to the wrapped set, we can also use it to add new elements by passing it a string containing HTML markup. Consider

```
$('#p').add('<div>Hi there!</div>')
```

This fragment creates a wrapped set of all `<p>` elements in the document, and then creates a new `<div>`, and adds it to the wrapped set. Note that doing so only adds the new element to the wrapped set; no action has been taken in this statement to add the new element to the DOM. We might then use the jQuery `append()` method (patience, we'll be talking about such methods soon enough) to append the elements we selected, as well as the newly created HTML, to some part of the DOM.

Augmenting the wrapped set with `add()` is easy and powerful, but now let's look at the jQuery methods that let us remove elements from a wrapped set.

Honing the contents of the wrapped set

We saw that it's a simple matter in jQuery to create wrapped sets from multiple selectors chained together with the `add()` method to form an *or* relationship. It's also possible to chain selectors together to form an *except* relationship by employing the `not()` method. This is similar to the `:not` selector filter we discussed earlier, but can be employed in a similar fashion to the `add()` method to remove elements from the wrapped set anywhere within a jQuery chain of commands.

Let's say that we want to select all `` elements in a page that sport a `title` attribute *except* for those that contain the text *puppy* in the `title` attribute value. We could come up with a single selector that expresses this condition (namely `img[title]:not([title*=puppy])`), but for the sake of illustration, let's pretend that we forgot about the `:not` filter. By using the `not()` method, which removes any elements from a wrapped set that match the passed selector expression, we can express an *except* type of relationship. To perform the described match, we can write

```
$('#img[title]').not('[title*=puppy]')
```

Type this expression into the Wrapped Set Lab page, and execute it. You'll see that only the tan puppy image is selected. The black puppy, which is included in the original wrapped set because it possesses a `title` attribute, is removed by the `not()` invocation because its `title` contains the text *puppy*.

Note that the selectors we can pass to the `not()` method are limited to *filter* expressions that omit any element reference (allowing it to imply all element types). If we had passed the more explicit selector `img[title*=puppy]` to the `not()`

method instead, we would not get the expected result because element selectors are not supported.

Command syntax: not

not (expression)

Removes elements from the matched set according to the value of the `expression` parameter. If the parameter is a jQuery filter selector, any matching elements are removed. If an element reference is passed, that element is removed from the set.

Parameters

`expression` (String|Element|Array) A jQuery filter expression, element reference, or array of element references defining what is to be removed from the wrapped set.

Returns

The wrapped set.

As with `add()`, the `not()` method can also be used to remove individual elements from the wrapped set by passing a reference to an element or an array of element references. The latter is interesting and powerful because, remember, any jQuery wrapped set can be used as an array of element references.

At times, we may want to filter the wrapped set in ways that are difficult or impossible to express with a selector expression. In such cases, we may need to resort to programmatic filtering of the wrapped set items. We could iterate through all the elements of the set and use the `not(element)` method to remove the specific elements that do not meet our selection criteria. But the jQuery team didn't want us to have to resort to doing all that work on our own and so have defined the `filter()` method.

The `filter()` method, when passed a function, invokes that function for each wrapped element and removes any element whose function invocation returns the value `false`. Each invocation has access to the current wrapped element via the function context (`this`) in the body of the filtering function.

For example, let's say that, for some reason, we want to create a wrapped set of all `<td>` elements that contain a numeric value. As powerful as the jQuery selector expressions are, such a requirement is impossible to express using them. For such situations, the `filter()` method can be employed, as follows:

```
$('#td').filter(function(){return this.innerHTML.match(/^^\d+$/)});
```

This jQuery expression creates a wrapped set of all `<td>` elements and then invokes the function passed to the `filter()` method for each, with the current

matched elements as the `this` value for the invocation. The function uses a regular expression to determine if the element content matches the described pattern (a sequence of one or more digits), returning `false` if not. Every element whose filter function invocation returns `false` is removed from the wrapped set.

Command syntax: filter

filter(expression)

Filters out elements from the wrapped set using a passed selector expression, or a filtering function.

Parameters

`expression` (String|Function) Specifies a jQuery selector used to remove all elements that do not match from the wrapped set, or a function that makes the filtering decision. This function is invoked for each element in the set, with the current element set as the function context for that invocation. Any element that returns an invocation of `false` is removed from the set.

Returns

The wrapped set.

Again, bring up the Wrapped Set Lab, type the previous expression in, and execute it. You will see that the table cells for the Invented column are the only `<td>` elements that end up being selected.

The `filter()` method can also be used with a passed selector expression that conforms to the same constraints that we described earlier for the `not()` method, namely, filter selectors with an implied element type. When used in this manner, it operates in the inverse manner than the corresponding `not()` method, removing any elements that *do not* match the passed selector. This isn't a powerful method, as it's usually easier to use a more restrictive selector in the first place, but it can be useful within a chain of jQuery commands. Consider, for example,

```
$('.img').addClass('seeThrough').filter('[title*=dog]')  
  ➔ .addClass('thickBorder')
```

This chained statement selects all images and applies the `seeThrough` class to them and then reduces the set to only those image elements whose `title` attribute contains the string `dog` before applying another class named `thickBorder`. The result is that all the images end up semi-transparent, but only the tan dog gets the thick border treatment.

The `not()` and `filter()` methods give us powerful means to adjust a set of wrapped elements on the fly, based on just about any criteria regarding aspects

of the wrapped elements. We can also subset the wrapped set, based on the position of the elements within the set. Let's see which methods allow us to do that.

Obtaining subsets of the wrapped set

Sometimes we may wish to obtain a subset of the wrapped set, based on the position of elements within the set. jQuery provides a method to do that named `slice()`. This command creates and returns a *new* set from any contiguous portion, or a slice, of an original wrapped set. The syntax for this command follows:

Command syntax: slice

`slice(begin, end)`

Creates and returns a new wrapped set containing a contiguous portion of the matched set.

Parameters

- `begin` (Number) The zero-based position of the first element to be included in the returned slice.
- `end` (Number) The optional zero-based index of the first element not to be included in the returned slice, or one position beyond the last element to be included. If omitted, the slice extends to the end of the set.

Returns

The newly created wrapped set.

If we want to obtain a wrapped set that contains a single element from another set, based on its position in the original set, we could employ the `slice()` method, passing the zero-based position of the element within the wrapped set. For example, to obtain the third element, we write

```
$('.*').slice(2,3);
```

This statement selects all elements on the page and then generates a new set containing the third element in the matched set.

Note that this is different from `$('.*').get(2)`, which returns the third *element* in the wrapped set, not a wrapped set containing the element.

Therefore, a statement such as

```
$('.*').slice(0,4);
```

selects all elements on the page and then creates a set containing the first four elements.

To grab elements from the end of the wrapped set, the statement

```
$('.*').slice(4);
```

matches all elements on the page and then returns a set containing all but the first four elements.

And we're not done yet! jQuery also gives us the ability to obtain subsets of a wrapped set, based on the relationship of the wrapped items with other elements in the DOM. Let's see how.

2.3.4 Getting wrapped sets using relationships

jQuery allows us to get new wrapped sets from an existing set, based on the hierarchical relationships of the wrapped element to the other elements within the HTML DOM. Note that these methods operate in a slightly different manner than most earlier methods in this section that modify the wrapped set upon which they are called. Like the `slice()` method, the methods we'll see in this section return a *new* wrapped set, leaving the original set unchanged.

Table 2.4 shows these methods and their descriptions. Each of these methods accepts an optional selector expression that any selected elements must match. If no such selector parameter is passed, all eligible elements are selected.

These methods give us a large degree of freedom to select elements from the DOM, based on relationships to the other DOM elements. But we're still not done. Let's see how jQuery deals further with wrapped sets.

Table 2.4 Methods to obtain new wrapped set based on relationships

Method	Description
<code>children()</code>	Returns a wrapped set consisting of all unique children of the wrapped elements.
<code>contents()</code>	Returns a wrapped set of the contents of the elements, which may include text nodes, in the wrapped set. (Frequently used to obtain the contents of <code><iframe></code> elements.)
<code>next()</code>	Returns a wrapped set consisting of all unique next siblings of the wrapped elements.
<code>nextAll()</code>	Returns a wrapped set containing all the following siblings of the wrapped elements.
<code>parent()</code>	Returns a wrapped set consisting of the unique direct parents of all wrapped elements.
<code>parents()</code>	Returns a wrapped set consisting of the unique ancestors of all wrapped elements. This includes the direct parents as well as the remaining ancestors all the way up to, but not including, the document root.
<code>prev()</code>	Returns a wrapped set consisting of all unique previous siblings of the wrapped elements.
<code>prevAll()</code>	Returns a wrapped set containing all the previous siblings of the wrapped elements.
<code>siblings()</code>	Returns a wrapped set consisting of all unique siblings of the wrapped elements.

All of the methods in table 2.4, with the exception of `contents()`, accept a parameter containing a string that can be used to filter the results.

2.3.5 Even more ways to use a wrapped set

As if all that were not enough, there are still a few more tricks that jQuery has up its sleeve to let us define our collections of wrapped objects.

The `find()` method lets us search through an existing wrapped set and returns a new set that contains all elements that match a passed selector expression. For example, given a wrapped set in variable `wrappedSet`, we can get another wrapped set of all citations (`<cite>` elements) within paragraphs with

```
wrappedSet.find('p cite')
```

Note that if this were all to occur in a single statement, we could also accomplish this by passing a context parameter to a jQuery selector:

```
$('.p cite', wrappedSet)
```

Like many other jQuery wrapped set methods, the `find()` method's power comes when it's used within a jQuery chain of operations.

Command syntax: find

find(selector)

Returns a new wrapped set containing all elements of the original set that match the passed selector expression.

Parameters

`selector` (String) A jQuery selector that elements must match to become part of the returned set.

Returns

The newly created wrapped set.

In addition to finding elements in a wrapped set that match a selector, jQuery also provides a method to find elements that contain a specified string. The `contains()` method will return a new wrapped set that consists of all elements that contain the passed string anywhere within its body content. Consider

```
$('.p').contains('Lorem ipsum')
```

This expression yields a wrapped set containing all paragraphs that contain the text *Lorem ipsum*. Note that the string test is applied to all aspects of the body

content, including markup and attribute values of children elements, but it doesn't match markup or attribute values of the original elements being tested.

Command syntax: contains

contains(text)

Returns a new wrapped set composed of elements that contain the text string passed as the `text` parameter

Parameters

`text` (String) The text that an element must contain in order to be added to the returned set

Returns

The newly created wrapped set

The last method that we'll examine in this section is one that allows us to test a wrapped set to see if it contains at least one element that matches a given selector expression. The `is()` method returns `true` if at least one element matches the selector, and `false` if not. For example:

```
var hasImage = $('*').is('img');
```

This statement sets the value of the `hasImage` variable to `true` if the current page has an image element.

Command syntax: is

is(selector)

Determines if any element in the wrapped set matches the passed selector expression

Parameters

`selector` (String) The selector expression to test against the elements of the wrapped set

Returns

`true` if at least one element matches the passed selector; `false` if not

2.3.6 Managing jQuery chains

We've made a big deal (and will continue to do so, because it *is* a big deal) about the ability to chain jQuery wrapper methods together to perform a lot of activity in a single statement. This chaining ability not only allows us to write powerful operations in a concise manner, but it also improves efficiency because wrapped sets do not have to be recomputed in order to apply multiple commands to them.

Depending upon the methods used in a command chain, multiple wrapped sets may be generated. For example, using the `clone()` method (which we'll explore in

detail in chapter 3) generates a new wrapped set, which creates copies of the elements in the first set. If, once a new wrapped set is generated, we had no way to reference the original set, our ability to construct versatile jQuery command chains would be curtailed.

Consider the following statement:

```
$( 'img' ).clone().appendTo( '#somewhere' );
```

Two wrapped sets are generated within this statement: the original wrapped set of all the `` elements on a page and a second wrapped set consisting of copies of those elements. The `clone()` method returns this second set as its result, and it's that set that's operated on by the `appendTo()` command.

But what if we subsequently want to apply a command, such as adding a class name, to the original wrapped set *after* it's been cloned? We can't tack it onto the end of the existing chain; that would affect the clones, not the original wrapped set of images.

jQuery provides for this need with the `end()` command. This method, when used within a jQuery chain, will back up to a previous wrapped set and return it as its value so that subsequent operations will apply to the previous set.

Consider

```
$( 'img' ).clone().appendTo( '#somewhere' ).end().addClass( 'beenCloned' );
```

The `appendTo()` method returns the set of new clones, but by calling `end()` we back up to the previous wrapped set (the original images), which gets operated on by the `addClass()` command. Without the intervening `end()` command, `addClass()` would have operated on the set of clones.

Command syntax: end

end()

Used within a chain of jQuery command to back up the wrapped set to a previously returned set

Parameters

none

Returns

The previous wrapped set

It might help to think of the wrapped sets generated during a jQuery command chain as being held on a stack. When `end()` is called, the topmost (most recent)

wrapped set is popped from the stack, leaving the previous wrapped set exposed for subsequent commands to operate upon.

Another handy jQuery method that modifies the wrapped set stack is `andSelf()`, which merges the two topmost sets on the stack into a single wrapped set.

Command syntax: `andSelf`

`andSelf()`

Merges the two previous wrapped sets in a command chain

Parameters

none

Returns

The merged wrapped set

2.4 Summary

This chapter focused on creating and adjusting sets of elements (referred in this chapter and beyond as the *wrapped set*) via the many methods that jQuery provides for identifying elements on an HTML page.

jQuery provides a versatile and powerful set of *selectors*, patterned after the selectors of CSS, for identifying elements within a page document in a concise but powerful syntax. These selectors include not only the CSS2 syntax currently supported by most browsers, but also CSS3 syntax; a handful of custom selectors; and, with a plugin, even some basic XPath selectors.

jQuery also allows us to create or augment a wrapped set using HTML fragments to create new elements on the fly. These orphaned elements can be manipulated, along with any other elements in the wrapped set, and eventually attached to parts of the page document.

jQuery provides a robust set of methods to adjust the wrapped set to hone the contents of the set, either immediately after creation or midway through a set of chained commands. Applying filtering criteria to an already existing set can also easily create new wrapped sets.

All in all, jQuery gives us a lot of tools to make sure that we can easily and accurately identify the page elements that we wish to manipulate.

In this chapter, we covered a lot of ground without *doing* anything to the DOM elements of the page. But now that we know how to select the elements that we want to operate upon, we're ready to start adding life to our pages with the power of the jQuery commands.

3

Bringing pages to life with jQuery

This chapter covers:

- Getting and setting element attributes
- Manipulating element class names
- Setting element content
- Dealing with form element values
- Modifying the DOM tree

Remember those days (not all that long ago) when fledgling page authors would try to add pizzazz to their pages with counterproductive abominations such as marquee; blinking text; loud background patterns that interfered with the readability of page text; annoying animated GIFs; and, perhaps worst of all, unsolicited background sounds that would play upon page load (and served to test how fast a user could close down the browser)?

We've come a long way since then.

Today's savvy web developers and designers know better and use the power given to them by Dynamic HTML (DHTML) to *enhance* a user's web experience, rather than showcase annoying tricks.

Whether it's to incrementally reveal content, create input controls beyond the basic set provided by HTML, or give users the ability to tune pages to their own liking, DHTML—or DOM manipulation—has allowed many a web developer to amaze (not annoy) their users.

On an almost daily basis, we come across web pages that do something that makes us say, "Wow! I didn't know you could do that!" And being the commensurate professionals that we are (not to mention insatiably curious about such things), we immediately start looking at the source code to find out *how* they did it.

But rather than having to code up all that script ourselves, we'll find that jQuery provides a robust set of tools to manipulate the DOM, making those types of "Wow!" pages possible with a minimum of code. Whereas the previous chapter introduced us to the many ways jQuery lets us select DOM elements into a wrapped set, this chapter puts the power of jQuery to work performing operations on those elements to bring life and that elusive wow factor to our pages.

3.1 Manipulating element properties and attributes

Some of the most basic components we can manipulate when it comes to DOM elements are the *properties* and *attributes* assigned to those elements. These properties and attributes are initially assigned to the DOM elements as a result of parsing their HTML markup and can be changed dynamically under script control.

To make sure we have our terminology and concepts straight, consider the following HTML markup for an image element:

```

```

In this element's markup, the *tag name* is `img`, and the markup for `id`, `src`, `alt`, `class`, and `title` represents the element's *attributes*, each of which consists of a name and a value. This element markup is read and interpreted by the browser to

create the JavaScript object that represents this element in the DOM. In addition to storing the attributes, this object possesses a number of *properties*, including some that represent the values of the markup attributes (and even one that maintains the list of the attributes themselves). Figure 3.1 shows a simplified overview of this process.

The HTML markup is translated by the browser into a DOM element that represents the image. A `NodeList` object (one of the container types defined by the DOM) is created and assigned as a property of the element named `attributes`. There is a dynamic association between the attributes and their corresponding properties (which we'll refer to as *attribute properties*). Changing an attribute results in a change in the corresponding attribute property and vice versa. Even so, the values may not always be identical. For example, setting the `src` attribute of the image element to `image.gif` will result in the `src` property being set to the full absolute URL of the image.

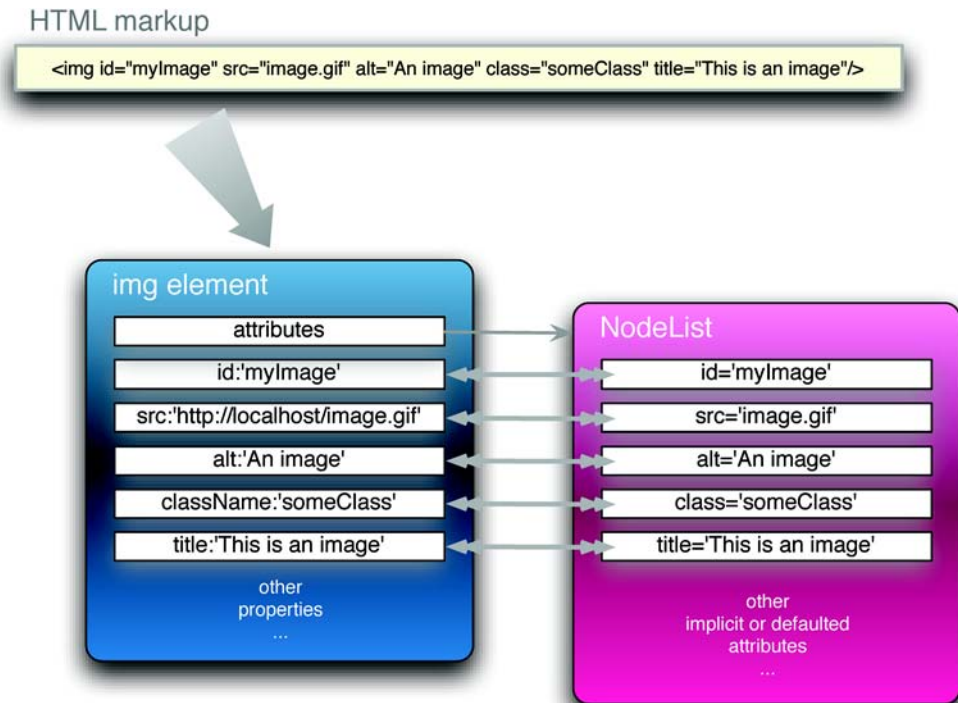


Figure 3.1 HTML markup is translated into DOM elements, including the attributes of the tag and the properties created from them.

For the most part, the name of a JavaScript attribute property matches that of any corresponding attribute, but there are some cases where they differ. For example, the `class` attribute in this example is represented by the `className` attribute property.

jQuery gives us the means to easily manipulate an element's attributes and gives us access to the element so that we can also change its properties. Which of these we choose to manipulate depends on what we want to do and how we want to do it.

Let's start by looking at getting and setting element properties.

3.1.1 Manipulating element properties

jQuery doesn't possess a specific command to obtain or modify the properties of elements. Rather, we use the native JavaScript notation to access the properties and their values. The trick is in getting to the element references in the first place.

The easiest way to inspect or modify the component elements of a matched set is with the `each()` command. The syntax of this command is as follows:

Command syntax: each

`each(iterator)`

Traverses all elements in the matched set invoking the passed iterator function for each.

Parameters

`iterator` (Function) A function called for each element in the matched set. The parameter passed to this function is set to the zero-based index of the element within the set, and the element itself is available as the `this` property of the function.

Returns

The wrapped set.

This command can be used to easily set a property value onto all elements in a matched set. For example, consider:

```
$('.img').each(function(n){
  this.alt='This is image['+n+' ] with an id of '+this.id;
});
```

This statement will invoke the inline function for each image element on the page, modifying its `alt` property using the order of the element and its `id` value. Note that, because this is an attribute property tied to an attribute of the same name, the `alt` attribute is also indirectly updated.

Similarly, we can collect all values for a specific property into an array using `each()`, as follows:

```
var allAlts = new Array();
$('img').each(function(){
    allAlts.push(this.alt);
});
```

If all we want to do is obtain the property value of a single element, remember that the matched set can be treated like a JavaScript array; we could obtain the property via

```
var altValue = $('#myImage')[0].alt;
```

Dealing with attributes is a little less straightforward than dealing with properties in JavaScript, so jQuery provides assistance for dealing with them. Let's look at how.

3.1.2 Fetching attribute values

As we'll find is true with many jQuery commands, the `attr()` command can be used either as a read or as a write operation. When jQuery commands can perform such disparate operations, the number and types of parameters passed into the command determine the variant of the command used.

The `attr()` command can be used to either fetch the value of an attribute from the first element in the matched set or set attribute values onto all matched elements.

The syntax for the fetch variant of the `attr()` command is as follows:

Command syntax: `attr`

`attr(name)`

Obtains the values assigned to the specified attribute for the first element in the matched set.

Parameters

`name` (String) The name of the attribute whose value is to be fetched.

Returns

The value of the attribute for the first matched element. The value `undefined` is returned if the matched set is empty or the attribute doesn't exist on the first element.

Even though we usually think of attributes as predefined by HTML, we can use `attr()` with custom attributes set through JavaScript or HTML markup. To illustrate

this, let's amend the `` element of our previous example with a custom markup attribute (highlighted in bold):

```

```

Note that we have added a custom attribute, unimaginatively named `custom`, to the element. We can retrieve that attribute's value, as if it were any of the standard attributes, with

```
$("#myImage").attr("custom")
```

WARNING Using a nonstandard attribute name such as `custom`, although a common sleight-of-hand trick, will cause your markup to be considered invalid; it will fail validation testing. This may have implications for accessibility, as well as for parsing by programs that expect your site to be written in valid HTML or XHTML.

Attribute names are not case sensitive in HTML. Regardless of how an attribute such as `title` is declared in the markup, we can access (or set, as we shall see) attributes using any variants of case: `title`, `TITLE`, `TitLE`, or any other combinations are all equivalent. In XHTML, even though attribute names must be lowercase in the markup, we can retrieve them using any case variant.

At this point you may be asking, “Why deal with attributes at all when accessing the properties is so easy (as seen in the previous section)?”

The answer to that question is that the jQuery `attr()` command is much more than a wrapper around the JavaScript `getAttribute()` and `setAttribute()` methods. In addition to allowing access to the set of element attributes, jQuery provides access to some commonly used properties that, traditionally, have been a thorn in the side of page authors everywhere due to their browser dependency.

This set of normalized-access names is shown in table 3.1.

Table 3.1 jQuery `attr()` normalized-access names

Normalized name	Source name
<code>class</code>	<code>className</code>
<code>cssFloat</code>	<code>styleFloat</code> for IE, <code>cssFloat</code> for others (when used with <code>.css</code>)
<code>float</code>	<code>styleFloat</code> for IE, <code>cssFloat</code> for others (when used with <code>.css</code>)
<code>for</code>	<code>htmlFor</code>

continued on next page

Table 3.1 jQuery `attr()` normalized-access names (continued)

Normalized name	Source name
<code>maxlength</code>	<code>maxLength</code>
<code>readonly</code>	<code>readOnly</code>
<code>styleFloat</code>	<code>styleFloat</code> for IE, <code>cssFloat</code> for others (when used with <code>.css</code>)

In addition to these helpful shortcuts, the set variant of `attr()` has some of its own handy features. Let's take a look.

3.1.3 Setting attribute values

There are two ways to set attributes onto elements in the wrapped set with jQuery. Let's start with the most straightforward that allows us set a single attribute at a time (for all elements in the wrapped set). Its syntax is as follows:

Command syntax: `attr`

`attr(name, value)`

Sets the named attribute onto all elements in the wrapped set using the passed value.

Parameters

- `name` (String) The name of the attribute to be set.
- `value` (String|Object|Function) Specifies the value of the attribute. This can be any JavaScript expression that results in a value, or it can be a function. See the following discussion for how this parameter is handled.

Returns

The wrapped set.

This variant of `attr()`, which may at first seem simple, is rather sophisticated in its operation.

In its most basic form, when the `value` parameter is any JavaScript expression that results in a value (including an array), the computed value of the expression is set as the attribute value.

Things get more interesting when the `value` parameter is a function reference. In such cases, the function is invoked for *each* element in the wrapped set, with the return value of the function used as the attribute value. When the function is invoked, it's passed a single parameter that contains the zero-based index of the element within the wrapped set. Additionally, the element is established

as the `this` variable for the function invocation, allowing the function to tune its processing for each specific element—the main power of using functions in this way.

Consider the following statement:

```
$('.*').attr('title',function(index) {  
    return 'I am element ' + index + ' and my name is ' +  
        (this.id ? this.id : 'unset');  
});
```

This command will run through all elements on the page, setting the `title` attribute of each element to a string composed using the `index` of the element within the DOM and the `id` attribute of each specific element.

We'd use this means of specifying the attribute value whenever that value is dependent upon other aspects of the elements, rather than some unrelated value.

The second set variant of `attr()` allows us to conveniently specify multiple attributes at a time.

Command syntax: `attr`

`attr(attributes)`

Sets the attributes and values specified by the passed object onto all elements of the matched set

Parameters

`attributes` (Object) An object whose properties are copied as attributes to all elements in the wrapped set

Returns

The wrapped set

This format is a quick and easy way to set multiple attributes onto all the elements of a wrapped set. The passed parameter can be any object reference, commonly an object literal, whose properties specify the names and values of the attributes to be set. Consider:

```
$('.input').attr(  
    { value: '', title: 'Please enter a value' }  
);
```

This statement sets the value of all `<input>` elements to the empty string, as well as sets the `title` to the string `Please enter a value`.

Note that if any property value in the object passed as the `value` parameter is a function reference, it operates in a manner similar to that described for the

previous format of `attr()`; the function is invoked for each individual element in the matched set.

WARNING Internet Explorer won't allow the name attribute of `<input>` elements to be changed. If you want to change the name of `<input>` elements in Internet Explorer, you must replace the element with a new element possessing the desired name.

Now we know how to get and set attributes. But what about getting rid of them?

3.1.4 Removing attributes

In order to remove an attribute from DOM elements, jQuery provides the `removeAttr()` command. Its syntax is as follows:

Command syntax: `removeAttr`

`removeAttr(name)`

Removes the specified attribute from every matched element

Parameters

`name` (String) The name of the attribute to be removed

Returns

The wrapped set

Note that removing an attribute doesn't remove any corresponding property from the JavaScript DOM element, though it may cause its value to change. For example, removing a `readonly` attribute from an element would cause the value of the element's `readOnly` property to flip from `true` to `false`, but the property itself isn't removed from the element.

Now let's look at some examples of how we might use this knowledge on our pages.

3.1.5 Fun with attributes

Let's say that we want to make all links on our site that pointed to external domains open in a new window. This is fairly trivial if we're in total control of the entire markup, as shown:

```
<a href="http://external.com" target="_blank">Some External Site</a>
```

That's all well and good, but what if we're running a Content Management System or a wiki, where end users will be able to add content, and we can't rely on them to add the `target="_blank"` to all external links? First, let's try and determine what we want; we want all links whose `href` attribute begins with `http://` to open in a new window (which we have determined can be done by setting the `target` attribute to `_blank`).

We can use the techniques we've learned in this section to do this concisely, as follows:

```
$("#a[href^=http://]").attr("target", "_blank");
```

First, we select all links with an `href` attribute starting with `http://` (which indicates that the reference is external). Then, we set its `target` attribute to `_blank`. Mission accomplished with a single line of jQuery code!

Another excellent use for jQuery's attribute functionality is helping to solve a long-standing issue with web applications (rich and otherwise): the Dreaded Double Submit Problem. This is a common problem in web applications when the latency of form submissions, sometimes several seconds or longer, gives users an opportunity to press the submit button multiple times, causing all manner of grief for the server-side code.

For our solution, we'll hook into the form's `submit` event and disable the submit button after its first press. That way, users won't get the opportunity to click the submit button more than once and will get a visual indication (assuming that disabled buttons appear so in their browser) that the form is in the process of being submitted. Don't worry about the details of event handling in the following example (we'll get more than enough of that coming up in chapter 5), but concentrate on the use of the `attr()` command:

```
$("#form").submit(function() {  
    $("#:submit",this).attr("disabled", "disabled");  
});
```

Within the body of the event handler, we grab all submit buttons that are inside our form with the `:submit` selector and modify the `disabled` attribute to the value `"disabled"` (the official W3C-recommended setting for the attribute). Note that when building the matched set, we provide a context value (the second parameter) of `this`. As we'll find out when we dive into event handling in chapter 5, the `this` pointer always refers to the page element to which the event was bound while operating inside event handlers.

WARNING Disabling the submit button(s) in this way doesn't relieve the server-side code from its responsibility to guard against double submission or any other types of validation. Adding this type of feature to the client code makes things nicer for the end user and helps prevent the double-submit problem under normal circumstances. It doesn't protect against attacks or other hacking attempts, and server-side code must continue to be on its guard.

We mentioned the `className` property earlier in this section as an example of the case where markup attribute names differ from property names; but, truth be told, class names are a bit special in other respects and are handled as such by jQuery. The next section will describe a better way to deal with class names than by directly accessing the `className` property or using the `attr()` command.

3.2 Changing element styling

If we want to change the styling of an element, we have two options. We can add or remove a CSS class, causing the existing stylesheet to restyle the element based on its new classes. Or we can operate on the DOM element itself, applying styles directly.

Let's look at how jQuery makes it simple to make changes to an element's style classes.

3.2.1 Adding and removing class names

The class name attributes and properties of DOM elements are unique in their format and semantics and are also important to the creation of rich user interfaces. The addition of class names to and removal of class names from an element is one of the primary means by which their stylistic rendering can be modified dynamically.

One of the aspects of element class names that make them unique—and a challenge to deal with—is that each element can be assigned any number of class names. In HTML, the `class` attribute is used to supply these names as a space-delimited string. For example:

```
<div class="someClass anotherClass yetAnotherClass"></div>
```

Unfortunately, rather than manifesting themselves as an array of names in the DOM element's corresponding `className` property, the class names appear as the

space-delimited string. How disappointing, and how cumbersome! This means that whenever we want to add class names to or remove class names from an element that already has class names, we need to parse the string to determine the individual names when reading it and be sure to restore it to valid space-delimited format when writing it.

Although it's not a monumental task to write code to handle all that, it's always a good idea to abstract such details behind an API that hides the mechanical details of such operations. Luckily, jQuery has already done that for us.

Adding class names to all the elements of a matched set is an easy operation with the following `addClass()` command:

Command syntax: `addClass`

`addClass` (*names*)

Adds the specified class name or class names to all elements in the wrapped set

Parameters

names (String) A string containing the class name to add or, if multiple class names are to be added, a space-delimited string of class names

Returns

The wrapped set

Removing class names is as straightforward with the following `removeClass()` command:

Command syntax: `removeClass`

`removeClass` (*names*)

Removes the specified class name or class names from each element in the wrapped set

Parameters

names (String) A string containing the class name to remove or, if multiple class names are to be removed, a space-delimited string of class names

Returns

The wrapped set

Often, we may want to switch a set of styles back and forth, perhaps to indicate a change between two states or for any other reasons that make sense with our interface. jQuery makes it easy with the `toggleClass()` command.

Command syntax: toggleClass**toggleClass(name)**

Adds the specified class name if it doesn't exist on an element, or removes the name from elements that already possess the class name. Note that each element is tested individually, so some elements may have the class name added, and others may have it removed.

Parameters

name (String) A string containing the class name to toggle.

Returns

The wrapped set.

One situation where the `toggleClass()` command is most useful is when we want to switch visual renditions between elements quickly and easily. Remember the zebra-stripe example of figure 1.1? What if we had some valid reason to swap the colored background from the odd rows to the even rows (and perhaps back again) when certain events occurred? The `toggleClass()` command would make it almost trivial to add a class name to every other row, while removing it from the remainder.

Let's give it a whirl. In the file `chapter3/zebra.stripes.html`, you'll find a copy of the same page from chapter 1 with some minor changes. We added the following function to the `<script>` element in the page header:

```
function swap() {  
    $('tr').toggleClass('striped');  
}
```

This function uses the `toggleClass()` command to toggle the class named `stripe` for all `<tr>` elements. We also added calls to this function as the `onmouseover` and `onmouseout` attributes of the table:

```
<table onmouseover="swap();" onmouseout="swap();">
```

The result is that every time the mouse cursor enters or leaves the table, all `<tr>` elements with the class `striped` will have the class removed, and all `<tr>` elements without the class will have it added. This (rather annoying) activity is shown in the two parts of figure 3.2.

Manipulating the stylistic rendition of elements via CSS class names is a powerful tool, but sometimes we want to get down to the nitty-gritty styles themselves as declared directly on the elements. Let's see what jQuery offers us for that.

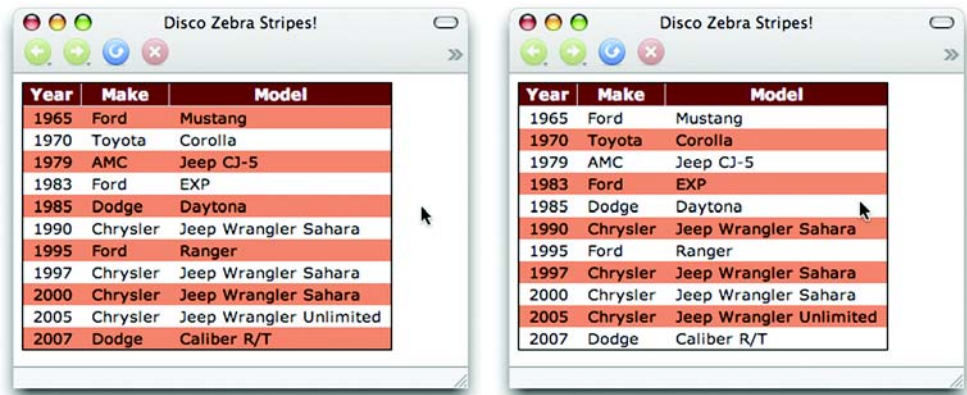


Figure 3.2 The presence or absence of the striped class is toggled whenever the mouse cursor enters or leaves the table.

3.2.2 Getting and setting styles

Although modifying the class of an element allows us to choose which predetermined set of defined stylesheet styles should be applied, sometimes we want to override the stylesheet altogether. Applying styles directly on the elements themselves will automatically override stylesheets, giving us more fine-grained control over individual elements and their styles.

The `css()` method works similarly to the `attr()` method, allowing us to set an individual CSS property by specifying its name and value, or a series of elements by passing in an object. First, let's look at specifying a name and value.

Command syntax: `css`

`css(name, value)`

Sets the named CSS style property to the specified value for each matched element.

Parameters

name (String) The name of the CSS property to be set.

value (String|Number|Function) A string, number, or function containing the property value. If a function is passed as this parameter, it will be invoked for each element of the wrapped set with its return value serving as the value for the CSS property. The `this` property for each function invocation will be set to the element being evaluated.

Returns

The wrapped set.

As described, the value accepts a function in a similar fashion to the `attr()` commands. This means that we can, for instance, expand the width of all elements in the wrapped set by 20 pixels as follows:

```
$( "div.expandable" ).css( "width", function() {  
    return $(this).width() + 20 + "px";  
});
```

Don't worry that we haven't discussed the `width()` command yet. It does exactly what you would expect it to (namely, return the width of the element as a number), and we'll discuss it in more detail shortly. One interesting side note is that the normally problematic `opacity` property will work perfectly across browsers by passing in a value between 0.0 and 1.0; no more messing with IE alpha filters, `-moz-opacity`, and the like!

Next, let's look at using the shortcut form of the `css()` command, which works exactly as the shortcut version of `attr()` worked.

Command syntax: `css`

`css(properties)`

Sets the CSS properties specified as keys in the passed object to their associated values for all matched elements

Parameters

`properties` (Object) Specifies an object whose properties are copied as CSS properties to all elements in the wrapped set

Returns

The wrapped set

As in the shortcut version of the `attr()` command, we can use functions as values to any CSS property in the `properties` parameter object, and they will be called on each element in the wrapped set to determine the value that should be applied.

Lastly, we can use `css()` with a name passed in to retrieve the computed style of the property associated with that name. When we say computed style, we mean the style after all linked, embedded, and inline CSS has been applied. Impressively, this works perfectly across all browsers, even for `opacity`, which returns a string representing a number between 0.0 and 1.0.

Command syntax: css**css (name)**

Retrieves the computed value of the CSS property specified by `name` for the first element in the wrapped set

Parameters

`name` (String) Specifies the name of a CSS property whose computed value is to be returned

Returns

The computed value

Keep in mind that this variant of the `css()` command always returns a string, so if you need a number or some other type, you'll need to parse the returned value.

For a small set of CSS values that are commonly accessed, jQuery thoughtfully provides convenience commands that easily access these values and convert them to the most commonly used types. Specifically, we can get (or set) the width and height of an element as a number by using the convenient `width()` and `height()` commands. To set the width or height:

Command syntax: width and height**width (value)****height (value)**

Sets the width or height of all elements in the matched set

Parameters

`value` (Number) The value to be set in pixels

Returns

The wrapped set

Keep in mind that these are shortcuts for the more verbose `css()` function, so

```
$( "div.myElements" ).width(500)
```

is identical to

```
$( "div.myElements" ).css( "width", "500px" )
```

To retrieve the width or height:

Command syntax: width and height

`width()`

`height()`

Retrieves the width or height of the first element of the wrapped set

Parameters

none

Returns

The computed width or height as a number

The fact that the width and height values are returned from these functions as numbers isn't the only convenience that these commands bring to the table. If you've ever tried to find the width or height of an element by looking at its `style.width` or `style.height` property, you were confronted with the sad fact that these properties are only set by the corresponding `style` attribute of that element; to find out the dimensions of an element via these properties, you have to set them in the first place. Not exactly a paragon of usefulness!

The `width()` and `height()` commands, on the other hand, compute and return the size of the element. Although knowing the precise dimensions of an element in simple pages that let their elements lay out wherever they end up isn't usually necessary, knowing such dimensions in Rich Internet Applications is crucial to be able to correctly place active elements such as context menus, custom tool tips, extended controls, and other dynamic components.

Let's put them to work. Figure 3.3 shows a sample set up with two primary elements: a test subject `<div>` that contains a paragraph of text (also with a border

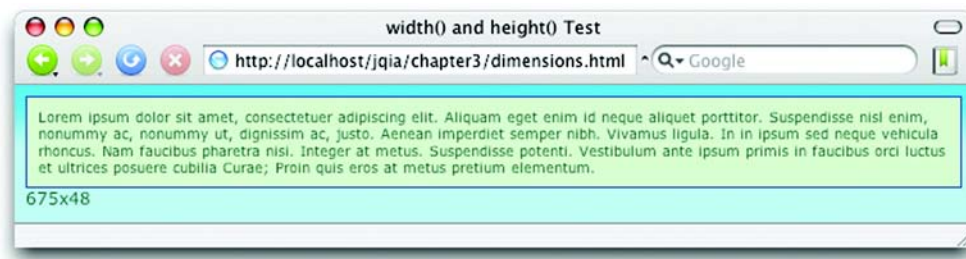


Figure 3.3 The width and height of the test element aren't fixed and depend on the width of the browser window.

and background color for emphasis) and a second `<div>` in which to display the dimensions.

The dimensions of the test subject aren't known in advance because no style rules specifying dimensions are applied. The width of the element is determined by the width of the browser window, and its height depends on how much room will be needed to display the contained text. Resizing the browser window would cause both dimensions to change.

In our page, we define a function that will use the `width()` and `height()` commands to obtain the dimensions of the test subject `<div>` (named `testSubject`) and display the resulting values in the second `<div>` (named `display`).

```
function report() {
  $('#display').html(
    $('#testSubject').width()+'x'+$('#testSubject').height()
  );
}
```

We call this function in the ready handler of the page, resulting in the display of the values 675 and 48 for that particular size of browser window, as shown in figure 3.3.

We also add a call to the function in the `onresize` attribute of the `<body>` element:

```
<body onresize="report();">
```

Resizing the browser results in the display shown in figure 3.4.

This ability to determine the computed dimensions of an element at any point is crucial to accurately positioning dynamic elements on our pages.

The full code of this page is shown in listing 3.1 and can be found in the file `chapter3/dimensions.html`.

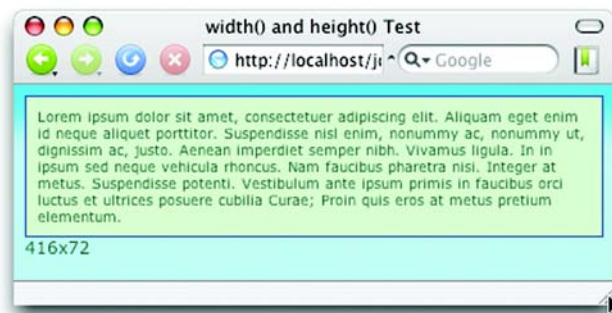


Figure 3.4
Resizing the browser causes the test subject to change size; this change is reflected in the computed values.

Listing 3.1 Dynamically tracking the dimensions of an element

```

<html>
  <head>
    <title>width() and height() Test</title>
    <link rel="stylesheet" type="text/css" href="../common.css">
    <script type="text/javascript"
      src="../scripts/jquery-1.2.1.js"></script>
    <script type="text/javascript">
      $(function(){
        report();
      });
      function report() {
        $('#display').html(
          $('#testSubject').width()+'x'+$('#testSubject').height()
        );
      }
    </script>
    <style>
      #testSubject {
        background-color: plum;
        border: 1px solid darkmagenta;
        padding: 8px;
        font-size: .85em;
      }
    </style>
  </head>
  <body onresize="report();">
    <div id="testSubject">
      Lorem ipsum dolor sit amet, consectetur adipiscing elit.
      Aliquam eget enim id neque aliquet porttitor. Suspendisse
      nisl enim, nonummy ac, nonummy ut, dignissim ac, justo.
      Aenean imperdiet semper nibh. Vivamus ligula. In in ipsum
      sed neque vehicula rhoncus. Nam faucibus pharetra nisi.
      Integer at metus. Suspendisse potenti. Vestibulum ante
      ipsum primis in faucibus orci luctus et ultrices posuere
      cubilia Curae; Proin quis eros at metus pretium elementum.
    </div>
    <div id="display"></div>
  </body>
</html>

```

Invokes reporting function at page ready

Displays width and height of test subject

Applies styling to test subject

Reports dimensions on window resize

Declares test subject with dummy text

Displays dimensions in this area

You may have picked up on the fact that we embedded behavior in the HTML markup of this example in violation of the rules of Unobtrusive JavaScript. That's OK for now, but in the next chapter we'll learn a better way to bind event handlers.

Now that we've explored manipulating the styles on a wrapped set of elements, let's take a look at a couple of related style-oriented activities that you might want to accomplish, and how to achieve them.

3.2.3 More useful style-related commands

It's extremely common to need to determine whether an element has a particular class. With jQuery, we can do that by calling the `hasClass()` function.

```
$("#p:first").hasClass("surpriseMe")
```

This will return `true` if any element in the matched set has the specified class. The syntax of this command is as follows:

Command syntax: `hasClass`

hasClass(name)

Determines if any element of the matched set possesses the passed class name

Parameters

`name` (String) The class name to be checked

Returns

Returns `true` if any element in the wrapped set possesses the passed class name;
`false` if not

Recalling the `is()` command from chapter 2, we could achieve the same thing with

```
$("#p:first").is(".surpriseMe")
```

In fact, jQuery's inner workings implement the `hasClass()` function exactly that way! But arguably, the `hasClass()` command makes for more readable code.

Another commonly desired ability is to obtain the list of classes defined for a particular element as an array instead of the cumbersome space-separated list. We could try to achieve that by writing

```
$("#p:first").attr("class").split(" ");
```

Recall that the `attr()` command will return `undefined` if the attribute in question doesn't exist, so this statement will throw an error if the `<p>` element doesn't possess any class names. We could solve this by first checking for the attribute, and if we wanted to wrap the entire thing in a repeatable, useful jQuery extension, we could write

```
$.fn.getClassNames = function() {  
  if (name = this.attr("className")) {
```



```
        return name.split(" ");
    }
    else {
        return [];
    }
};
```

But don't worry about the specifics of the syntax for extending jQuery; we'll go into that in more detail in chapter 7. What's important is that we can use `get-classNames()` anywhere in our script to obtain an array of class names or an empty array if an element has no classes. Nifty!

Now that we've learned how to get and set the styles of elements, let's discuss different ways for modifying their contents.

3.3 Setting element content

When it comes to modifying the contents of elements, there's an ongoing debate regarding which technique is better: using DOM API methods or changing their inner HTML. In most cases, modifying an element's HTML is easier and more effective, so jQuery gives us a number of methods to do so.

3.3.1 Replacing HTML or text content

First is the simple `html()` command, which allows us to retrieve the HTML contents of an element when used without parameters or, as we've seen with other jQuery functions, to set its contents when used with a parameter.

Here's how to get the HTML content of an element:

Command syntax: `html`

`html()`

Obtains the HTML content of the first element in the matched set.

Parameters

none

Returns

The HTML content of the first matched element. The returned value is identical to accessing the `innerHTML` property of that element.

Here's how to set the HTML content of all matched elements:

Command syntax: html**html (text)**

Sets the passed HTML fragment as the content of all matched elements

Parameters

`text` (String) The HTML fragment to be set as the element content

Returns

The wrapped set

We can also set or get only the text contents of elements. The `text()` command, when used without parameters, returns a string that's the concatenation of all text. For example, let's say we have the following HTML fragment:

```
<ul id="theList">
  <li>One</li>
  <li>Two</li>
  <li>Three</li>
  <li>Four</li>
</ul>
```

The statement

```
var text = $('#theList').text();
```

results in variable `text` being set to `OneTwoThreeFour`.

Command syntax: text**text ()**

Concatenates all text content of the wrapped elements and returns it as the result of the command

Parameters

none

Returns

The concatenated string

We can also use the `text` command to set the text content of the wrapped elements. The syntax for this format is as follows:

Command syntax: text**text (content)**

Sets the text content of all wrapped elements to the passed value. If the passed text contains angle brackets (< and >), these characters are replaced with their equivalent HTML entities.

Parameters

`content` (String) The text content to be set into the wrapped elements. Any angle bracket characters are escaped as HTML entities.

Returns

The wrapped set.

Note that setting the inner HTML or text of elements using these commands will replace contents that were previously in the elements, so use these commands carefully. If you don't want to bludgeon all of an element's previous content, a number of other methods will leave the contents of the elements as they are but modify their contents or surrounding elements. Let's look at them.

3.3.2 Moving and copying elements

To add content to the end of existing content, the `append()` command is available.

Command syntax: append**append (content)**

Appends the passed HTML fragment or elements to the content in all matched elements.

Parameters

`content` (String|Element|Object) A string, element, or wrapped set to append to the elements of the wrapped set. See the following description for details.

Returns

The wrapped set.

This function accepts a string containing an HTML fragment, a reference to an existing or newly created DOM element, or a jQuery wrapped set of elements.

Consider the following simple case:

```
$( 'p' ).append( '<b>some text<b>' );
```

This statement appends the HTML fragment created from the passed string to the end of the existing content of all `<p>` elements on the page.

A more complex use of this command identifies already-existing elements of the DOM as the items to be appended. Consider the following:

```
$( "p.appendToMe" ).append( $( "a.appendMe" ) )
```

This statement appends all links with the class `appendMe` to `<p>` elements with the class `appendToMe`. The disposition of the original elements depends on the number of elements serving as the target of the `append`. If there is a single target, the element is removed from its original location—performing a *move* operation of the original element to a new parent. In the case where there are multiple targets, the original element remains in place and copies of it are appended to each of the targets—a *copy* operation.

In place of a full-blown wrapped set, we can also reference a specific DOM element, as shown:

```
var toAppend = $( "a.appendMe" )[ 0 ];
$( "p.appendToMe" ).append( toAppend );
```

Whether the element identified by `toAppend` is moved or copied again depends on the number of elements identified by `$("p.appendToMe")`: a move if one element is matched, a copy if more than one element is matched.

If we want to move or copy an element from one place to another, a simpler approach uses the `appendTo()` command, which allows us to grab an element and move it somewhere else in the DOM.

Command syntax: `appendTo`

`appendTo(target)`

Moves all elements in the wrapped set to the end of the content of the specified target(s).

Parameters

target (String|Element) A string containing a jQuery selector or a DOM element. Each element of the wrapped set will be appended to that location. If more than one element matches a string selector, the element will be copied and appended to each element matching the selector.

Returns

The wrapped set.

A common semantic for most functions in this section is that an element will be *moved* if the destination identifies only one target. If the destination denotes multiple target elements, the source element will remain in its original location and be *copied* to each destination.

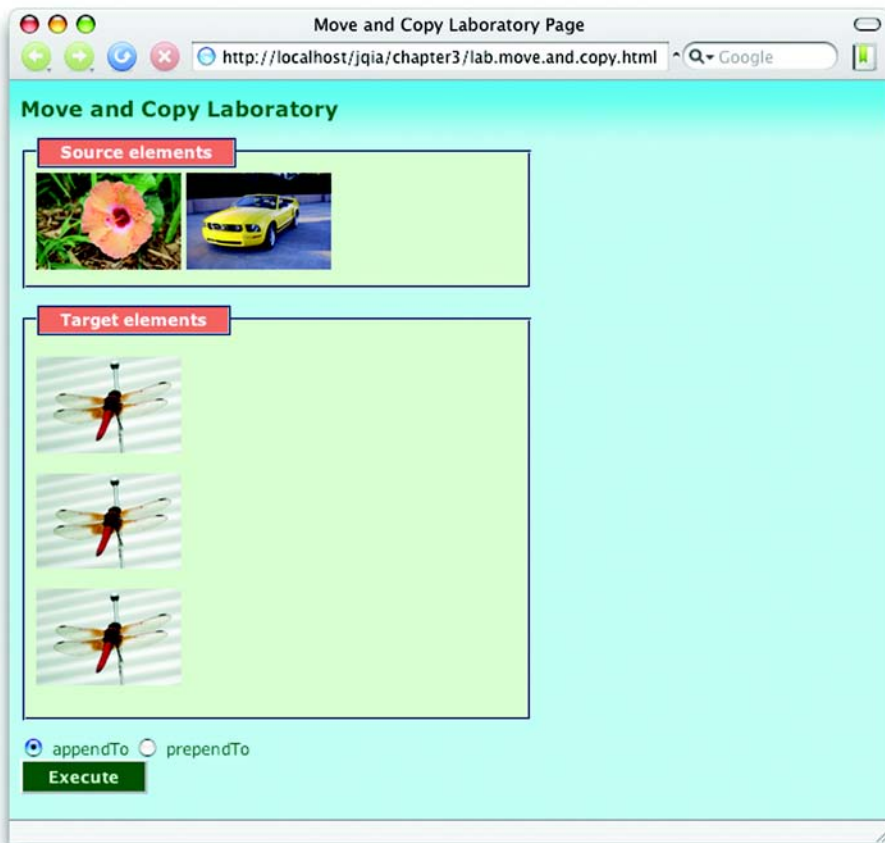


Figure 3.5 The Move and Copy Laboratory Page is set up to illustrate the operation of the `appendTo` and `prependTo` commands.

Before we move on to other commands that work in a similar fashion, let's look at an example to make sure that this important concept is clear. We've set up a lab page with some elements to serve as the source of an `appendTo()` operation and some to serve as targets. Upon initial display, this Move and Copy Laboratory Page looks as shown in figure 3.5.

The HTML markup for the test candidates in the fieldsets is as follows:

```
<fieldset id="source">
  <legend>Source elements</legend>
  
  
</fieldset>
```

```
<fieldset id="targets">
  <legend>Target elements</legend>
  <p></p>
  <p></p>
  <p></p>
</fieldset>
```

The source fieldset contains two images: one with an id of `flower`, and one with an id of `car`. These image elements will serve as the source of the commands that we'll apply. The target fieldset contains three `<p>` elements, each of which contains an image. These paragraph elements will serve as the target of our commands.

Display this page, which can be found in the file `chapter3/lab.move.and.copy.html`, in your browser. Leaving the `appendTo` radio button checked, click the `Execute` button, which will execute code equivalent to the following:

```
$('#flower').appendTo('#targets p')
$('#car').appendTo('#targets p:first')
```

The first statement executes the `appendTo()` command on the `flower` image, specifying the three paragraph elements as the target. Because there's more than one target element, we would expect the `flower` image to be copied. The second statement issues the same command for the `car` image, but specifying only the first of the paragraph elements as the target. Because there is only one target, we would expect the `car` image to be moved.

The display of figure 3.6, taken after the click of the `Execute` button, shows that these expectations were correct.

It's clear from these results that when there are multiple targets, the source element is copied, and when there is only a single target the source element is moved.

A number of related commands work in a fashion similar to `append()` and `appendTo()`:

- `prepend()` and `prependTo()`—Work like `append()` and `appendTo()`, but insert the source element before the destination target's contents instead of after. These commands can also be demonstrated in the `Move and Copy Laboratory` by clicking the `PrependTo` radio button before clicking `Execute`.
- `before()` and `insertBefore()`—Insert the element before the destination elements instead of before the destination's first child.
- `after()` and `insertAfter()`—Insert the element after the destination elements instead of after the destination's last child.

Because the syntax of these commands is so similar to that of the `append` class of commands, we won't waste the space to show individual syntax descriptions for

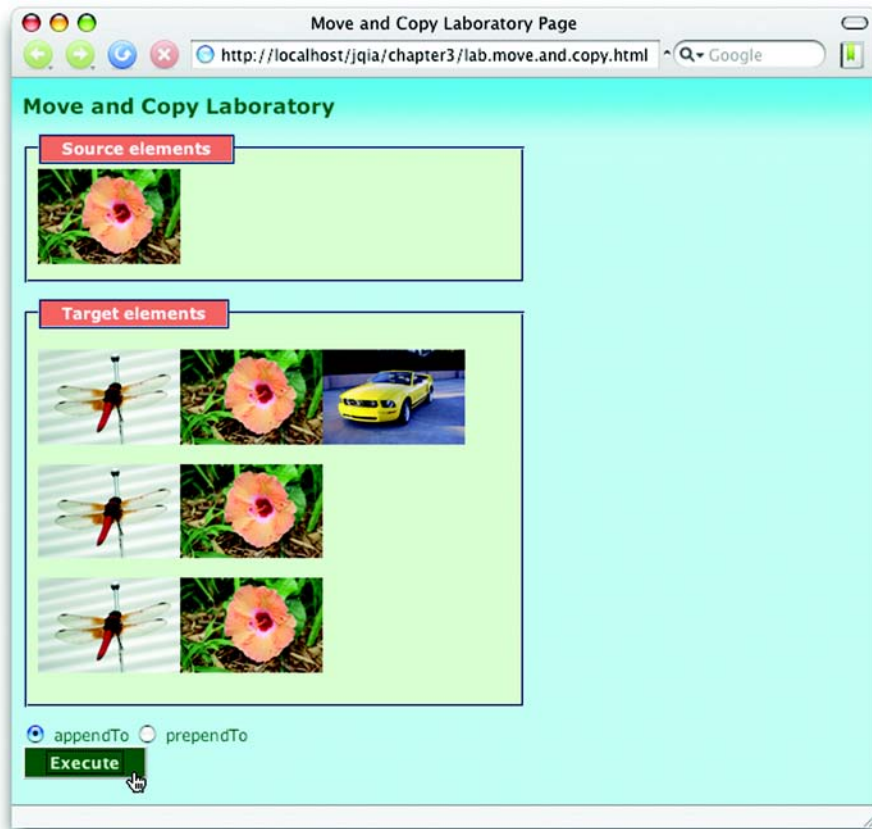


Figure 3.6 After execution, it's clear that the car has been moved and the flower has been copied.

them. Please refer back to the syntax blocks for `append()` and `appendTo()` for the format of the syntax for these commands.

One more thing before we move on...

Remember back in the previous chapter when we showed how to create new HTML fragments with the jQuery `$()` wrapper function? Well, that becomes a really useful trick when paired with the `appendTo()`, `prependTo()`, `insertBefore()`, and `insertAfter()` commands. Consider the following:

```
$('<p>Hi there!</p>').insertAfter('p img');
```

This statement creates a friendly paragraph and inserts a copy of it after every image element within a paragraph element.

Sometimes, rather than inserting elements into other elements, we want to do the opposite. Let's see what jQuery offers for that.

3.3.3 Wrapping elements

Another type of DOM manipulation that we'll often need to perform is to wrap an element (or series of elements) in some markup. For example, we might want to wrap all links of a certain class inside a `<div>`. We can accomplish such DOM modifications by using jQuery's `wrap()` command. Its syntax is as follows:

Command syntax: wrap

`wrap(wrapper)`

Wraps the elements of the matched set with the passed HTML tags or a clone of the passed element.

Parameters

`wrapper` (String|Element) The opening and closing tags of the element with which to wrap each element of the matched set, or an element to be cloned and server as the wrapper.

Returns

The wrapped set.

To wrap each link with the class `surprise` in a `<div>` with the class `hello`, we write

```
$("#a.surprise").wrap("<div class='hello'></div>")
```

If we wanted to wrap the link in a clone of the first `<div>` element on the page:

```
$("#a.surprise").wrap($("#div:first")[0]);
```

When multiple elements are collected in a matched set, the `wrap()` method operates on each one individually. If we'd rather wrap all the elements in the set as a unit, we can use the `wrapAll()` method instead:

Command syntax: wrapAll

`wrapAll(wrapper)`

Wraps the elements of the matched set, as a unit, with the passed HTML tags or a clone of the passed element.

Parameters

`wrapper` (String|Element) The opening and closing tags of the element with which to wrap each element of the matched set, or an element to be cloned and server as the wrapper.

Returns

The wrapped set

Sometimes we may not want to wrap the elements that are in a matched set, but rather their *contents*. For just such cases, the `wrapInner()` method is available:

Command syntax: wrapInner

`wrapInner(wrapper)`

Wraps the contents, to include text nodes, elements of the matched set with the passed HTML tags or a clone of the passed element.

Parameters

`wrapper` (String|Element) The opening and closing tags of the element with which to wrap each element of the matched set, or an element to be cloned and server as the wrapper.

Returns

The wrapped set

Now that we know how to create, wrap, copy, and move elements, we may wonder how we make them go away.

3.3.4 Removing elements

If we want to empty or remove a set of elements, this can be accomplished with the `remove()` command whose syntax is as follows:

Command syntax: remove

`remove()`

Removes all elements in the wrapped set from the page DOM

Parameters

none

Returns

The wrapped set

Note that, as with many other jQuery commands, the wrapped set is returned as the result of this command. The elements that were removed from the DOM are still referenced by this set (and hence not yet eligible for garbage collection) and can be further operated upon using other jQuery commands including `appendTo()`, `prependTo()`, `insertBefore()`, `insertAfter()`, and any other similar behaviors we'd like.

To empty DOM elements of their contents, we can use the `empty()` command. Its syntax is as follows:

Command syntax: `empty`

`empty()`

Removes the content of all DOM elements in the matched set

Parameters

none

Returns

The wrapped set

A common idiom is to use the `remove()` and `after()` commands to effect a replacement operation. Consider the following:

```
$("#div.elementToReplace").after("<p>I am replacing the div</p>").remove();
```

Because the `after()` function returns the original wrapped set containing the `<div>`, we can then remove it, resulting in a replacement of the original `<div>` with the newly created `<p>` element.

If this is an idiom that you'll find yourself using over and over again, remember that you can always wrap up such useful statements as extensions to jQuery with

```
$.fn.replaceWith = function(html) {  
    return this.after(html).remove();  
};
```

Its usage, to perform the same operation as shown in the previous example, is

```
$("#div.elementToReplace").replaceWith("<p>I am replacing the div</p>");
```

Here, we see another example of creating extensions to jQuery. It's important when doing so to return the wrapped set so that the chain can be continued after it has been called (unless the purpose of the extension is to return other data). You should be getting the hang of extending jQuery, but don't worry if you aren't; it'll get fuller treatment later in this book (in chapter 7, to be exact).

Sometimes, we don't want to move elements, but to copy them...

3.3.5 Cloning elements

One more way that we can manipulate the DOM is to make copies of elements to attach elsewhere in the tree. jQuery provides a handy wrapper method for doing so with its `clone()` command.

Command syntax: clone

`clone(copyHandlers)`

Creates copies of the elements in the wrapped set and returns a new wrapped set that contains them. The elements and any children are copied. Event handlers are optionally copied depending upon the setting of the `copyHandlers` parameter.

Parameters

`copyHandlers` (Boolean) If `true`, event handlers are copied. If `false`, or omitted, handlers are not copied.

Returns

The newly created wrapped set.

Making a copy of existing elements with `clone()` isn't useful unless we do something with the carbon copies. Generally, once the wrapped set containing the clones is generated, another jQuery command is applied to stick them somewhere in the DOM. For example:

```
$('#img').clone().appendTo('fieldset.photo');
```

This statement makes copies of all image elements and appends them to all `<fieldset>` elements with the class name `photo`.

A slightly more complex example is as follows:

```
$('#ul').clone().insertBefore('#here');
```

This command chain performs a similar operation but the targets of the cloning operation—all `` elements—are copied, including their children (it's likely that any `` element will have a number of `` children).

One last example:

```
$('#ul').clone().insertBefore('#here').end().hide();
```

This statement performs the same operation as the previous example, but after the insertion of the clones, the `end()` command is used to select the original wrapped set (the original targets) and hide them. This emphasizes how the cloning operation creates a new set of elements in a new wrapper.

Now that we've discussed handling general DOM elements, let's take a brief look at handling a special type of element: form elements.

3.4 Dealing with form element values

Because form elements have special properties, jQuery's core contains a number of convenience functions for activities like getting and setting their values, serializing them, and selecting elements based on form properties. They will serve us well in simple cases, but the Form Plugin—an officially sanctioned plugin developed by members of the jQuery Core Team—provides a much more robust set of functionality. We'll discuss the Form Plugin in chapter 9.

NOTE When we use the term *form element*, we are referring to the elements that can appear within a form and possess `name` and `value` attributes that submit to the server as request parameters when the form is submitted. Dealing with such elements by hand in script can be tricky because, not only can elements be disabled, but the W3C defines an *unsuccessful* state for controls. This state determines which elements should be ignored during a submission, and it's a tad on the complicated side.

That said, let's take a look at one of the most common operations we'll want to perform on a form element: getting access to its value. jQuery's `val()` command takes care of the most common cases, returning the `value` attribute of a form element for the first element in the wrapped set. Its syntax is as follows:

Command syntax: `val`

`val()`

Returns the `value` property of the first element in the matched set. When the element is a multi-select element, the returned value is an array of all selections.

Parameters

none

Returns

The fetched value or values.

This command, although quite useful, has a number of limitations of which we need to be wary. If the first element in the wrapped set isn't a form element, a JavaScript error is thrown. This command also doesn't distinguish between the

checked or unchecked states of check boxes and radio buttons, and will return the value of check boxes or radio buttons as defined by their `value` attribute, regardless of whether they are checked or not.

For radio buttons, the power of jQuery selectors combined with the `val()` method saves the day. Consider a form with a radio group (a set of radio buttons with the same name) named `radioGroup` and the following expression:

```
$('#[name=radioGroup]:checked').val()
```

This expression returns the value of the single checked radio button (or undefined if none is checked). That's a lot easier than looping through the buttons looking for the checked element, isn't it? Since `val()` only considers the first element in a wrapped set, it's not as useful for check box groups where more than one control might be checked.

If we want to obtain the values with which the controls would be submitted through a form submission, we'll be much better off using the `serialize()` command (which we'll see in chapter 8) or the official Form Plugin.

Another common operation we'll perform is to set the value of a form element. The `val()` command is also used for this purpose by supplying a value. Its syntax is as follows:

Command syntax: `val`

`val(value)`

Sets the passed value as the `value` of all matched form elements

Parameters

`value` (String) A string that's set as the `value` property of each form element in the wrapped set

Returns

The wrapped set

Like the `get` variant of this command, this function has its limitations. It can't set multiple values into a multiselect list, for example. This is the reason that much more robust functionality is available in the Form Plugin. In addition to lifting the limitations mentioned so far, it's capable of such operations as retrieving an array of values for check box groups, serializing the elements in the wrapped set, clearing fields, and even converting a DOM form into a format suitable for use with Ajax.

Another way that the `val()` method can be used is to cause check box or radio elements to become checked, or to select options within a `<select>` element. The syntax of this variant of `val()` is as follows:

Command syntax: `val`

`val(values)`

Causes any check boxes, radio buttons, or options of `<select>` elements in the wrapped set to become checked or selected if their values match any of the values passed in the `values` array.

Parameters

`values` (Array) An array of values that will be used to determine which elements are to be checked or selected.

Returns

The wrapped set.

Consider the following statement:

```
$('#input,select').val(['one', 'two', 'three']);
```

This statement will search all the `<input>` and `<select>` elements on the page for values that match any of the input strings: *one*, *two* or *three*. Any check boxes or radio buttons that are found to match will become checked, and any options that match will become selected.

This makes `val()` useful for much more than just text elements.

3.5 Summary

In this chapter, we've gone beyond the art of selecting elements and started manipulating them. With the techniques we've learned so far, we can select elements using powerful criteria, and then move them surgically to any part of the page.

We can choose to copy elements, or to move them, or even create brand new elements from scratch. We can append, prepend, or wrap any element or set of elements on the page. And we've learned how we can treat a single element or a set of elements the same, leading to powerful yet succinct logic.

With that behind us, we're ready to start looking into more advanced concepts, starting with the typically messy job of handling events in our pages.

4

*Events are where
it happens!*

This chapter covers

- The event models as implemented by the browsers
- Using jQuery to bind event handlers to elements
- The `Event` object instance
- Triggering event handlers under script control

Anyone familiar with the Broadway show *Cabaret*, or its subsequent Hollywood film, probably remembers the song “Money Makes the World Go Around.” Although this cynical view might be applicable to the physical world, in the virtual realm of the World Wide Web, it’s events that make it all happen!

Like many other GUI management systems, the interfaces presented by HTML web pages are *asynchronous* and *event-driven* (even if the HTTP protocol used to deliver them to the browser is wholly synchronous in nature). Whether a GUI is implemented as a desktop program using Java Swing, X11, the .NET framework, or a page in a web application using HTML and JavaScript, the procedure is pretty much the same:

- 1 Set up the user interface.
- 2 Wait for something interesting to happen.
- 3 React accordingly.
- 4 Repeat.

The first step sets up the *display* of the user interface; the others define its *behavior*. In web pages, the browser handles the setup of the display in response to the markup (HTML and CSS) that we send to it. The script we include in the page defines the behavior of the interface.

This script takes the form of *event handlers*, also known as *listeners*, that react to the various events that occur while the page is displayed. These events could be generated by the system (such as timers or the completion of asynchronous requests) but are most often the result of some user action (such as moving or clicking the mouse or entering text via the keyboard). Without the ability to react to these events, the World Wide Web’s greatest use might be limited to showing pictures of kittens.

Although HTML itself *does* define a small set of built-in semantic actions that require no script on our part (such as reloading pages as the result of clicking an anchor tag or submitting a form via a submit button), any other behaviors that we wish our pages to exhibit require us to handle the various events that occur as our users interact with those pages.

In this chapter, we examine the various ways that the browsers expose these events, how they allow us to establish handlers to control what happens when these events occur, and the challenges that we face due to the multitude of differences between the browser event models. We then see how jQuery cuts through the browser-induced fog to relieve us of these burdens.

Let’s start off by examining how browsers implement their events models.

JavaScript you need to know!

One of the great benefits that jQuery brings to web applications is the ability to implement a great deal of scripting-enabled behavior without having to write a whole lot of script ourselves. jQuery handles the nuts-and-bolts details so that we can concentrate on the job of making our applications do what it is that our applications need to do!

Up to this point, the ride has been almost free. You only needed rudimentary JavaScript skills to code and understand the jQuery examples we introduced in the previous chapters; in this chapter and the chapters that follow, you *must* understand a handful of fundamental JavaScript concepts to make effective use of the jQuery library.

Depending on your background, you may already be familiar with these concepts, but some page authors can write a lot of JavaScript without a firm grasp on exactly what's going on under the covers—the flexibility of JavaScript makes such a situation possible. Before we proceed, it's time to make sure that you've wrapped your head around these core concepts.

If you're already comfortable with the workings of the JavaScript `Object` and `Function` classes and have a good handle on concepts like *function contexts* and *closures*, you may want to continue reading this and the upcoming chapters. If these concepts are unfamiliar or hazy, we strongly urge you to turn to appendix A to help you get up to speed on these necessary concepts.

4.1 Understanding the browser event models

Long before anyone considered standardizing how browsers would handle events, Netscape Communications Corporation introduced an event-handling model in its Netscape Navigator browser; all modern browsers still support this model, which is probably the best understood and most employed by the majority of page authors.

This model is known by a few names. You may have heard it termed the Netscape Event Model, the Basic Event Model, or even the rather vague Browser Event Model; but most people have come to call it the DOM Level 0 Event Model.

NOTE The term *DOM Level* is used to indicate what level of requirements an implementation of the W3C DOM Specification meets. There isn't a DOM Level 0, but that term is used to informally describe what was implemented prior to the DOM Level 1.

The W3C didn't create a standardized model for event handling until DOM Level 2, introduced in November 2000. This model enjoys support from all modern standards-compliant browsers such as Firefox, Camino (as well as other Mozilla browsers), Safari, and Opera. Internet Explorer continues to go its own way and supports a subset of the DOM Level 2 Event Model functionality, albeit using a proprietary interface.

Before we see how jQuery makes that irritating fact a non-issue, let's spend time getting to know how the event models operate.

4.1.1 The DOM Level 0 Event Model

The DOM Level 0 Event Model is probably the event model that most web developers employ on their pages. In addition to being somewhat browser-independent, it's fairly easy to use.

Under this event model, event handlers are declared by assigning a reference to a function instance to properties of the DOM elements. These properties are defined to handle a specific event type; for example, a click event is handled by assigning a function to the `onclick` property, and a mouseover event by assigning a function to the `onmouseover` property of elements that support these event types.

The browsers allow us to specify the body of an event handler function as attribute values in the DOM elements' HTML, providing a shorthand for creating event handlers. An example of defining such handlers is shown in listing 4.1. This page can be found in the downloadable code for this book in the file `chapter4/dom.0.events.html`.

Listing 4.1 Declaring DOM Level 0 event handlers

```
<html>
<head>
  <title>DOM Level 0 Events Example</title>
  <script type="text/javascript"
    src="../../scripts/jquery-1.2.1.js">
  </script>
  <script type="text/javascript">
    $(function(){
      $('#vstar')[0].onmouseover = function(event) {
        say('Whee!');
      }
    });
    function say(text) {
      $('#console').append('<div>'+new Date()+ ' '+text+'</div>');
    }
  </script>
</head>
```

1 Ready handler defines mouseover handler

2 Utility function emits text to console

```

<body>
  
  <div id="console"></div>
</body>
</html>

```

3 ** element is instrumented**
4 **<div> element serves as console**

In this example, we employ both styles of event handler declaration: declaring under script control and declaring in a markup attribute.

The page first declares a ready handler **1** in which a reference to the image element with the `id` of `vstar` is obtained (using `jQuery`), and its `onmouseover` property is set to a function instance that we declare inline. This function becomes the event handler for the element when a mouseover event is triggered on it. Note that this function expects a single parameter to be passed to it. We'll learn more about this parameter shortly.

We also declare a small utility function, `say()` **2**, that we use to emit text messages to a `<div>` element on the page **4**. This will save us the trouble of nuisance alerts to indicate when things happen on our page.

In the body of the page (along with the console element), we define an image element **3** on which we'll define event handlers. We've already seen how to define one under script control in the ready handler **1**, but here we declare a handler for a click event using the `onclick` attribute of the `` element.

Loading this page into a browser (found in the file `chapter4/dom.0.events.html`), waving the mouse pointer over the image a few times, and then clicking the image result in a display similar to that shown in figure 4.1.

We declare the click event handler in the `` element markup using the following attribute:

```
onclick="say('Vroom vroom!');"
```

This might lead us to believe that the `say()` function becomes the click event handler for the element, but that's not the case. When handlers are declared via markup attributes, an anonymous function is automatically created using the value of the attribute as the function body. The action taken as a result of the attribute declaration is equivalent to (assuming that `imageElement` is a reference to the image element) the following:

```
imageElement.onclick = function(event) {
  say('Vroom vroom!');
}
```

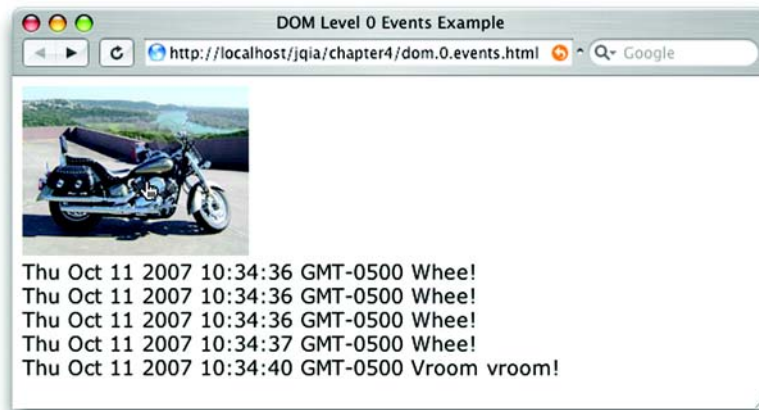


Figure 4.1 Waving the mouse over the image and clicking it result in the event handlers firing and emitting their messages to the console.

Note how the value of the attribute is used as the body of the generated function, and note that the function is created so that the event parameter is available within the generated function.

Before we move on to examining what that parameter is all about, we should note that using the attribute mechanism of declaring DOM Level 0 event handlers violates the precepts of Unobtrusive JavaScript that we explored in section 1.2. When using jQuery in our pages, we should adhere to the principles of Unobtrusive JavaScript and avoid mixing behavior defined by JavaScript with display markup. We'll see that jQuery provides a better way to declare event handlers than either of these means before the end of this chapter.

But first, let's examine what that event parameter is all about.

The Event instance

When an event handler is fired, an instance of a class named `Event` is passed to the handler as its first parameter in most browsers. Internet Explorer, always the life of the party, does things in its own proprietary way by tacking the `Event` instance onto a window property named `event`.

In order to deal with this discrepancy we'll often see the following used as the first statement in an event handler:

```
if (!event) event = window.event;
```

This levels the playing field by using object detection to check if the event parameter is undefined (or null) and assigning the value of the window's event property

to it if so. After this statement, the event parameter can be referenced regardless of how it was made available to the handler.

The properties of the `Event` instance provide a great deal of information regarding the event that has been fired and is currently being handled. This includes details such as which element the event was triggered on, the coordinates of mouse events, and which key was clicked for keyboard events.

But not so fast. Not only does Internet Explorer use a proprietary means to get the `Event` instance to the handler, but it also uses a proprietary definition of the `Event` class in place of the W3C-defined standard—we're not out of the object-detection woods yet.

For example, to get a reference to the target element—the element on which the event was triggered—we access the `target` property in standards-compliant browsers and the `srcElement` property in Internet Explorer. We deal with this inconsistency by employing object detection with a statement such as the following:

```
var target = (event.target) ? event.target : event.srcElement;
```

This statement tests if `event.target` is defined and, if so, assigns its value to the local `target` variable; otherwise, it assigns `event.srcElement`. We take similar steps for other `Event` properties of interest.

Up until this point, we've acted as if event handlers are only pertinent to the elements that serve as the trigger to an event—the image element of listing 4.1, for example. But events propagate throughout the DOM tree. Let's find out about that.

Event bubbling

When an event is triggered on an element in the DOM tree, the event-handling mechanism of the browser checks to see if a handler has been established for that particular event on that element and, if so, invokes it. But that's hardly the end of the story.

After the target element has had its chance to handle the event, the event model checks with the parent of that element to see if *it* has established a handler for the event type, and if so, it's also invoked—after which *its* parent is checked, then its parent, then its parent, and on and on, all the way up to the top of the DOM tree. Because the event handling propagates upward like the bubbles in a champagne flute (assuming we view the DOM tree with its root at the top), this process is termed *event bubbling*.

Let's modify the example of listing 4.1 so that we can see this process in action. Consider the code in listing 4.2.

Listing 4.2 Events propagating from the point of origin to the top of the DOM tree

```

<html id="greatgreatgrandpa">
  <head>
    <title>DOM Level 0 Events Example</title>
    <script type="text/javascript"
      src="../scripts/jquery-1.2.1.js">
    </script>
    <script type="text/javascript">
      $(function(){
        $('*').each(function(){
          var current = this;
          this.onclick = function(event) {
            if (!event) event = window.event;
            var target = (event.target) ?
              event.target : event.srcElement;
            say('For ' + current.tagName + '#' + current.id +
              ' target is ' + target.id);
          }
        });
      });

      function say(text) {
        $('#console').append('<div>' + text + '</div>');
      }
    </script>
  </head>

  <body id="greatgrandpa">
    <div id="grandpa">
      <div id="pops">
        
      </div>
    </div>
    <div id="console"></div>
  </body>
</html>

```

① Selects every element on the page

② Applies onclick handler to every selected element

We do a lot of interesting things in the changes to this page. First, we remove any handling for the mouseover event so that we can concentrate on the click event. We also embed the image element that will serve as the target for our event experiment in a couple of nested `<div>` elements to place the image element deeper within the DOM hierarchy. We also give almost every element in the page a specific and unique `id`—even the `<body>` and `<html>` tags!

We retain the console and its `say()` utility function for the same reporting purposes used in the previous example.

Now let's look at even more interesting changes.

In the ready handler for the page, we use jQuery to select all elements on the page and to iterate over each one with the `each()` method **1**. For each matched element, we record its instance in the local variable `current` and establish an `onclick` handler **2**. This handler first employs the browser-dependent tricks that we discussed in the previous section to locate the `Event` instance and identify the event target, and then emits a console message. This message is the most interesting part of this example.

It displays the tag name and `id` of the current element, putting *closures* to work (please read section A.2.4 in appendix A if closures are a subject that gives you heartburn), followed by the `id` of the target. By doing so, each message that's logged to the console displays the information about the current element of the bubble process, as well as the target element that started the whole shebang.

Loading the page (located in the file `chapter4/dom.0.propagation.html`) and clicking the image result in the display of figure 4.2.

This clearly illustrates that, when the event is fired, it's delivered first to the target element and then to each of its ancestors in turn, all the way up to the `<html>` element itself.

This is a powerful ability because it allows us to establish handlers on elements at any level to handle events occurring on its descendents. Consider a handler on a `<form>` element that reacts to any change event on its child elements to effect dynamic changes to the display based upon the elements' new values.

But what if we don't *want* the event to propagate? Can we stop it?



Figure 4.2 The console messages clearly show the propagation of the event as it bubbles up the DOM tree from the target element to the tree root.

Affecting event propagation and semantics

There may be occasions where we want to prevent an event from bubbling any further up the DOM tree. This might be because we're fastidious and we know that we've already accomplished any processing necessary to handle the event, or we may want to forestall unwanted handling that might occur higher up in the chain.

Regardless of the reason, we can prevent an event from propagating any higher via mechanisms provided on the `Event` instance. For standards-compliant browsers, we call the `stopPropagation()` method of the `Event` instance to halt the propagation of the event further up the ancestor hierarchy. In Internet Explorer, we set a property named `cancelBubble` to `true` in the `Event` instance. Interestingly, many modern standards-compliant browsers support the `cancelBubble` mechanism even though it's not part of any W3C standard.

Some events have default semantics associated with them. As examples, a click event on an anchor element will cause the browser to navigate to the element's `href`, and a submit event on a `<form>` element will cause the form to be submitted. Should we wish to cancel these semantics—sometimes termed the *default actions*—of the event, we set the return value for the handler to `false`.

A frequent use for such an action is in the realm of form validation. In the handler for the form's submit event, we can make validation checks on the form's `<input>` elements and return `false` if any problems with the data entry are detected.

We may also see the following on `<form>` elements:

```
<form name="myForm" onsubmit="return false;" ...
```

This effectively prevents the form from being submitted under any circumstances except under script control (via `form.submit()`, which doesn't trigger a submit event)—a common trick used in many Ajax applications where asynchronous requests will be made in lieu of form submissions.

Under the DOM Level 0 Event Model, almost every step we take in an event handler involves using browser-specific detection in order to figure out what action to take. What a headache! But don't put away the aspirin yet—it doesn't get any easier when we consider the more advanced event model.

4.1.2 The DOM Level 2 Event Model

One severe shortcoming of the DOM Level 0 Event Model is that, because a property is used to store a reference to a function that's to serve as an event handler, only one event handler per element can be registered for any specific event type

at a time. If we have two things that we want to do when an element is clicked, the following statements aren't going to let that happen:

```
someElement.onclick = doFirstThing;  
someElement.onclick = doSecondThing;
```

Because the second assignment replaces the previous value of the `onclick` property, only `doSecondThing` is invoked when the event is triggered. Sure, we could wrap both functions in another single function that calls both; but as pages get more complicated, as is highly likely in Rich Internet Applications, it becomes increasingly difficult to keep track of such things. Moreover, if we use multiple reusable components or libraries in a page, they may have no idea of the event-handling needs of the other components.

We could employ other solutions: implementing the Observable pattern that establishes a publish/subscribe scheme for the handlers, or even tricks using closures. But all of these add complexity to pages that are already complex enough.

Besides the establishment of a *standard* event model, the DOM Level 2 Event Model was designed to address these types of problems. Let's see how event handlers, even multiple handlers, are established on DOM elements under this more advanced model.

Establishing events

Rather than assigning a function reference to an element property, DOM Level 2 event handlers—also termed *listeners*—are established via an element *method*. Each DOM element defines a method named `addEventListener()` that's used to attach event handlers (listeners) to the element. The format of this method is as follows:

```
addEventListener(eventType, listener, useCapture)
```

The `eventType` parameter is a string that identifies the type of event to be handled. This string is, generally, the same event names we used in the DOM Level 0 Event Model without the *on* prefix: for example, `click`, `mouseover`, `keydown`, and so on.

The `listener` parameter is a reference to the function (or inline function) that's to be established as the handler for the named event type on the element. As in the basic event model, the `Event` instance is passed to this function as its first parameter.

The final parameter, `useCapture`, is a Boolean whose operation we'll explore in a few moments when we discuss event propagation in the Level 2 Model. For now, leave it set to `false`.

Let's once again change the example of listing 4.1 to use the more advanced event model. We'll concentrate only on the click event type; this time, we'll establish *three* click event handlers on the image element. The new example code can be found in the file `chapter4/dom.2.events.html` and is shown in listing 4.3.

Listing 4.3 Establishing event handlers with the DOM Level 2 Model

```
<html>
  <head>
    <title>DOM Level 2 Events Example</title>
    <script type="text/javascript"
      src="../../scripts/jquery-1.2.1.js">
    </script>
    <script type="text/javascript">
      $(function(){
        var element = $('#vstar')[0];
        element.addEventListener('click',function(event) {
          say('Whee once!');
        },false);
        element.addEventListener('click',function(event) {
          say('Whee twice!');
        },false);
        element.addEventListener('click',function(event) {
          say('Whee three times!');
        },false);
      });

      function say(text) {
        $('#console').append('<div>'+text+'</div>');
      }
    </script>
  </head>

  <body>
    
    <div id="console"></div>
  </body>
</html>
```

① Establishes three event handlers!

This code is simple but clearly shows how we have the ability to establish multiple event handlers on the same element for the same event type—something we were not able to do easily with the Basic Event Model. In the ready handler ① for the page, we grab a reference to the image element and then establish *three* event handlers for the click event.

Loading this page into a standards-compliant browser (*not* Internet Explorer) and clicking the image result in the display shown in figure 4.3.



Figure 4.3 Clicking the image once demonstrates that all three handlers established for the click event are triggered.

Note that even though the handlers fire in the order in which they were established, *this order isn't guaranteed by the standard!* Testers of this code never observed an order other than the order of establishment, but it would be foolish to write code that relies on this order. Always be aware that multiple handlers established on an element may fire in random order.

Now, what's up with that `useCapture` parameter?

Event propagation

We saw earlier that, with the Basic Event Model, once an event was triggered on an element the event propagated from the target element upwards in the DOM tree to all the target's ancestors. The advanced Level 2 Model also provides this bubbling phase but ups the ante with an additional phase: *capture phase*.

Under the DOM Level 2 Event Model, when an event is triggered, the event first propagates from the root of the DOM tree down to the target element and then propagates again from the target element up to the DOM root. The former phase (root to target) is called *capture phase*, and the latter (target to root) is called *bubble phase*.

When a function is established as an event handler, it can be flagged as a capture handler in which case it will be triggered during capture phase, or as a bubble handler to be triggered during bubble phase. As you might have guessed by this time, the `useCapture` parameter to `addEventListener()` identifies which type of handler is established. A value of `false` for this parameter establishes a bubble handler, whereas a value of `true` registers a capture handler.

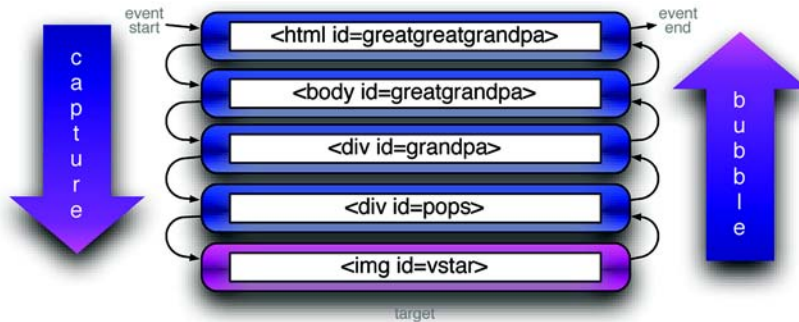


Figure 4.4 Propagation in the DOM Level 2 Event Model traverses the DOM hierarchy twice: once from top to target during capture phase and once from target to top during bubble phase.

Think back a moment to the example of listing 4.2 where we explored the propagation of the Basic Model events through a DOM hierarchy. In that example we embedded an image element within two layers of `<div>` elements. Within such a hierarchy, the propagation of a click event with the `` element as its target would move through the DOM tree as shown in figure 4.4.

Let's put that to the test, shall we? Listing 4.4 shows the code for a page containing the element hierarchy of figure 4.4 (chapter4/dom.2.propagation.html).

Listing 4.4 Tracking event propagation with bubble and capture handlers

```
<html id="greatgreatgrandpa">
  <head>
    <title>DOM Level 2 Propagation Example</title>
    <script type="text/javascript"
      src="../../scripts/jquery-1.2.1.js">
    </script>
    <script type="text/javascript">
      $(function(){
        $('*').each(function(){
          var current = this;
          this.addEventListener('click',function(event) {
            say('Capture for ' + current.tagName + '#' + current.id +
              ' target is ' + event.target.id);
          },true);
          this.addEventListener('click',function(event) {
            say('Bubble for ' + current.tagName + '#' + current.id +
              ' target is ' + event.target.id);
          },false);
        });
      });
    </script>
  </head>
  <body id="greatgrandpa">
    <div id="grandpa">
      <div id="pops">
        <img id="vstar" />
      </div>
    </div>
  </body>
</html>
```

Establishes listeners on all elements

```
        function say(text) {
            $('#console').append('<div>'+text+'</div>');
        }
    </script>
</head>

<body id="greatgrandpa">
    <div id="grandpa">
        <div id="pops">
            
        </div>
    </div>
    <div id="console"></div>
</body>
</html>
```

This code changes the example of listing 4.2 to use the DOM Level 2 Event Model API to establish the event handlers. In the ready handler ❶, we use jQuery’s powerful abilities to run through every element of the DOM tree. On each, we establish two handlers: one capture handler and one bubble handler. Each handler emits a message to the console identifying which type of handler it is, the current element, and the id of the target element.

With the page loaded into a standards-compliant browser, clicking the image results in the display in figure 4.5, showing the progression of the event through the handling phases and the DOM tree.

Note that, because we defined both capture and bubble handlers for the target, two handlers were executed for the target and all its ancestor nodes.

Well, now that we’ve gone through all the trouble to understand that, we should know that capture handlers are hardly ever used in web pages. The simple reason for that is that Internet Explorer (still inexplicably the most dominant browser) doesn’t support the DOM Level 2 Event Model. Although it *does* have a proprietary model corresponding to the bubble phase of the Level 2 standard, it doesn’t support any semblance of a capture phase.

Before we look at how jQuery is going to help sort all this mess out, let’s briefly examine the Internet Explorer Model.

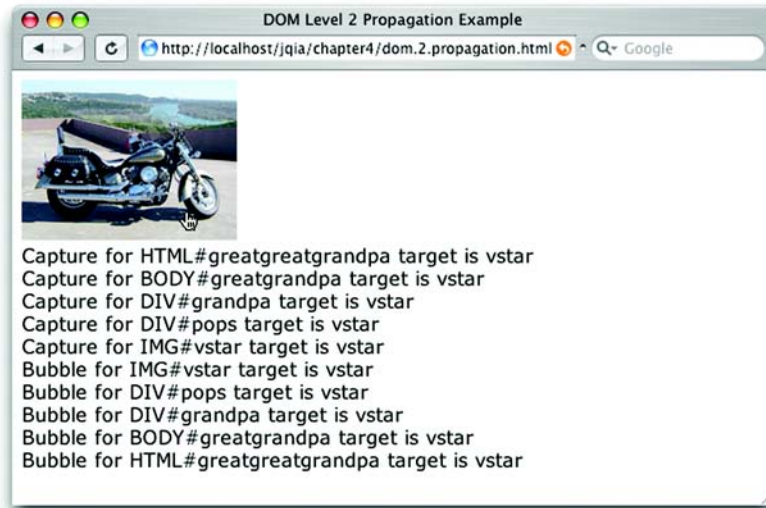


Figure 4.5 Clicking the image results in each handler emitting a console message that identifies the path of the event.

4.1.3 The Internet Explorer Event Model

Internet Explorer (both IE6 and, most disappointingly, IE7) doesn't provide support for the DOM Level 2 Event Model. Both these versions of Microsoft's browser provide a proprietary interface that closely resembles the bubble phase of the standard model.

Rather than `addEventListener()`, the Internet Explorer Model defines a method named `attachEvent()` for each DOM element. This method, as follows, accepts two parameters similar to those of the standard model:

```
attachEvent(eventName, handler)
```

The first parameter is a string that names the event type to be attached. The standard event names aren't used; the name of the corresponding element property from the DOM Level 0 Model is used—`onclick`, `onmouseover`, `onkeydown`, and so on.

The second parameter is the function to be established as the handler, and as in the Basic Model, the `Event` instance must be fetched from the window `.event` property.

What a mess! Even when using the relatively browser-independent DOM Level 0 Model, we're faced with a tangle of browser-dependent choices to make at each stage of event handling. And when using the more capable DOM Level 2 or Internet

Explorer Model, we even have to diverge our code when establishing the handlers in the first place.

Well, jQuery is going to make our lives simpler by hiding these browser disparities from us as much as it possibly can. Let's see how!

4.2 The jQuery Event Model

Although it's true that the creation of Rich Internet Applications requires a hefty reliance on event handling, the thought of writing event-handling code on a large scale while dealing with the browser differences is enough to daunt even the most intrepid of page authors.

We could hide the differences behind an API that abstracts the differences away from our page code, but why bother when jQuery has already done it for us?

jQuery's event implementation, which we'll refer to informally as the jQuery Event Model, exhibits the following features:

- Provides a unified method for establishing event handlers
- Allows multiple handlers for each event type on each element
- Uses standard event-type names: for example, click or mouseover
- Makes the `Event` instance available as a parameter to the handlers
- Normalizes the `Event` instance for the most often used properties
- Provides unified methods for event canceling and default action blocking

With the notable exception of support for a capture phase, the feature set of the jQuery Event Model closely resembles that of the Level 2 Model while supporting both standards-compliant browsers and Internet Explorer with a single API. The omission of capture phase should not be an issue for the vast majority of page authors who never use it (or even know it exists) due to its lack of support in IE.

Is it really that simple? Let's find out.

4.2.1 Binding event handlers using jQuery

Using the jQuery Event Model, we can establish event handlers on DOM elements with the `bind()` command. Consider the following simple example:

```
$('#img').bind('click',function(event){alert('Hi there!');});
```

This statement binds the supplied inline function as the click event handler for every image on a page. The full syntax of the `bind()` command is as follows:

Command syntax: bind**bind(eventType, data, listener)**

Establishes a function as the event handler for the specified event type on all elements in the matched set.

Parameters

- eventType** (String) Specifies the name of the event type for which the handler is to be established. This event type can be namespaced with a suffix separated from the event name with a period character. See the remainder of this section for details.
- data** (Object) Caller-supplied data that's attached to the `Event` instance as a property named `data` for availability to the handler functions. If omitted, the handler function can be specified as the second parameter.
- listener** (Function) The function that's to be established as the event handler.

Returns

The wrapped set.

Let's put `bind` into action. Taking the example of listing 4.3 and converting it from the DOM Level 2 Model to the jQuery Model, we end up with the code shown in listing 4.5 and found in the file `chapter4/jquery.events.html`.

Listing 4.5 Establishing event handlers without the need for browser-specific code

```
<html>
  <head>
    <title>DOM Level 2 Events Example</title>
    <script type="text/javascript"
      src="../scripts/jquery-1.2.1.js">
    </script>
    <script type="text/javascript">
      $(function(){
        $('#vstar')
          .bind('click',function(event) {
            say('Whee once!');
          })
          .bind('click',function(event) {
            say('Whee twice!');
          })
          .bind('click',function(event) {
            say('Whee three times!');
          });
      });

      function say(text) {
        $('#console').append('<div>'+text+'</div>');
      }
    </script>
  </head>
</html>
```

1 Binds three event handlers to the image


```
</script>
</head>

<body>
  
  <div id="console"></div>
</body>
</html>
```

The changes to this code, limited to the body of the ready handler, are minor but significant ❶. We create a wrapped set consisting of the target `` element and apply three `bind()` commands to it—remember, jQuery chaining lets us apply multiple commands in a single statement—each of which establishes a click event handler on the element.

Loading the page into a standards-compliant browser and clicking the image result in the display of figure 4.6, which not surprisingly, is the exact same result we saw in figure 4.3 (except for the URL and window caption).

Perhaps more importantly, when loaded into Internet Explorer, it also works as shown in figure 4.7. This was not possible using the code from listing 4.3 without adding a lot of browser-specific testing and branching code to use the correct event model for the current browser.

At this point, page authors who have wrestled with mountains of browser-specific event-handling code in their pages are no doubt singing “Happy Days Are Here Again” and spinning in their office chairs. Who could blame them?

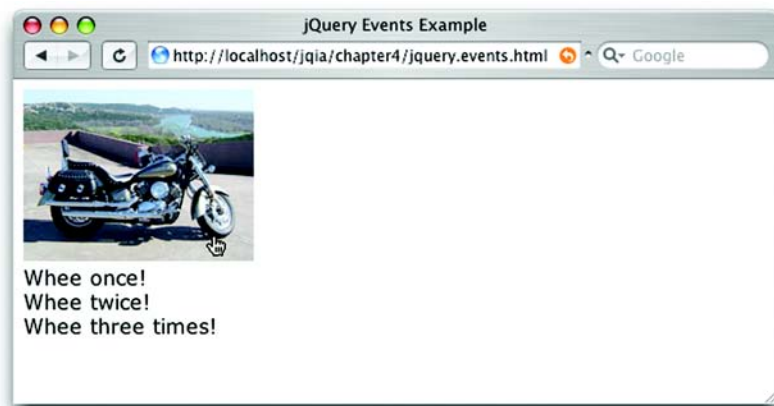


Figure 4.6 Using the jQuery Event Model allows us to specify multiple event handlers as in the DOM Level 2 Model.

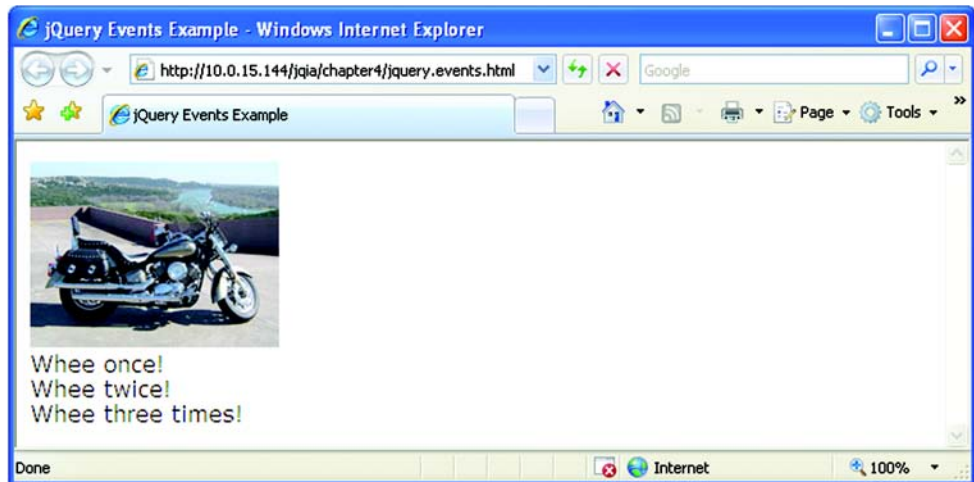


Figure 4.7 The jQuery Event Model allows us to use a unified code base to support events in Internet Explorer.

Another little nifty event handling extra that jQuery provides for us is the ability to group event handlers by assigning them to a namespace. Unlike conventional namespacing (which assigns namespaces via a prefix), the event names are namespaced by adding a *suffix* to the event name separated by a period character.

By grouping event bindings in this way, we can easily act upon them later as a unit.

Take, for example, a page that has two modes: a display mode and an edit mode. When in edit mode, event listeners are placed on many of the page elements, but these listeners are not appropriate for display mode and need to be removed when the page transitions out of edit mode. We could namespace the edit mode events with code such as

```
$('#thing1').bind('click.editMode',someListener);
$('#thing2').bind('click.editMode',someOtherListener);
...
$('#thingN').bind('click.editMode',stillAnotherListener);
```

By grouping all these bindings into a namespace named `editMode`, we can later operate upon them as a whole. For example, when the page leaves edit mode and it comes time to remove all the bindings we could do this easily with

```
$('.*).unbind('click.editMode');
```

This will remove all `click` bindings (the explanation of the `unbind()` method is coming up in the next section) in the namespace `editMode` for all elements on the page.

In addition to the `bind()` command, jQuery provides a handful of shortcut commands to establish specific event handlers. Because the syntax of each of these commands is identical except for the method name of the command, we'll save some space and present them all in the following single syntax descriptor:

Command syntax: *specific event binding*

`eventTypeName(listener)`

Establishes the specified function as the event handler for the event type named by the method's name. The supported commands are as follows:

- | | | | |
|------------|------------|-------------|----------|
| ▪ blur | ▪ focus | ▪ mousedown | ▪ resize |
| ▪ change | ▪ keydown | ▪ mousemove | ▪ scroll |
| ▪ click | ▪ keypress | ▪ mouseout | ▪ select |
| ▪ dblclick | ▪ keyup | ▪ mouseover | ▪ submit |
| ▪ error | ▪ load | ▪ mouseup | ▪ unload |

Note that when using these shortcut methods, we cannot specify a data value to be placed in the `event.data` property.

Parameters

`listener` (Function) The function that's to be established as the event handler.

Returns

The wrapped set.

jQuery also provides a specialized version of the `bind()` command, named `one()`, that establishes an event handler as a one-shot deal. Once the event handler executes the first time, it's automatically removed as an event handler. Its syntax is similar to the `bind()` command and is as follows:

Command syntax: *one*

`one(eventType, data, listener)`

Establishes a function as the event handler for the specified event type on all elements in the matched set. Once executed, the handler is automatically removed.

Parameters

- `eventType` (String) Specifies the name of the event type for which the handler is to be established.
- `data` (Object) Caller-supplied data that's attached to the `Event` instance for availability to the handler functions. If omitted, the handler function can be specified as the second parameter.
- `listener` (Function) The function that's to be established as the event handler.

Returns

The wrapped set.

These commands give us many choices to bind an event handler to matched elements. And once a handler is bound, we may eventually need to remove it.

4.2.2 Removing event handlers

Typically, once an event handler is established, it remains in effect for the remainder of the life of the page. Particular interactions may dictate that handlers be removed based on certain criteria. Consider, for example, a page where multiple steps are presented, and once a step has been completed, its controls revert to read-only.

For such cases, it would be advantageous to remove event handlers under script control. We've seen that the `one()` command can automatically remove a handler after it has completed its first (and only) execution, but for the more general case where we'd like to remove event handlers under our own control, jQuery provides the `unbind()` command.

The syntax of `unbind()` is as follows:

Command syntax: `unbind`

`unbind(eventType, listener)`

`unbind(event)`

Removes events handlers from all elements of the wrapped set as specified by the optional passed parameters. If no parameters are provided, all listeners are removed from the elements.

Parameters

- `eventType` (String) If provided, specifies that only listeners established for the specified event type are to be removed.
- `listener` (Function) If provided, identifies the specific listener that's to be removed.
- `event` (Event) Removes the listener that triggered the event described by this Event instance.

Returns

The wrapped set.

This command can be used to remove event handlers from the elements of the matched set at various levels of granularity. All listeners can be removed by omitting parameters, or listeners of a specific type can be removed by providing that event type.

Specific handlers can be removed by providing a reference to the function originally established as the listener. For this to be possible, a reference to the function must be retained when binding the function as an event listener in the first place. For this reason, when a function that's eventually to be removed as a handler is

originally established as a listener, it's either defined as a top-level function (so that it can be referred to by its top-level variable name) or a reference to it is retained by some other means. Supplying the function as an anonymous inline reference would make it impossible to later reference the function in a call to `unbind()`.

So far, we've seen that the jQuery Event Model makes it easy to establish (as well as remove) event handlers without worries about browser differences, but what about writing the event handlers themselves?

4.2.3 Inspecting the Event instance

When an event handler established with the `bind()` command is invoked, the `Event` instance is passed to it as the first parameter to the function. This eliminates the need to worry about the `window.event` property under Internet Explorer, but what about accessing the divergent properties of the `Event` instance?

Even when using jQuery to establish handlers, the `Event` instance passed to the event handler is a clone of the native object as defined by the browser. That means that in standards-compliant browsers, the `Event` instance will follow the standardized layout of properties, and under Internet Explorer, the instance will use the proprietary layout. Before the proprietary instance is passed to the event handler, jQuery does its best to fix up the object so that the most commonly accessed properties and methods of that object follow the standardized format. So once again, except for the most obscure of `Event` properties, we can write the code for our event handlers without regard for browser platform.

Table 4.1 shows the `Event` properties that are safe to access in a platform-independent manner.

Table 4.1 Safe `Event` instance properties

Property	Description
<code>altKey</code>	Set to <code>true</code> if the Alt key was pressed when the event was triggered, <code>false</code> if not. The Alt key is labeled Option on most Mac keyboards.
<code>ctrlKey</code>	Set to <code>true</code> if the Ctrl key was pressed when the event was triggered, <code>false</code> if not.
<code>data</code>	The value, if any, passed as the second parameter to the <code>bind()</code> command when the handler was established.
<code>keyCode</code>	For <code>keyup</code> and <code>keydown</code> events, this returns the key that was pressed. Note that for alphabetic characters, the uppercase version of the letter will be returned, regardless of whether the user typed an uppercase or lowercase letter. For example, both <code>a</code> and <code>A</code> will return 65. You can use <code>shiftKey</code> to determine which case was entered. For key-press events, use the <code>which</code> property, which is reliable across browsers.

continued on next page

Table 4.1 Safe Event instance properties (continued)

Property	Description
<code>metaKey</code>	Set to <code>true</code> if the Meta key was pressed when the event was triggered, <code>false</code> if not. The Meta key is the Ctrl key on PCs and the Command key on Macs.
<code>pageX</code>	For mouse events, specifies the horizontal coordinate of the event relative from the page origin.
<code>pageY</code>	For mouse events, specifies the vertical coordinate of the event relative from the page origin.
<code>relatedTarget</code>	For some mouse events, identifies the element that the cursor left or entered when the event was triggered.
<code>screenX</code>	For mouse events, specifies the horizontal coordinate of the event relative from the screen origin.
<code>screenY</code>	For mouse events, specifies the vertical coordinate of the event relative from the screen origin.
<code>shiftKey</code>	Set to <code>true</code> if the Shift key was pressed when the event was triggered, <code>false</code> if not.
<code>target</code>	Identifies the element for which the event was triggered.
<code>type</code>	For all events, specifies the type of event that was triggered (for example, <code>click</code>). This can be useful if you're using one event handler function for multiple events.
<code>which</code>	For keyboard events, specifies the numeric code for the key that caused the event, and for mouse events, specifies which button was pressed (1 for left, 2 for middle, 3 for right). This should be used instead of <code>button</code> , which can't be relied on to function consistently across browsers.

Importantly, the `keypress` property isn't reliable cross-browser for non-alphabetic characters. For instance, the left arrow key has a code of 37, which works reliably on `keyup` and `keydown` events. Safari returns nonstandard results for these keys on a `keypress` event.

We can get a reliable, case-sensitive character code in the `which` property of `keypress` events. During `keyup` and `keydown` events, we can only get a case-insensitive key code (so `a` and `A` both return 65), but we can determine case by checking `shiftKey`.

The `Event` instance contains not only properties that give us information regarding the event that's handled, but also possesses a handful of methods that lets us control the propagation of the event. Let's dig into those.

4.2.4 Affecting the event propagation

In addition to standardizing the most-used properties of the `Event` instance, jQuery provides the same benefit for the standard methods used to affect event propagation.

The `stopPropagation()` method will prevent the event from bubbling further up the DOM tree (if needed, refer back to figure 4.4 for a reminder of how events propagate), and the `preventDefault()` method will cancel any semantic action that the event might cause. Some examples of such semantic actions are link traversal for `<a>` elements, forms submissions, and toggling the state of check boxes on a click event.

If we want to both stop the propagation of the event, as well as cancel its default behavior, we can return `false` as the return value of the listener function.

In addition to allowing us to set up event handling in a browser-independent manner, jQuery provides a set of commands that gives us the ability to trigger event handlers under script control. Let's look at those.

4.2.5 Triggering event handlers

Event handlers are designed to be invoked when their associated event triggers the propagation of the event through the DOM hierarchy. But there may be times when we want to trigger the execution of a handler under script control. We could define such event handlers as top-level functions so that we can invoke them by name, but as we've seen, defining event handlers as inline anonymous functions is much more common and so darned convenient!

jQuery has provided means to assist us in avoiding top-level functions by defining a series of methods that will automatically trigger event handlers on our behalf under script control. The most general of these commands is `trigger()`, whose syntax is as follows:

Command syntax: `trigger`

`trigger(eventType)`

Invokes any event handlers established for the passed event type for all matched elements

Parameters

`eventType` (String) Specifies the name of the event type for handlers which are to be invoked

Returns

The wrapped set

Note that the `trigger()` command (as well as the convenience commands that we'll introduce in a moment) does not cause an event to be triggered and to propagate through the DOM hierarchy. As there's no dependable cross-browser means to generate an event, jQuery calls the handlers as normal functions.

Each handler called is passed a minimally populated instance of `Event`. Because there's no event, properties that report values, such as the location of a mouse event, have no value. The `target` property is set to reference the element of the matched set to which the handler was bound.

Also because there's no event, no event propagation takes place. The handlers bound to the matched elements will be called, but no handlers on the ancestors of those elements will be invoked. Remember, these commands are convenient ways to call an event handler, not to try and emulate an event.

In addition to the `trigger()` command, jQuery provides convenience commands for most of the event types. The syntax for all these commands is exactly the same except for the command name, and that syntax is described as follows:

Command syntax: *eventName*

eventName ()

Invokes any event handlers established for the named event type for all matched elements. The supported commands are as follows:

- blur
- click
- focus
- select
- submit

Parameters

none

Returns

The wrapped set.

In addition to binding, unbinding, and triggering event handlers, jQuery offers high-level functions that further make dealing with events on our pages as easy as possible.

4.2.6 Other event-related commands

There are often interaction styles that are commonly applied to pages in Rich Internet Applications and are implemented using combinations of behaviors.

jQuery provides a few event-related convenience commands that make it easier to use these interaction behaviors on our pages. Let's look at them.

Toggleing listeners

The first of these is the `toggle()` command, which establishes a pair of click event handlers that swap off with each other on every other click event. Its syntax is as follows:

Command syntax: toggle

`toggle(listenerOdd, listenerEven)`

Establishes the passed functions as click event handlers on all elements of the wrapped set that toggle between each other with every other trigger of a click event

Parameters

- `listenerOdd` (Function) A function that serves as the click event handler for all odd-numbered clicks (the first, the third, the fifth, and so on)
- `listenerEven` (Function) A function that serves as the click event handler for all even-numbered clicks (the second, the fourth, the sixth, and so on)

Returns

The wrapped set

A common use for this convenience command is to toggle the enabled state of an element based upon how many times it has been clicked. We can simulate this using the image element of our previous examples, changing its opacity to reflect whether it's enabled (fully opaque) or disabled (partially transparent). We could do this example for real by toggling the read-only state of a text input control, but that would not make as clear a visual statement for demonstration purposes. Let's fake it with the image example.

Consider figure 4.8, which shows a page containing the image in a time-lapse display of the page in three states:

- 1 On initial display
- 2 After clicking the image once
- 3 After clicking the image again

The code for this example is shown in listing 4.6 and can be found in the file `chapter4/toggle.html`.

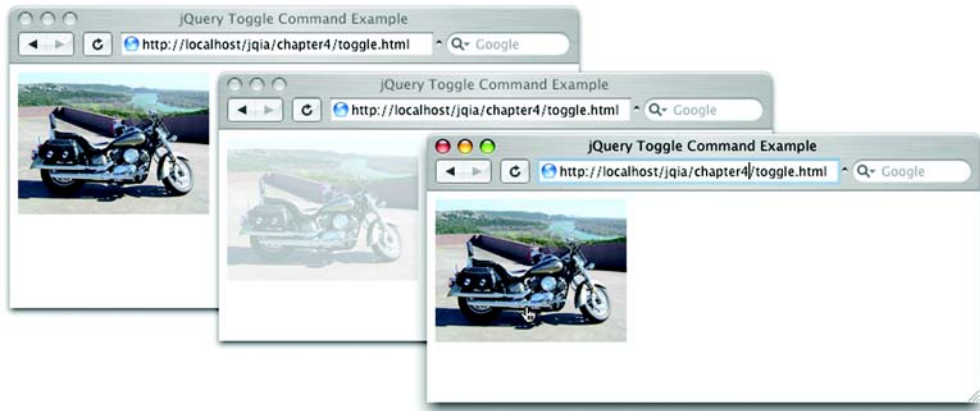


Figure 4.8 The `toggle()` command makes it easy to toggle the visual state of the image.

Listing 4.6 Invoking complementary listeners on every other click event

```

<html>
  <head>
    <title>jQuery Toggle Command Example</title>
    <script type="text/javascript"
      src="../scripts/jquery-1.2.1.js">
    </script>
    <script type="text/javascript">
      $(function(){
        $('#vstar').toggle(
          function(event) {
            $(event.target)
              .css('opacity',0.4);
          },
          function(event) {
            $(event.target)
              .css('opacity',1.0);
          }
        );
      });
    </script>
  </head>

  <body>
    
  </body>
</html>

```

- 1 Applies `toggle()` command to image
- 2 Odd listener grays out the image
- 3 Even listener restores full opacity

In this example, we apply the `toggle()` command ❶ to the images supplying an odd listener ❷ that reduces the opacity value to 0.4 (*graying out* the image, a common term for indicating disablement) and an even listener that restores the opacity to its full value of 1.0 ❸. Because the `toggle()` command handles all the swapping out for us, we don't need to bother keeping track of whether the current click is odd or even. How convenient.

All we accomplished in this example was the toggling of the image from full to partial opacity, but it's easy to imagine supplying listeners that would toggle any complementary states: enabled versus disabled, for example.

Another common multi-event scenario that's frequently employed in Rich Internet Applications involves mousing into and out of elements.

Hovering over elements

Events that inform us when the mouse pointer has entered an area, as well as when it has left that area, are essential to building many of the user interface elements that are commonly presented to users on our pages. Among these element types, the menus used as navigation systems for web applications are a common example.

A vexing behavior of the `mouseover` and `mouseout` event types often hinders the easy creation of such elements when a `mouseout` event fires as the mouse is moved over an area and its children. Consider the display in figure 4.9 (available in the file `chapter4/hover.html`).

This page displays two identical (except for naming) sets of areas: an outer area and an inner area. Load this page into your browser as you follow the rest of this section.

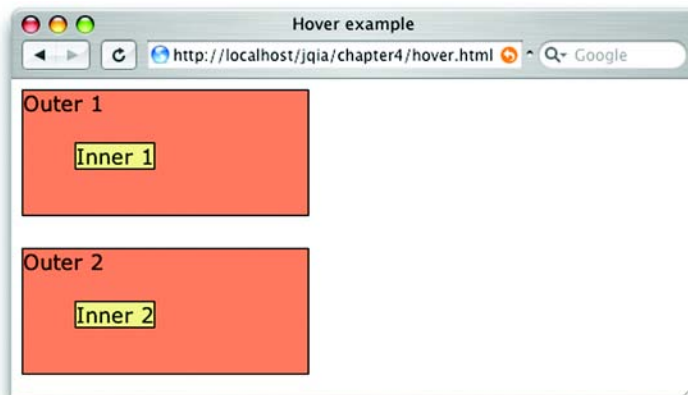


Figure 4.9
This page helps demonstrate when mouse events fire as the mouse pointer is moved over an area and its children.

For the top set, the following script in the ready handler establishes handlers for the mouse events:

```
$('#outer1')
  .bind('mouseover',report)
  .bind('mouseout',report);
```

This statement establishes a function named `report` as the event handler for both the `mouseover` and `mouseout` events. The `report()` function is defined as follows:

```
function report(event) {
  $('#console').append('<div>'+event.type+'</div>');
}
```

This listener merely adds a `<div>` element containing the name of the event that fired to a `<div>` named `console` that's defined at the bottom of the page, allowing us to see when each event fires.

Now, let's move the mouse pointer into the area labeled Outer 1 (being careful not to enter Inner 1). We'll see (from looking at the bottom of the page) that a `mouseover` event has fired. Move the pointer back out of the area. As expected, we'll see that a `mouseout` event has fired.

Let's refresh the page to start over, clearing the console.

Now, move the mouse pointer into Outer 1 (noting the event), but this time continue inward until the pointer enters Inner 1. As the mouse enters Inner 1, a `mouseout` event fires for Outer 1. If we wave our pointer over the inner area, we'll see a flurry of `mouseout` and `mouseover` events. This *is* the expected behavior. Even though the pointer is still within the bounds of Outer 1, when the pointer enters a contained element, the event model considers the transition from the area of Outer 1 for its contained element to be leaving the outer area.

Expected or not, we don't always want that behavior. Often, we want to be informed when the pointer leaves the bounds of the outer area and don't care whether the pointer is over a contained area or not.

We could write our handlers to detect when a mouse event is the result of leaving the area or the result of merely entering a contained element, but luckily we won't have to. jQuery comes to our aid with the `hover()` command.

The syntax of this command is as follows:

Command syntax: hover**hover**(overListener, outListener)

Establishes handlers for the mouseover and mouseout events for matched elements. These handlers only fire when the area covered by the elements is entered and exited, ignoring transitions to child elements.

Parameters

overListener (Function) The function to become the mouseover handler.

outListener (Function) The function to become the mouseout handler.

Returns

The wrapped set.

We use the following script to establish mouse event handlers for the second set of areas (Outer 2 and its Inner 2 child) on the hover.html example page:

```
$('#outer2').hover(report, report);
```

As with the first set of areas, the `report()` function is established as the mouseover and mouseout handlers for Outer 2. But unlike the first set of areas, when we pass the mouse pointer over the boundaries between Outer 2 and Inner 2, neither of these handlers is invoked. This is useful for those situations where we have no need to react when the mouse pointer passes over child elements.

With all these event-handling tools under our belts, let's use what we've learned so far and look at an example page that makes use of them, as well as some of the other jQuery techniques that we've learned from previous chapters!

4.3 Putting events (and more) to work

Now that we've covered how jQuery brings order to the chaos of dealing with disparate event models across browsers, let's work an example page that puts the knowledge that we've gained so far to use. This page uses not only events but also some jQuery techniques that we've explored in earlier chapters, including some heavy-weight jQuery selectors.

The page that we'll create in this section is a small part of an online order form for an Asian restaurant named Bamboo Asian Grille. For brevity, we're going to restrict ourselves to the appetizer section of the menu, but you can apply the lessons learned to the remainder of the menu form, which you can complete to practice your jQuery skills.

The goal for this example seems simple: to allow users to select the type and number of appetizers they would like added to their order. No problem, right? A series of check boxes and text boxes will do nicely as the expected GUI element for making multiple choices and specifying the quantities.

But there's a small catch: for each appetizer, other options must be presented. For example, when ordering Crab Rangoon, diners can choose sweet-and-sour sauce, hot mustard, or (for those who can't decide) both. Again, this shouldn't be a problem because we can associate a series of radio buttons representing the options with each appetizer entry.

But as it turns out, this *does* lead to a small problem. With a little HTML coding and some CSS magic, we create the layout shown in figure 4.10.

Even with only five appetizer choices and their corresponding options, the number of controls is overwhelming; it may even be difficult to see the choices the diner has made so far. The form works as required, but its usability leaves much to be desired.

We can solve this usability dilemma by applying a principle known as *progressive disclosure*. We don't need to present options for an appetizer the user isn't ordering, so we'll hide the radio button options until the user needs to see them.

Progressively disclosing information as it's needed will vastly improve the usability of the form by reducing the confusing clutter, as shown in figure 4.11.

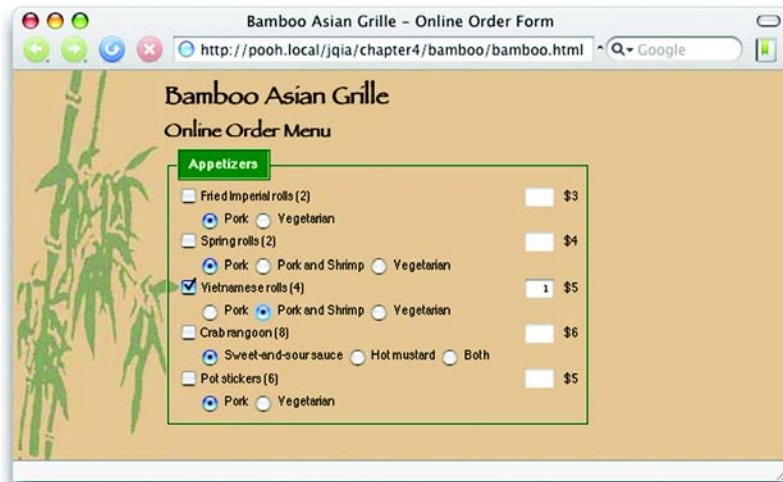


Figure 4.10 All our appetizers and options are displayed, but the screen is a jumbled mess!



Figure 4.11 By hiding the options until they're needed, we reduce the confusion and clutter, and the user isn't overwhelmed by controls that aren't relevant.

As a bonus, we'll also instrument the controls so that when a quantity is entered by the hungry user, the displayed dollar amount will reflect the price for the quantity chosen. Let's see what it takes to make all that happen.

The full page code for this example is available in the file `chapter4/bamboo/bamboo.html`, but let's start by examining the structure of the HTML that's used to implement the display of *one* appetizer entry as shown in listing 4.7.

Listing 4.7 HTML structure for a single appetizer entry

```

<div>
  <label>
    <input type="checkbox" name="appetizers"
      value="imperial" />
    Fried Imperials rolls (2)
  </label>
  <span price="3">
    <input type="text" name="imperial.quantity"
      disabled="disabled" value="1" />
    $<span></span>
  </span>
  <div>
    <input type="radio" name="imperial.option"
      value="pork" checked="checked" />
    Pork
  </div>
</div>

```

1 Label construct contains control
2 Uses custom attribute to hold price data
3 Holds place for computed price
4 Contains options to be conditionally displayed

```
<input type="radio" name="imperial.option"
      value="vegetarian"/>
  Vegetarian
</label>
</div>
</div>
```

We repeat this HTML structure for each appetizer entry. Note that this snippet contains no visual rendition information; such information is factored out to CSS definitions (which can be found in the file `bamboo.css` in the same folder as the HTML file).

Similarly, note that there's no script embedded within the HTML markup. The behavior of our page will be defined following the principles of Unobtrusive JavaScript with all script properly sequestered from the HTML markup.

We should emphasize some aspects of this structure because they will become important when adding the behaviors to these elements. First, note that the check box elements (as well as the radio elements further into the markup) are contained within `<label>` elements ❶ that also hold the text that corresponds to the controls. This makes clicking the text of the label flip the checked state of the contained control as if the user clicked the control itself. This is a handy usability enhancement that makes it easier for people to use our pages (especially for so-called *sloppy clickers*, a group to which at least one of your authors belongs).

Another notable feature is the `` element that contains the quantity text box and price display ❷. This element is adorned with a custom attribute named `price` that we use to store the price value for the appetizer. We'll need this value to calculate the price when the quantity is entered, and the attribute will also serve as a useful selector handle in our jQuery selectors. (The use of custom attributes in this fashion is considered controversial by some; please see the sidebar for more information.)

Note also that the `` element created to contain the computed amount is initially empty ❸. We could just fill it in statically, but that means we'd have price information in two places—generally considered not a best practice. Later, we'll see how we fill this in as part of the setup behavior of the page.

The final major construct in our markup is the `<div>` element ❹ that contains the radio buttons representing the appetizer options. This is the element that will be hidden until an appetizer is checked.

With the markup all settled, let's develop the behavior of the page step by step, starting with hiding the container element for the radio button options.

Custom attributes: heroic or heinous?

The use of custom attributes to adorn DOM elements with attributes not defined by the HTML or XHTML Specifications is a practice that has both its supporters and detractors. To some, it's a useful leveraging of the tools that HTML and the browsers make available to us; to others, it's an affront to all that is good because using custom attributes can prevent the page from passing validation testing.

Your authors take no sides on this issue and leave it to you, the reader, to determine whether you think that using custom attributes is a useful mechanism or a wart on the face of a page.

Without the use of the attribute, the price data could be stored in a JavaScript variable containing an object hash that associates an appetizer name (`imperial`, for example) with its price.

The custom attribute tactic can be said to be advantageous over the JavaScript variable mechanism because adding new appetizer entries means adding a new, self-contained construct to the page without having to remember to update the object hash with the price of added appetizers.

Again, we leave it to you to determine which approach you feel suits you best.

Inspecting the HTML structure of each appetizer entry allows us to concoct a selector that matches the `<div>` elements and to use the `hide()` command on them as follows:

```
$('.fieldset div div').hide();
```

NOTE We could initially hide these elements with CSS, but doing so in script ensures that users who turn off JavaScript (yes, there are still people who do that) will get a usable interface, albeit at the price of some computation load when the page is displayed. There are other reasons to do the hiding in the ready handler that we'll discuss in chapter 5 when we examine wrapper methods like `hide()` in greater detail.

Having tucked the appetizer options away for later display, we now turn our attention to enabling the usability behaviors that we want the elements to exhibit. Let's start by tying the display of the radio button options associated with an appetizer to whether that appetizer is checked or not.

To react to a change in the state of an appetizer check box, which should trigger the change in visibility of the `<div>` element containing its options, we establish

a listener for a click event on the check boxes in which we can adjust the visibility of the options based upon the state of the check box. Let's look at the following statement, which establishes this listener:

```
$('.checkbox').click(function(){
  var checked = this.checked;
  $('div',$('this').parents('div:first'))
  .css('display',checked ? 'block':'none');
  $('input[type=text]',$('this').parents('div:first'))
  .attr('disabled',!checked)
  .css('color',checked ? 'black' : '#f0f0f0')
  .val(1)
  .change()
  .each(function(){ if (checked) this.focus();});
});
```

All *that* just to hide and show a <div>?

Well, no. Hiding and showing the options is just one of the things that we need to do when the state of one of the check boxes changes. Let's look at each step in this fragment to see what it does for us.

First, the click handler stores the checked state of the check box in a variable named `checked`. This makes for easy reference in the code, and it establishes a local variable that we can use in any closures that we create.

Next, the handler locates the <div> containing the appetizer options and sets its CSS `display` value to hide those options when the check box is unchecked or to show them when the check box is checked. The jQuery expression that we use to locate the element to be shown or hidden is `$('div',$('this').parents('div:first'))`, which equates to “find the <div> elements in the first ancestor element of `this` that is a <div>.” Because we know from our HTML structure that there will be only one match, we don't need to differentiate any further.

Being astute, you'll have noted that, because we know the initial state of the check box is unchecked and the options are hidden, we could have written less code by using the `toggle()` command without needing to query the state of the check box. That type of assumptive code can be fragile and break easily when its assumptions change, so it's generally better to make explicitly certain that the visibility of our options matches the state of their respective check boxes.

Our handler isn't done yet; it still needs to adjust the state of the quantity text boxes. These boxes are initially disabled and will only be enabled when an appetizer is checked. We locate the text box with `$('input[type=text]',$('this').parents('div:first'))`, a similar selector to the one we just employed that says “locate the <input> element of type `text` in my first parent <div>.”

To this element, we do the following:

- Use the `attr()` command to set its disabled state to correspond to the inverse of the check box state.
- Apply a CSS `color` value so that the text is invisible when the control is disabled. (Note that this doesn't work on all browsers—some like Opera and Internet Explorer don't allow us to override the color of disabled fields.)
- Set the value to 1. If we're enabling the control, this is the default value we want to use; when disabling the field, we want to revert to this default.
- Call the change handler of the text box (which we haven't defined yet, but don't worry because that's next). This change handler will compute the price for the appetizer and display it. Because we changed the value under the covers (to 1), we need to call this handler to ensure that the price display is accurate.
- Employ the `each()` method to obtain a reference to the element and assign focus to that element if the check box is in checked state. Don't you just love closures that give us access to the checked local variable?

NOTE When pondering what type of event to handle for the check boxes, you may initially have thought of capturing change events as opposed to click events. For our scheme to work, we need to be immediately notified when the state of a check box changes. Immediate notification occurs within Safari and the Mozilla-based browsers, but Internet Explorer does not trigger change events until *after* focus has been blurred from the control, making the change event unsuitable for this use; instead, we rely on the click event.

Now let's turn our attention to the change handler for the text box. When the value in the text box changes, we want to recompute and display the cost of the appetizer—a simple calculation made by multiplying the price of one appetizer by the quantity.

The statement to add the change handler is as follows:

```
$('#span[price] input[type=text]').change(function(){
    $('#~ span:first',this).text(
        $(this).val() *
        $(this).parents("span[price]:first").attr('price')
    );
});
```

After locating the text boxes (with a selector that reads “find all `<input>` elements of type `text` that are within `` elements possessing a `price` attribute”), we assign a change handler that finds the `` to be updated and sets its text content to the computed value; the expression `$('#~ span:first',this)` locates the first sibling of `this` that’s a `` element. The computation is made by obtaining the value of the text box and multiplying it by the value of the `price` attribute on the parent ``.

If any of these rather advanced selector expressions has you scratching your head, it might be a good time to review the selector syntax presented in chapter 2.

Before we let the user interact with our page, we have one more thing that we need to do. Remember how we left the `` elements that are to contain the computed values blank? Now it’s time to fill those in.

The values of the quantity text boxes were preset to 1, so all we need to do is to perform the same computation that occurs when the values are changed. But we don’t want to repeat any code so we trigger the change handler on the text boxes and let that change handler do its thing.

```
$('#span[price] input[type=text]').change();
```

With that, we’ve completed whipping up the appetizer order form—at least to the point where we’ve met our stated goals. This example exposed us to some very important lessons:

- It showed us how to establish click and change handlers on elements that can be used to effect whatever user interface changes we want when triggered.
- We saw how to trigger handlers under script control to avoid both repeated code and the need to factor common code out into global named functions.
- We were exposed to some mighty fancy selectors used to pick and choose which elements we wanted to perform operations on.

The complete code for the page is shown in listing 4.8.

Listing 4.8 Complete code for the appetizer order form

```
<html>
<head>
  <title>Bamboo Asian Grille - Online Order Form</title>
  <link rel="stylesheet" type="text/css" href="bamboo.css">
  <script type="text/javascript"
    src="../../scripts/jquery-1.2.1.js"></script>
  <script type="text/javascript">
    $(function(){
```

```

$('fieldset div div').hide();
$(':checkbox').click(function(){
    var checked = this.checked;
    $('div',$$(this).parents('div:first'))
        .css('display',checked ? 'block':'none');
    $('input[type=text'],$$(this).parents('div:first'))
        .attr('disabled',!checked)
        .css('color',checked ? 'black' : '#f0f0f0')
        .val(1)
        .change()
        .each(function(){ if (checked) this.focus();});
});
$('span[price] input[type=text]').change(function(){
    $('~ span:first',this).text(
        $(this).val() *
        $(this).parents("span[price]:first").attr('price')
    );
});
$('span[price] input[type=text]').change();
});
</script>
</head>

<body>
<h1>Bamboo Asian Grille</h1>
<h2>Online Order Menu</h2>
<fieldset>
<legend>Appetizers</legend>

<div>
<label>
<input type="checkbox" name="appetizers"
    value="imperial"/>
    Fried Imperials rolls (2)
</label>
<span price="3">
<input type="text" name="imperial.quantity"
    disabled="disabled" value="1"/>
    $<span></span>
</span>
<div>
<label>
<input type="radio" name="imperial.option"
    value="pork" checked="checked"/>
    Pork
</label>
<label>
<input type="radio" name="imperial.option"
    value="vegetarian"/>
    Vegetarian
</label>

```

```
</div>
</div>

<div>
  <label>
    <input type="checkbox" name="appetizers" value="spring"/>
    Spring rolls (2)
  </label>
  <span price="4">
    <input type="text" name="spring.quantity"
      disabled="disabled" value="1"/>
    $<span></span>
  </span>
  <div>
    <label>
      <input type="radio" name="spring.option" value="pork"
        checked="checked"/>
      Pork
    </label>
    <label>
      <input type="radio" name="spring.option"
        value="shrimp"/>
      Pork and Shrimp
    </label>
    <label>
      <input type="radio" name="spring.option"
        value="vegetarian"/>
      Vegetarian
    </label>
  </div>
</div>

<div>
  <label>
    <input type="checkbox" name="appetizers" value="vnrolls"/>
    Vietnamese rolls (4)
  </label>
  <span price="5">
    <input type="text" name="vnrolls.quantity"
      disabled="disabled" value="1"/>
    $<span></span>
  </span>
  <div>
    <label>
      <input type="radio" name="vnrolls.option" value="pork"
        checked="checked"/>
      Pork
    </label>
    <label>
      <input type="radio" name="vnrolls.option"
        value="shrimp"/>

```

```

        Pork and Shrimp
    </label>
    <input type="radio" name="vnrolls.option"
        value="vegetarian"/>
    <label>Vegetarian</label>
</div>
</div>

<div>
    <label>
        <input type="checkbox" name="appetizers" value="rangoon"/>
        Crab rangoon (8)
    </label>
    <span price="6">
        <input type="text" name="rangoon.quantity"
            disabled="disabled" value="1"/>
        $<span></span>
    </span>
</div>
<div>
    <label>
        <input type="radio" name="rangoon.option"
            value="sweet checked="checked"/>
        Sweet-and-sour sauce
    </label>
    <label>
        <input type="radio" name="rangoon.option" value="hot"/>
        Hot mustard
    </label>
    <label>
        <input type="radio" name="rangoon.option" value="both"/>
        Both
    </label>
</div>
</div>

<div>
    <label>
        <input type="checkbox" name="appetizers"
            value="stickers"/>
        Pot stickers (6)
    </label>
    <span price="5">
        <input type="text" name="stickers.quantity"
            disabled="disabled" value="1"/>
        $<span></span>
    </span>
</div>
<div>
    <label>
        <input type="radio" name="stickers.option"
            value="pork" checked="checked"/>

```

```
        Pork
    </label>
    <label>
        <input type="radio" name="stickers.option"
            value="vegetarian" />
        Vegetarian
    </label>
</div>
</div>

<div></div>

</fieldset>
</body>
</html>
```

This code is robust in that it's independent of the number of appetizer entries. You'll note that nowhere in the JavaScript is it necessary to tell the code what elements correspond to appetizer entries. The power of jQuery selectors allows us to automatically locate them. New appetizer entries can be added at will—as long as they follow the prescribed format—and the code will automatically instrument them along with the previously existing entries.

In many ways, the code could stand some improvements. In the interest of brevity and focusing on the lessons at hand, we took a number of shortcuts that should be fixed before putting any such code into production. The following list details some areas for improvement (or even blatant shortcomings of the code) that you're encouraged to explore as exercises:

- As written, the code assumes that users will enter only valid numeric values into the quantity fields. We know better! Add validation that ensures that only valid numeric entries are made. What should you do when an invalid entry is made?
- When the options for unchecked appetizers are hidden, they are still enabled and will be submitted along with the rest of the visible elements. This is wasted bandwidth and more data for the server-side code to sift through. How would you enable and disable the radio options at appropriate times?
- The form is incomplete. In fact, without a `<form>` element, it isn't a form at all! Complete the HTML to make a valid form that can be submitted to the server.

- Man does not live by appetizers alone! How would you go about adding new sections for entrees, beverages, and desserts? Banana flambé sounds delightful! How would these new sections affect the setup of the JavaScript code?
- As diners are selecting (and deselecting) their choices, you could provide a running total of the order amount. How would you go about keeping track of the order total?
- If the use of custom attributes is not to your liking, refactor the page to eliminate them. But be sure that the price information remains defined in *one* place only!
- Perhaps the biggest flaw in the code is that it depends greatly on the positional relationships of the elements in an appetizer entry. This allowed the markup to remain simple but at the expense of both creating a strong binding between the structure of an entry and the supporting code and introducing complex jQuery selectors. How would you go about making the code more robust so that changes to the structure of an entry would have less impact on the code? Adding CSS class names to tag the elements (rather than relying on positional relationships) would be one fine way of accomplishing this; how would you go about it? What other ideas do you have?

If you come up with ideas that you're proud of, be sure to visit the Manning web page for this book at <http://www.manning.com/bibeault>, which contains a link to the discussion forum. You're encouraged to post your solutions for all to see and discuss!

4.4 Summary

Building upon the jQuery knowledge that we've gained so far, this chapter introduced us to the world of event handling.

We learned that there are vexing challenges to implementing event handling in web pages, but such handling is essential for creating pages in Rich Internet Applications. Not insignificant among those challenges is the fact that there are three event models that each operate in different ways across the set of modern popularly used browsers.

The legacy Basic Event Model, also informally termed the DOM Level 0 Event Model, enjoys somewhat browser-independent operation to declare event listeners, but the implementation of the listener functions requires divergent browser-dependent code. This event model is probably the most familiar to page authors, and assigns event listeners to DOM elements by assigning references to the listener functions to properties of the elements; the `onclick` property, for example.

Although simple, this model suffers from a you-only-get-one-shot problem; only one listener can be defined for any event type on a particular DOM element.

We can avoid this deficiency by using the DOM Level 2 Event Model, a more advanced and standardized model in which an API binds handlers to their event types and DOM elements. Versatile though this model is, it enjoys support only by standards-compliant browsers such as Firefox, Safari, Camino, and Opera.

For Internet Explorer 6 and 7, an API-based proprietary event model that provides a subset of the functionality of the DOM Level 2 Model is available.

Coding all event handling in a series of `if` statements—one clause for the standard browsers and one for Internet Explorer—is a good way to drive ourselves to early dementia. Luckily jQuery comes to the rescue and saves us from that fate.

jQuery provides a general `bind()` command to establish event listeners of any type on any element, as well as event-specific convenience commands such as `change()` and `click()`. These methods operate in a browser-independent fashion and normalize the `Event` instance passed to the handlers with the standard properties and methods most commonly used in event listeners.

jQuery also provides the means to remove event handlers, causes them to be invoked under script control, and even defines some higher-level commands that make implementing common event-handling tasks as easy as possible.

We explored a few examples of using events in our pages. In the next chapter, we'll look at how jQuery builds upon these capabilities to put animation and animated effects to work for us.

5

Sprucing up with animations and effects

This chapter covers

- Showing and hiding elements without animations
- Showing and hiding elements using core jQuery animated effects
- Other built-in effects
- Writing our own custom animations

Have you ever looked at a website built in Flash and become envious because of all the pretty effects that Flash developers have at their disposal? Heck, you might have even become tempted to learn Flash purely for those slick effects.

Not long ago, smooth effects and animations weren't realistic options using JavaScript. Between cross-browser issues and slow browser implementations, trying to fade or zoom elements, or even move them around the screen, was extraordinarily difficult. Thankfully, that state of affairs has passed, and jQuery provides a trivially simple interface for doing all sorts of neat effects.

But before we dive into adding whiz-bang effects to our pages, we need to contemplate the question: *should we?* Like a Hollywood blockbuster that's all special effects and no plot, a page that overuses effects can elicit the opposite reaction than we intended. Be mindful that effects should be used to *enhance* the usability of a page, not hinder it.

With that caution in mind, let's see what jQuery has to offer.

5.1 Showing and hiding elements

Perhaps the most common type of dynamic effect we'll want to perform on an element or group of elements is the simple act of showing or hiding them. We'll get to more fancy animations (like fading an element in or out) in a bit, but sometimes we want to keep it simple and pop elements into existence or make them vanish!

The commands for showing and hiding elements are pretty much what we'd expect: `show()` to show the elements in a wrapped set and `hide()` to hide them. We're going to delay presenting their formal syntax for reasons that will become clear in a bit; for now, let's concentrate on using these commands with no parameters.

As simple as these methods may seem, we should keep a few things in mind. First, jQuery hides elements by changing the `display` value of the `style` property to `none`. If an element in the wrapped set is already hidden, it will remain hidden but still be returned for chaining. For example, suppose we have the following HTML fragment:

```
<div style="display:none;">This will start hidden</div>
<div>This will start shown</div>
```

If we apply `$("#div").hide().addClass("fun")`, we'll end up with the following:

```
<div style="display:none;" class="fun">This will start hidden</div>
<div style="display:none;" class="fun">This will start shown</div>
```

Second, if an element starts as hidden by having its `display` style property value explicitly set to `none`, the `show()` command will always set its `display` style property value to `block`. That's even if the element would typically default to `inline` for its `display` value—as would a `` element, for example. If the element starts out without an explicitly declared `display` value, and we use the jQuery `hide()` command to hide it, the `show()` command will remember the original value and restore it to that original `display` state.

So it's usually a good idea not to use `style` attributes on the elements we want initially hidden, but to apply the `hide()` command to them in the page's ready handler. This prevents them from being displayed on the client, and also makes sure everything is in a known initial state and will behave as expected during subsequent hide and show operations.

Let's see about putting these commands to good use.

5.1.1 Implementing a collapsible list

Inundating a user with too much information at once is a common and classic user interface boo-boo. It's best to allow users to ask for information in digestible chunks that are under their control. This is a tiny glimpse into a larger principle known as *progressive disclosure* (which we were introduced to in the previous chapter) in which the data presented to users is kept to the minimum and expanded as required to perform the task at hand.

A good example of this might be browsing the filesystem of a computer. This information is frequently presented as a hierarchical list in which the content of folders is nested to the depth required to represent all files and folders on the system. It would be ludicrous to try to present all the files and folders on the system at once! A better approach is to allow each level in the list to be opened or closed to reveal the contained hierarchical information. You've surely seen such controls in any application that allows you to browse the filesystem.

In this section, we'll see how to use the `hide()` and `show()` commands to instrument a nested list that acts in this manner.

NOTE There are some nifty plugins that provide this type of control out-of-the-box if, once you understand the principles behind it, you'd rather use a ready-made solution than write your own.

To start, let's take a look at the HTML structure of the list we'll use to test our code.

```
<body>
  <fieldset>
    <legend>Collapsible List &mdash; Take 1</legend>
```

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>
    Item 3
    <ul>
      <li>Item 3.1</li>
      <li>
        Item 3.2
        <ul>
          <li>Item 3.2.1</li>
          <li>Item 3.2.2</li>
          <li>Item 3.2.3</li>
        </ul>
      </li>
      <li>Item 3.3</li>
    </ul>
  </li>
  <li>
    Item 4
    <ul>
      <li>Item 4.1</li>
      <li>
        Item 4.2
        <ul>
          <li>Item 4.2.1</li>
          <li>Item 4.2.2</li>
        </ul>
      </li>
    </ul>
  </li>
  <li>Item 5</li>
</ul>
</fieldset>
</body>
```

NOTE Because this section focuses on the effects, let's assume the list we'll instrument is small enough to send completely as part of the page. Obviously, this would not be true for something as massive as the data set for an entire filesystem. In such cases, you'd want to go back to the server for more and more data as you need it, but that's not what we want to focus on in this chapter. After you've read the Ajax chapter, you can revisit these examples and apply your skills to enhance these controls with such abilities.

When displayed in the browser (prior to tinkering with it), this list appears as shown in figure 5.1.

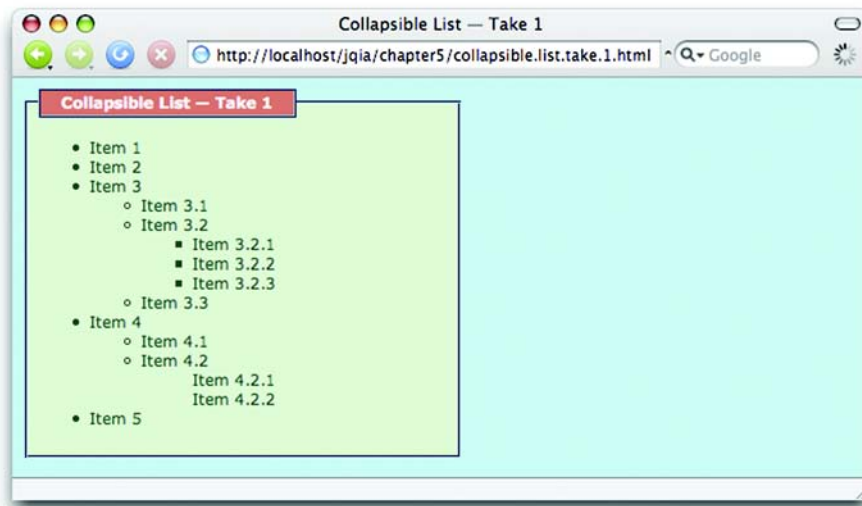


Figure 5.1 The nested list before we get our hands on it

A list of this size is relatively easy to digest, but it's easy to imagine it getting longer and deeper, in which case presenting the full list to users could cause them to suffer from the dreaded Information Overload Syndrome.

We want to instrument all list items that contain another embedded list so that the embedded content is hidden until the users choose to view it by clicking the item. After such a click, the child content of the item will be revealed, and a subsequent click will hide the contents again.

If we only hide the content of items that contain child lists (items 3 and 4 of the topmost list, for example), we'd likely confuse the users because they'd have no way of knowing these items are active and expandable; we need to visually distinguish these expandable items from those that are *leaf* items (those that can't be expanded). We do that by changing the cursor to the hand when it passes over an active item, as well as replacing its list marker with the ubiquitous plus- and minus-sign graphics to indicate that the item can be either expanded or collapsed.

We'll start the list off in its fully collapsed state and let the users take it from there. After instrumentation, the list will appear as shown in figure 5.2 when initially displayed.

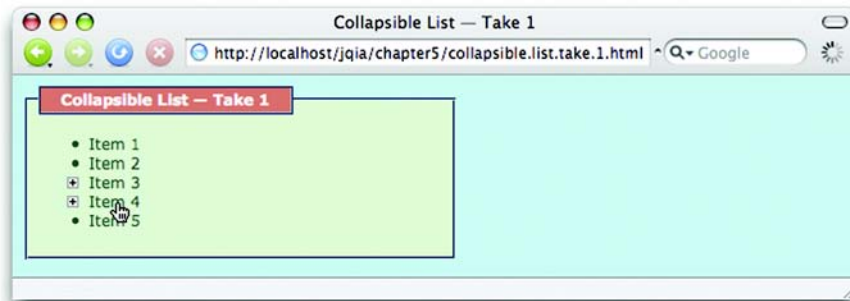


Figure 5.2 After instrumentation, the list has been fully collapsed and the expandable items are visually distinct.

We can see that for the list items that have content (items 3 and 4) the following occurs:

- The children of the list items 3 and 4 are hidden.
- The list markers for the items have been replaced with a plus-sign graphic to indicate that the item can be expanded.
- The mouse cursor changes to the hand when it hovers over these items.

Clicking these active items reveals their children as shown in the series of displays in figure 5.3.

Let's examine how we apply this behavior to the DOM elements of the list, setting it up within the ready handler, as shown in listing 5.1.

Listing 5.1 Ready-handler code that instruments the list with expandable behavior

```

$(function(){
  $('li:has(ul)') ← 1
    .click(function(event){ ← 2
      if (this == event.target) {
        if ($(this).children().is(':hidden')) { ← 3
          $(this)
            .css('list-style-image', 'url(minus.gif)')
            .children().show();
        } else {
          $(this)
            .css('list-style-image', 'url(plus.gif)')
            .children().hide();
        }
      }
    })
  return false; ← 4
})

```



```

        .css('cursor', 'pointer') ← 5
        .click();
    $('li:not(:has(ul))').css({ ← 7
        cursor: 'default',
        'list-style-image': 'none'
    });
});

```

There isn't a whole lot of code to this ready handler, but there *is* a lot of activity.

First, we select all list items that possess list children by applying the jQuery containment selector `li:has(ul)` ❶, and apply a chained series of jQuery commands to the matched elements, beginning with attaching a `click` handler ❷.

This click handler checks to make sure that the `target` element of the event matches `this`. This is true only when the clicked item is the same as the one on which the listener was established; it allows us to ignore clicks on the child elements. After all, we only want to open and close an item when users click the parent item, not one of its children.

If we determine that a parent item has been clicked, we then determine if its children are hidden or shown by employing the handy `is()` command using the `:hidden` filter ❸. If the children are hidden, we reveal them using `show()`, and if shown, we hide them using `hide()`. In either case, we change the parent item marker to the plus or minus image (as appropriate) and return `false` as the value of the listener ❹ to prevent needless propagation.

We set the mouse cursor shape to the active pointer using the `css()` command ❺ and hide the child elements for the active items, performing the actions defined within the `else` clause of the `if` statement in the click handler, by invoking the click handler ❻.

As the final step before users can interact with the page, we need to sledgehammer some styling elements for the leaf items ❼. We've set the `list-style-image` style (which controls the item marker) of the active items to one of the plus or minus GIF images, and we don't want that setting to be inherited by the list items that are children of those items. To prevent that inheritance, we explicitly set that `list-style-image` style property value to `none` for all leaf list items. Because we have set it directly on the items, it will take precedence over any inherited value.

We do the same for the mouse cursor for the leaf items by setting it to the default mouse cursor shape. Otherwise leaf items contained by an active parent would inherit the active cursor shape.

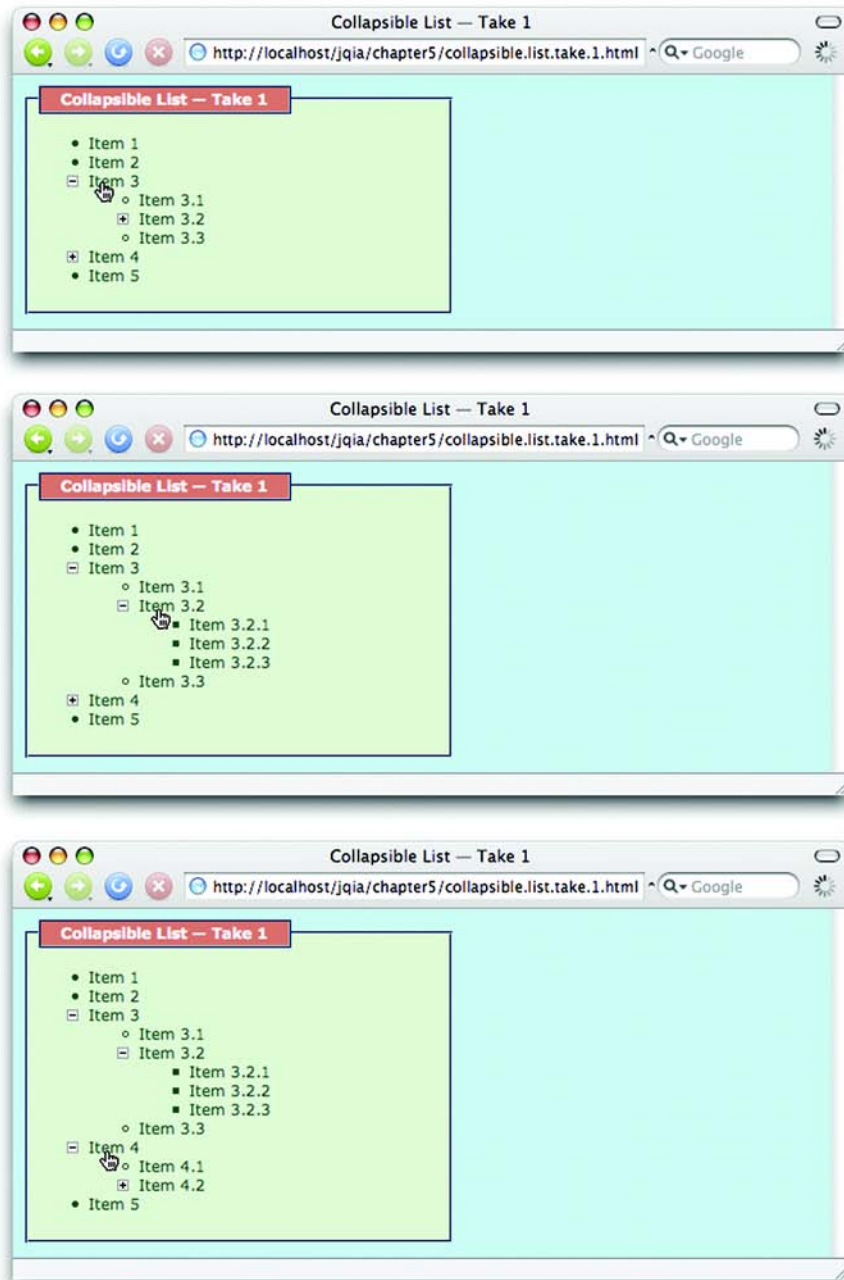


Figure 5.3 Clicking active elements causes their contents to be revealed.

The full code for this page can be found in file `chapter5/collapsible.list.take.1.html`. (If you surmise that the *take 1* part of this name indicates that we'll be revisiting this example, you're right!)

That wasn't too difficult for the amount of functionality enabled, but as it turns out, it can be even easier.

5.1.2 Toggling the display state of elements

Toggling the display state of elements between revealed or hidden—as for the collapsible list example—is such a common occurrence that jQuery defines a command named `toggle()` that makes it even easier.

Let's apply this command to the collapsible list and see how it helps to simplify the previous code. Listing 5.2 shows only the ready handler for the refactored page (no other changes are necessary) with the new changes highlighted in bold. The complete page code can be found in file `chapter5/collapsible.list.take.2.html`.

Listing 5.2 Changes to the collapsible list to use the `toggle()` command

```
$(function(){
  $('li:has(ul)')
    .click(function(event){
      if (this == event.target) {
        $(this).children().toggle();
        $(this).css('list-style-image',
          ($ (this).children().is(':hidden')) ?
          'url(plus.gif)' : 'url(minus.gif)');
      }
      return false;
    })
    .css('cursor', 'pointer')
    .click();
  $('li:not(:has(ul))').css({
    cursor: 'default',
    'list-style-image': 'none'
  });
});
```

Note that we no longer need the conditional statement to determine whether to hide or show the elements; `toggle()` takes care of swapping the displayed state. We still use the `.is(':hidden')` test as part of a simpler ternary expression to determine the appropriate image to use for the item marker.

Instantaneously making elements appear and disappear is handy, but sometimes we want the transition to be less abrupt. Let's see what's available for that.

5.2 Animating the display state of elements

Human cognitive ability being what it is, making items pop into and out of existence instantaneously can be jarring to us. If we blink at the wrong moment, we could miss the transition, leaving us to wonder, "What just happened?"

Gradual transitions of a short duration help us know what's changing and *how* we got from one state to the other. And that's where the jQuery core effects come in, of which there are three sets:

- Show and hide (there's a bit more to these commands than we let on in section 5.1)
- Fade in and fade out
- Slide down and slide up

Let's look more closely at each of these effect sets.

5.2.1 Showing and hiding elements gradually

The `show()`, `hide()`, and `toggle()` commands are more complex than we led you to believe in the previous section. When called with no parameters, these commands effect a simple manipulation of the display state of the wrapped elements, causing them to instantaneously be revealed or hidden from the display. But when passed parameters, these effects can be animated so that their changes in display status take place over a period of time.

Now we're ready to look at the full syntaxes of these commands.

Command syntax: hide**hide(speed, callback)**

Causes the elements in the wrapped set to become hidden. If called with no parameters, the operation takes place instantaneously by setting the `display` style property value of the elements to `none`. If a `speed` parameter is provided, the elements are hidden over a period of time by adjusting their size and opacity downward to zero, at which time their `display` style property value is set to `none` to remove them from the display.

An optional callback can be specified that's invoked when the animation is complete.

Parameters

- `speed` (Number|String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: *slow*, *normal*, or *fast*. If omitted, no animation takes place, and the elements are immediately removed from the display.
- `callback` (Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated.

Returns

The wrapped set.

Command syntax: show**show(speed, callback)**

Causes any hidden elements in the wrapped set to be revealed. If called with no parameters, the operation takes place instantaneously by setting the `display` style property value of the elements to their previous setting (such as `block` or `inline`) if the element was hidden via a jQuery effect. If the element was not hidden via jQuery, the `display` style property value defaults to `block`.

If a `speed` parameter is provided, the elements are revealed over a specified duration by adjusting their size and opacity upward.

An optional callback can be specified that's invoked when the animation is complete.

Parameters

- `speed` (Number|String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: *slow*, *normal*, or *fast*. If omitted, no animation takes place and the elements are immediately revealed in the display.
- `callback` (Function) An optional function invoked when the animation is complete. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated.

Returns

The wrapped set.

Command syntax: toggle**toggle (speed, callback)**

Performs `show()` on any hidden wrapped elements and `hide()` on any non-hidden wrapped elements. See the syntax description of these commands for their respective semantics.

Parameters

- `speed` (Number|String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: *slow*, *normal*, or *fast*. If omitted, no animation takes place.
- `callback` (Function) An optional function invoked when the animation is complete. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated.

Returns

The wrapped set.

Let's do a third take on the collapsible list, animating the opening and closing of the sections.

Given the previous information, you'd think that the only change we need to make to the code of *take 2* of this collapsible list implementation would be to change the call to the `toggle()` command to

```
toggle('slow')
```

But not so fast! When we make this change and test the page, we'll notice some weird things going on. First, recall that, in order to initially hide the collapsible elements, we called the `click` handler of the active items. That was well and good when all the handler did was to immediately hide the child elements. But now we've animated that activity; when the page loads, we see the child items hiding themselves in the animated fashion. That won't do at all!

We need to explicitly use the `hide()` command, without parameters, to hide the element before the user gets a chance to see them and then to set the markers to the plus image. You'll recall that we didn't do that in the earlier example because it would have created repeated code. Well, with the changes we've made, that's no longer an issue.

The second problem we'd notice is that marker images no longer act correctly. When the toggle action was instantaneous, we could safely check for the results of the action immediately after it took place. Now that the toggle action is animated, its results are no longer synchronous, and checking afterward for whether the children are hidden or not (in order to know which image the marker should be set to) is no longer possible.

Let's invert the sense of the test and check the state of the children *before* we issue the animated toggle.

The new ready handler, with changes highlighted in bold, is shown in listing 5.3.

Listing 5.3 Our list example augmented with the animated effects

```
$(function(){
  $('li')
    .css('pointer','default')
    .css('list-style-image','none');
  $('li:has(ul)')
    .click(function(event){
      if (this == event.target) {
        $(this).css('list-style-image',
           (!$ (this).children().is(':hidden')) ?
           'url(plus.gif)' : 'url(minus.gif)');
        $(this).children().toggle('slow');
      }
      return false;
    })
    .css({cursor:'pointer',
      'list-style-image':'url(plus.gif)'}
    .children().hide();
  $('li:not(:has(ul))').css({
    cursor: 'default',
    'list-style-image':'none'
  });
});
```

The page with these changes can be found in file `chapter5/collapsible.list.take.3.html`.

Knowing how much people like us love to tinker, we've set up a handy tool that we'll use to further examine the operation of these commands.

Introducing the jQuery Effects Lab Page

Back in chapter 2, we introduced the concept of lab pages to help us experiment with using jQuery selectors. For this chapter, we set up a lab page for exploring the operation of the jQuery effects in file `chapter5/lab.effects.html`.

Loading this page into your browser results in the display shown in figure 5.4.

This lab page consists of two main panels: a control panel in which we'll specify which effect will be applied and one that contains four test subject elements upon which the effects will act.

"Are they daft?" you might be thinking. "There are only two test subjects."

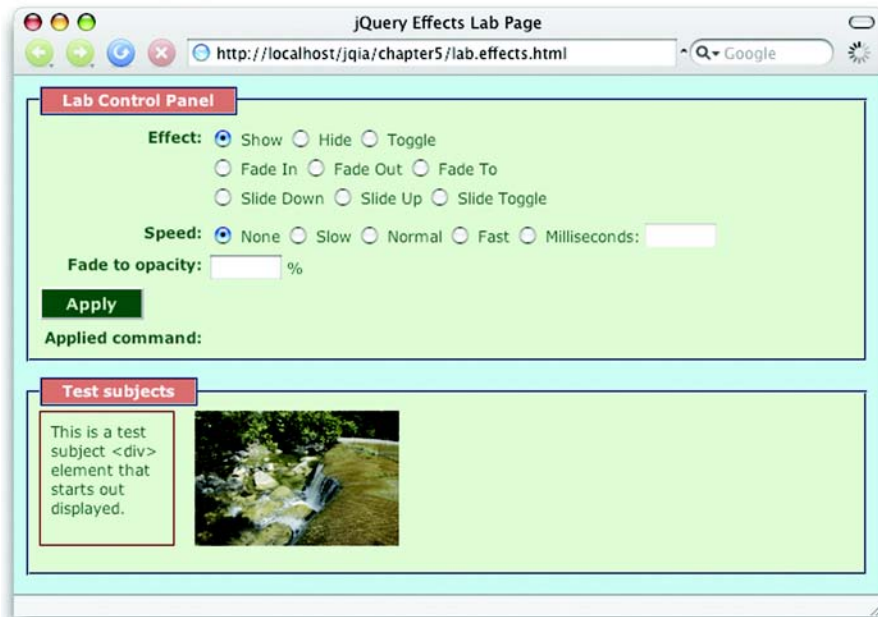


Figure 5.4 The initial state of the jQuery Effects Lab Page, which will help us examine the operation of the jQuery effects commands

No, your authors haven't lost it yet. There *are* four elements, but two of them (another `<div>` with text and another image) are initially hidden.

Let's use this page to demonstrate the operations of the commands we've discussed to this point. Display the page in your browser, and follow along with the ensuing exercises:

- *Exercise 1*—With the controls left as is after the initial page load, click the Apply button. This will execute a `show()` command with no parameters. The command that was applied is displayed below the Apply button for your information. Note how the two initially hidden test subject elements appear instantly. If you're wondering why the image on the far right appears a bit faded, its opacity has been purposefully set to 50 percent.
- *Exercise 2*—Select the Hide radio button, and click Apply to execute a parameterless `hide()` command. All of the test subjects vanish. Take special notice that the fieldset in which they reside has tightened up. This indicates that the elements have been completely removed from the display rather than made invisible.

- NOTE** When we say that an element has been *removed from the display* (here, and in the remainder of our discussion about effects), we mean that the element is no longer being taken into account by the browser’s layout manager, just as if its CSS `display` style property value has been set to `none`. It does *not* mean that the element has been removed from the DOM tree; none of the effects will ever cause an element to be removed from the DOM.
- *Exercise 3*—Next, select the Toggle radio button, and click Apply. Click Apply again. And again. You’ll note that each subsequent execution of `toggle()` flips the presence of the test subjects.
 - *Exercise 4*—Reload the page to reset everything to the initial conditions (in Firefox, set focus to the address bar and hit the Enter key). Select Toggle, and click Apply. Note how the two initially visible subjects vanish and the two that were hidden appear. This demonstrates that the `toggle()` command applies individually to each wrapped element, revealing the ones that are hidden and hiding those that aren’t.
 - *Exercise 5*—In this exercise, let’s move into the realm of animation. Refresh the page, and for the Speed setting, select Slow. Click Apply, and carefully watch the test subjects. The two hidden elements, rather than popping into existence, gradually grow from their upper left corners. If you want to see what’s going on, refresh the page, select Milliseconds for the speed and enter 10000 for the speed value. This will extend the duration of the effect to 10 (excruciating) seconds and give you plenty of time to observe the behavior of the effect.
 - *Exercise 6*—Choosing various combinations of Show, Hide, and Toggle, as well as various speeds, experiment with these effects until you feel you have a good handle on how they operate.

Armed with the jQuery Effects Lab Page, and the knowledge of how this first set of effects operates, let’s take a look at the next set of effects.

5.2.2 Fading elements into and out of existence

If you watched the operation of the `show()` and `hide()` effects carefully, you noted that they scaled the size of the elements (either up or down as appropriate) *and* adjusted the opacity of the elements as they grew or shrank. The next set of effects, `fadeIn()` and `fadeOut()`, only affect the opacity of the elements.

Other than the lack of scaling, these commands work in a fashion similar to `show()` and `hide()` (when animated) respectively. The syntaxes of these commands are as follow:

Command syntax: `fadeOut`

`fadeOut (speed , callback)`

Causes any matched elements that aren't hidden to be removed from the display by gradually changing their opacity to 0% and then removing the element from the display. The duration of the change in opacity is determined by the `speed` parameter. Any elements that are already hidden aren't affected.

Parameters

- `speed` (Number|String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: *slow*, *normal*, or *fast*. If omitted, the default is *normal*.
- `callback` (Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated.

Returns

The wrapped set.

Command syntax: `fadeIn`

`fadeIn (speed , callback)`

Causes any matched elements that are hidden to be shown by gradually changing their opacity to their natural value. This value is either the opacity originally applied to the element, or 100%. The duration of the change in opacity is determined by the `speed` parameter. Any elements that aren't hidden aren't affected.

Parameters

- `speed` (Number|String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: *slow*, *normal*, or *fast*. If omitted, the default is *normal*.
- `callback` (Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated.

Returns

The wrapped set.

Let's have some more fun with the jQuery Effects Lab Page. Display the lab, and run through a set of exercises similar to those we followed in the previous section

using the Fade In and Fade Out selections (don't worry about Fade To for now, we'll attend to that soon enough).

It's important to note that when the opacity of an element is adjusted, the jQuery `hide()`, `show()`, `fadeIn()`, and `fadeOut()` effects remember the original opacity of an element and honor its value. In the lab page, we purposefully set the initial opacity of the image at the far right to 50 percent before hiding it. Throughout all the opacity changes that take place when applying the jQuery effects, this original value is never stomped on.

Run through additional exercises in the lab until you're convinced that this is so and are comfortable with the operation of the fade effects.

Another effect that jQuery provides is via the `fadeTo()` command. This effect adjusts the opacity of the elements like the previously examined fade effects, but never removes the elements from the display. Before we start playing with `fadeTo()` in the lab, here's its syntax.

Command syntax: `fadeTo`

`fadeTo(speed, opacity, callback)`

Adjusts the opacity of the wrapped elements from their current setting to the new setting specified by `opacity`.

Parameters

- `speed` (Number|String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: *slow*, *normal*, or *fast*. If omitted, the default is *normal*.
- `opacity` (Number) The target opacity to which the elements will be adjusted specified as a value from 0.0 to 1.0.
- `callback` (Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated.

Returns

The wrapped set.

Unlike the other effects that adjust opacity while hiding or revealing elements, `fadeTo()` doesn't remember the original opacity of an element. This makes sense because the whole purpose of this effect is to explicitly change the opacity to a specific value.

Bring up the lab page, and cause all elements to be revealed (you should know how by now). Then work through the following exercises:

- *Exercise 1*—Select `Fade To` and a speed value slow enough for you to observe the behavior; 4000 milliseconds is a good choice. Now set the `Fade to Opacity` field (which expects a percentage value between 0 and 100, converted to 0.0 through 1.0 when passed to the command) to 10, and click `Apply`. The test subjects will fade to 10 percent opacity over the course of four seconds.
- *Exercise 2*—Set the opacity to 100, and click `Apply`. All elements, including the initially semi-transparent image, are adjusted to full opaqueness.
- *Exercise 3*—Set the opacity to 0, and click `Apply`. All elements fade away to invisibility, but note that once they’ve vanished, the enclosing fieldset does *not* tighten up. Unlike the `fadeOut()` effect, `fadeTo()` never removes the element from the display, even when it’s fully invisible.

Continue experimenting with the `Fade To` effect until you’ve mastered its workings. Then we’ll be ready to move on to the next set of effects.

5.2.3 Sliding elements up and down

Another set of effects that hide or show elements—`slideDown()` and `slideUp()`—also works in a similar manner to the `hide()` and `show()` effects, except that the elements appear to slide down from their tops when being revealed and to slide up into their tops when being hidden.

As with `hide()` and `show()`, the slide effects have a command that will toggle the elements between hidden and revealed: `slideToggle()`. The by-now-familiar syntaxes for these commands follow below.

Command syntax: `slideDown`

`slideDown(speed, callback)`

Causes any matched elements that are hidden to be shown by gradually increasing their vertical size. Any elements that aren’t hidden aren’t affected.

Parameters

- | | |
|-----------------------|---|
| <code>speed</code> | (Number String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: <i>slow</i> , <i>normal</i> , or <i>fast</i> . If omitted, the default is <i>normal</i> . |
| <code>callback</code> | (Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (<code>this</code>) is set to the element that was animated. |

Returns

The wrapped set.

Command syntax: slideUp**slideUp(speed, callback)**

Causes any matched elements that aren't hidden to be removed from the display by gradually decreasing their vertical size.

Parameters

- speed** (Number|String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: *slow*, *normal*, or *fast*. If omitted, the default is *normal*.
- callback** (Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (*this*) is set to the element that was animated.

Returns

The wrapped set.

Command syntax: slideToggle**slideToggle(speed, callback)**

Performs `slideDown()` on any hidden wrapped elements and `slideUp()` on any non-hidden wrapped elements. See the syntax description of these commands for their respective semantics.

Parameters

- speed** (Number|String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: *slow*, *normal*, or *fast*. If omitted, no animation takes place.
- callback** (Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (*this*) is set to the element that was animated.

Returns

The wrapped set.

Except for the manner in which the elements are revealed and hidden, these effects act similarly to the other show/hide effects. Convince yourself of this by displaying the jQuery Effects Lab Page and running through sets of exercises similar to those we applied to the other effects.

5.2.4 Stopping animations

We may have a reason now and again to stop an animation once it has started. This could be because a user event dictates that something else should occur or because we want to start a completely new animation. The `stop()` command will achieve this for us:

Command syntax: stop

`stop()`

Halts all animations that may be currently in progress for the elements in the matched set

Parameters

none

Returns

The wrapped set

Note that any changes that have already taken place for any animated elements will remain in effect. If we want to restore the elements to their original state, it's our responsibility to put the CSS values back to their starting values using the `css()` method or similar commands.

Now that we've seen the effects built into core jQuery, let's investigate writing our own!

5.3 Creating custom animations

The number of core effects supplied with jQuery is purposefully kept small (in order to keep jQuery's core footprint to a minimum) with the expectation that plugins are available to add more animations at the page author's discretion. It's also a simple matter to write our own animations.

jQuery publishes the `animate()` wrapper method that allows us to apply our own custom animated effects to the elements of the wrapped set. Let's take a look at its syntax.

Command syntax: animate**animate(properties, duration, easing, callback)****animate(properties, options)**

Applies an animation, as specified by the `properties` and `easing` parameters, to all members of the wrapped set. An optional callback function can be specified that's invoked when the animation is complete. An alternate format specifies a set of `options` in addition to the `properties`.

Parameters

<code>properties</code>	(Object) An object hash that specifies the end values that supported CSS styles should reach at the end of the animation. The animation takes place by adjusting the values of the style properties from the current value for an element to the value specified in this object hash.
<code>duration</code>	(Number String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: <i>slow</i> , <i>normal</i> , or <i>fast</i> . If omitted, no animation takes place.
<code>easing</code>	(String) The optional name of a function to perform easing of the animation. Easing functions must be registered by name and are often provided by plugins. Core jQuery supplies two easing functions registered as <i>linear</i> and <i>swing</i> .
<code>callback</code>	(Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (<code>this</code>) is set to the element that was animated.
<code>options</code>	(Object) Specifies the animation parameter values using an object hash. The supported properties are as follow: <ul style="list-style-type: none"> ■ <code>duration</code>—See previous description of <code>duration</code> parameter. ■ <code>easing</code>—See previous description of <code>easing</code> parameter. ■ <code>complete</code>—Function invoked when the animation completes. ■ <code>queue</code>—If <i>false</i>, the animation isn't queued and begins running immediately.

Returns

The wrapped set.

We create custom animations by supplying a set of CSS style properties and target values that those properties will converge towards as the animation progresses. Animations start with an element's original style value and proceed by adjusting that style value in the direction of the target value. The intermediate values that the style achieves during the effect (automatically handled by the animation engine) are determined by the duration of the animation and the easing function.

The specified target values can be absolute values, or we can specify relative values from the starting point. To specify relative values, prefix the value with `+=` or `-=` to indicate relative target values in the positive or negative direction, respectively.

The term *easing* is used to describe the manner in which the processing and pace of the frames of the animation are handled. By using some fancy math on the duration of the animation and current time position, some interesting variations to the effects are possible. The subject of writing easing functions is a complex, niche topic that's usually only of interest to the most hard-core of plugin authors; we're not going to delve into the subject of custom easing functions in this book. If you'd like to sample more easing functions than linear (which provides a linear progression) or swing (which speeds up slightly near the end of an animation), check out the Easing Plugin at <http://gsgd.co.uk/sandbox/jquery.easing.php>.

By default, animations are added to a queue for execution; applying multiple animations to an object will cause them to run serially. If you'd like to run animations in parallel, set the `queue` option to `false`.

The list of CSS style properties that can be animated is limited to those that accept numeric values for which there is a logical progression from a start value to a target value. This numeric restriction is completely understandable—how would we envision the logical progress from a source value to an end value for a non-numeric property such as `image-background`? For values that represent dimensions, jQuery assumes the default unit of pixels, but we can also specify `em` units or percentages by including the *em* or *%* suffixes.

Frequently animated style properties include `top`, `left`, `width`, `height`, and `opacity`. But if it makes sense for the effect we want to achieve, numeric style properties such as font size and border widths can also be animated.

NOTE If you want to animate a CSS value that specifies a color, you may be interested in the official jQuery Color Animation Plugin at <http://jquery.com/plugins/project/color>.

In addition to specific values, we can also specify one of the strings `hide`, `show`, or `toggle`; jQuery will compute the end value as appropriate to the specification of the string. Using `hide` for the `opacity` property, for example, will result in the opacity of an element being reduced to 0. Using any of these special strings has the added effect of automatically revealing or removing the element from the display (like the `hide()` and `show()` commands).

Did you notice when we introduced the core animations that there was no toggling command for the fade effects? That's easily solved using `animate()` and `toggle` as follows:

```
$('.animateMe').animate({opacity:'toggle'}, 'slow');
```


Let's try our hand at writing a few more custom animations.

5.3.1 A custom scale animation

Consider a simple *scale* animation in which we want to adjust the size of the elements to twice their original dimensions. We write such an animation as

```
$('.animateMe').each(function(){
  $(this).animate(
    {
      width: $(this).width() * 2,
      height: $(this).height() * 2
    },
    2000
  );
});
```

To implement this animation, we iterate over all the elements in the wrapped set via `each()` to apply the animation individually to each matched element. This is important because the property values that we need to specify for each element are based upon the individual dimensions for that element. If we always knew that we'd be animating a single element (such as if we were using an `id` selector) or applying the exact same set of values to each element, we could dispense with `each()` and animate the wrapped set directly.

Within the iterator function, the `animate()` command is applied to the element (identified via `this`) with style property values for `width` and `height` set to double the element's original dimensions. The result is that over the course of two seconds (as specified by the `duration` parameter of 2000), the wrapped elements (or element) will grow from their original size to twice that original size.

Now let's try something a bit more extravagant.

5.3.2 A custom drop animation

Let's say that we want to conspicuously animate the removal of an element from the display, perhaps because it's vitally important to convey to users that the item being removed is *gone* and that they should make no mistake about it. The animation we'll use to accomplish this will make it appear as if the element *drops* off the page, disappearing from the display as it does so.

If we think about it for a moment, we can figure out that, by adjusting the `top` position of the element, we can make it move down the page to simulate the drop; adjusting the `opacity` will make it seem to vanish as it does so. And finally, when all that's done, we want to remove the element from the display (similar to the animated `hide()`).

We accomplish this drop effect with the following code:

```
$('.animateMe').each(function(){
  $(this)
    .css('position','relative')
    .animate(
      {
        opacity: 0,
        top: $(window).height() - $(this).height() -
             $(this).position().top
      },
      'slow',
      function(){ $(this).hide(); });
});
```

There's a bit more going on here than with the previous effect. We once again iterate over the element set, this time adjusting the position and opacity of the elements. But to adjust the `top` value of an element relative to its original position, we first need to change its CSS `position` style property value to `relative`.

Then for the animation, we specify a target opacity of 0 and a computed `top` value. We don't want to move an element so far down the page that it moves below the window's bottom; this could cause scroll bars to be displayed where none may have been before, possibly distracting users. We don't want to draw their attention away from the animation—grabbing their attention is why we're animating in the first place! So we use the height and vertical position of the element, as well as the height of the window, to compute how far down the page the element should drop.

NOTE In most examples in this book, we've avoided using plugins as much as possible in order to focus on core jQuery. This doesn't always reflect real-world situations where core jQuery is usually used along with whatever plugins a page author needs to get the job done. The ease of writing jQuery plugins and the rich pool of plugins that are available are two of jQuery's greatest strengths. In this example animation (as well as the next that we'll examine), we've employed the services of the Dimensions Plugin's `position()` command to determine the initial location of the element relative to the page. We'll be looking into the Dimensions Plugin in more detail in chapter 9 (section 9.2, to be exact).

When the animation is complete, we want to remove the element from the display, so we specify a callback routine that applies the non-animated `hide()` command to the element (which is available to the function as its function context).

NOTE We did a little more work than we needed to in this animation, so we could demonstrate doing something that needs to wait until the animation is complete in the callback function. If we were to specify the value of the `opacity` property as `hide` rather than `0`, the removal of the element(s) at the end of the animation would be automatic, and we could dispense with the callback.

Now let's try one more type of “make it go away” effect for good measure.

5.3.3 A custom puff animation

Rather than dropping elements off the page, let's say that we want an effect that makes it appear as if the element dissipates away into thin air like a puff of smoke. To animate such an effect, we combine a scale effect with an opacity effect, growing the element while fading it away. One issue we need to deal with for this effect is that this dissipation would not fool the eye if we let the element grow in place with its upper-left corner anchored. We want the *center* of the element to stay in the same place as it grows, so we need to adjust the position of the element, in addition to its size, as part of the animation.

The code for our puff effect is

```
$('.animateMe').each(function(){
  var position = $(this).position();
  $(this)
    .css({position:'absolute',top:position.top,
          left:position.left})
    .animate(
      {
        opacity: 'hide',
        width: $(this).width() * 5,
        height: $(this).height() * 5,
        top: position.top - ($(this).height() * 5 / 2),
        left: position.left - ($(this).width() * 5 / 2)
      },
      'normal');
});
```

In this animation, we decrease the opacity to 0 while growing the element to five times its original size and adjusting its position by half that new size, resulting in the position of the center of the element remaining constant. We don't want the elements surrounding the animated element to be pushed out while the target element is growing, so we take it out of the flow by changing its position to `absolute` and explicitly setting its position coordinates.

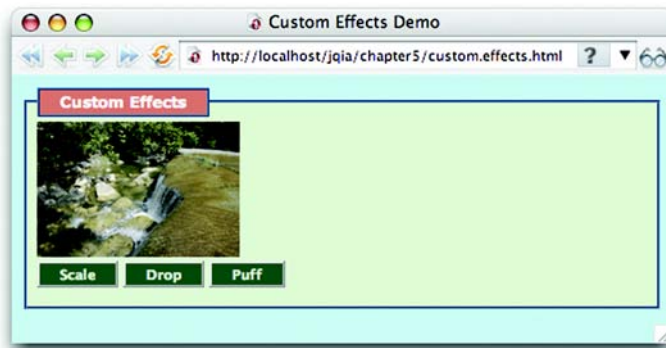


Figure 5.5
Initial display of the page that demonstrates the three custom effects: scale, drop, and puff

Because we specify `hide` for the `opacity` value, the elements are automatically hidden (removed from the display) once the animation is complete.

Each of these three custom effects can be observed by loading the page at `chapter5/custom.effects.html` whose display is shown in figure 5.5.

We purposefully kept the browser window to a minimum for the screen shot; you'll want to make the window bigger when running this page to properly observe the behavior of the effects. And although we'd love to show you how these effects behave, screenshots have obvious limitations. Nevertheless, figure 5.6 shows the puff effect in progress.

We'll leave it to you to try out the various effects on this page and observe their behavior.

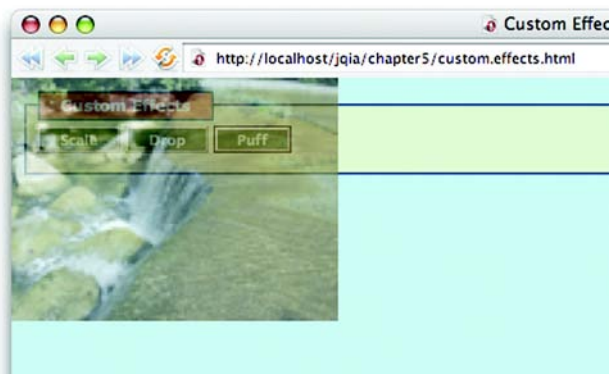


Figure 5.6
The puff effect expands and moves the image while simultaneously reducing its opacity.

5.4 Summary

This chapter introduced us to the animated effects that jQuery makes available out-of-the-box, as well as to the `animate()` wrapper method that allows us to create our own custom animations.

The `show()` and `hide()` commands, when used without parameters, reveal and conceal elements from the display immediately, without any animation. We can perform animated versions of the hiding and showing of elements with these commands by passing parameters that control the speed of the animation, as well as providing an optional callback that's invoked when the animation completes. The `toggle()` command toggles the displayed state of an element between hidden and shown.

Another set of wrapper methods, `fadeOut()` and `fadeIn()`, also hides and shows elements by adjusting the opacity of elements when removing or revealing them in the display. A third method, `fadeTo()`, animates a change in opacity for its wrapped elements without removing the elements from the display.

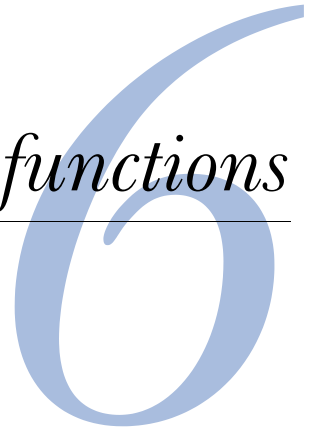
A final set of three built-in effects animates the removal or display of our wrapped elements by adjusting their vertical height: `slideUp()`, `slideDown()`, and `slideToggle()`.

For building our own custom animations, jQuery provides the `animate()` command. Using this method, we can animate any CSS style property that accepts a numeric value, most commonly the opacity and dimensions of the elements. We explored writing some custom animations that remove elements from the page in novel fashions.

We wrote the code for these custom effects as inline code within the on-page JavaScript. A much more common, and useful, method is to package these custom animations as jQuery commands. We'll learn how to do that in chapter 7, and you're encouraged to revisit these effects after you've read that chapter. Repackaging the custom effects found in this chapter, and any that you can think up on your own, would be an excellent follow-up exercise.

But before we write our own jQuery extensions, let's take a look at some high-level functions that jQuery provides.

jQuery utility functions



This chapter covers

- The jQuery browser detection flags
- Using other libraries with jQuery
- Functions for manipulating arrays
- Extending and merging objects
- Dynamically loading new script

Up to this point, we've spent a fair number of chapters examining jQuery *commands*—the term that we've applied to methods that operate upon a set of DOM elements wrapped by the `$()` function. But you may recall that way back in chapter 1, we also introduced the concept of *utility functions*—functions namespaced by `$` that don't operate on a wrapped set. These functions could be thought of as top-level functions except that they are defined on the `$` instance rather than `window`.

Generally, these functions either operate upon JavaScript objects *other* than DOM elements (that's the purview of the commands after all), or they perform some non-object-related operation.

You may wonder why we waited until this chapter to introduce these functions. Well, we had two primary reasons, which follow:

- We wanted to guide you into thinking in terms of using jQuery commands rather than resorting to lower-level operations that might feel more familiar but not be as efficient or as easy to code as using the jQuery commands.
- Because the commands take care of much of what we want to do when manipulating DOM elements on the pages, these lower-level functions are frequently most useful when writing the commands themselves (as well as other extensions) rather than in page-level code. (We'll be tackling how to write our own plugins to jQuery in the next chapter.)

In this chapter we're finally getting around to formally introducing most of the `$`-level utility functions, as well as a handful of useful flags. We'll put off talking about the utility functions that deal with Ajax until the chapter that deals exclusively with jQuery's Ajax functionality.

We'll start out with those flags that we mentioned.

6.1 Using the jQuery flags

Some of the information jQuery makes available to us as page authors, and even plugin authors, is available, not via methods or functions but as variables defined on `$`. These *flags* are generally focused on helping us divine the capabilities of the current browser so that we can make decisions based on such information when necessary.

The jQuery flags intended for public use are as follows:

- `$.browser`
- `$.boxModel`
- `$.styleFloat`

Let's start by looking at how jQuery informs us which browser is being used.

6.1.1 Detecting the user agent

Thankfully, almost blissfully, the jQuery commands that we've introduced so far shield us from having to deal with browser differences, even in traditionally problematic areas like event handling. But when we're the ones writing these commands (or other extensions), we often need to account for the differences in the ways browsers operate so that the users of our extensions don't have to.

But before we dive into seeing how jQuery helps us in this regard, let's talk about the concept of browser detection.

Why browser detection is *heinous*

OK, maybe the word *heinous* is too strong, but unless it's absolutely necessary, browser detection is a technique that should only be used when no other options are available.

Browser detection might seem, at first, like a logical way to deal with browser differences. After all, it's easy to say: "I know what the set of capabilities of browser X are, so testing for the browser makes perfect sense, right?" But browser detection is full of pitfalls and problems.

One of the major arguments against this technique is that the proliferation of browsers, as well as varying levels of support within versions of the same browser, makes this technique an unscalable approach to the problem.

You could be thinking, "Well, all I need to test for is Internet Explorer and Firefox." But why would you exclude the growing number of Safari users? What about Opera? Moreover, there are some niche, but not insignificant, browsers that share capability profiles with the more popular browsers. Camino, for example, uses the same technology as Firefox behind its Mac-friendly UI. And OmniWeb uses the same rendering engine as Safari.

There's no need to exclude support for these browsers, but it *is* a royal pain to have to test for them. And that's not even considering differences between versions—IE6 and IE7, for example.

Another argument against browser detection (or *sniffing* as it's sometimes called) is that it's getting harder and harder to know who's who.

Browsers identify themselves by setting a request header known as the *user agent* string. Parsing this string isn't for the faint-hearted. In addition, many browsers now allow their users to spoof this string, so we can't even believe what it tells us after we *do* parse it!

A JavaScript object named `navigator` gives us a partial glimpse into the user agent information, but even *it* has browser differences. We almost need to do browser detection in order to do browser detection!

Browser detection can be the following:

- *Imprecise*—Accidentally blocking browsers that our code would work within
- *Unscalable*—Leading to enormous nested if and if-else statements to sort things out
- *Inaccurate*—Due to users spoofing false user agent information

Obviously, we'd like to avoid using it whenever possible.

But what can we do instead?

What's the alternative?

If we think about it, we're not *really* interested in which browser anyone is using, are we? The only reason we're thinking about browser detection is so that we can know which browser capabilities we can use. It's the *capabilities* of a browser that we're after; using browser detection is just a ham-handed way of trying to determine those capabilities.

Why don't we figure out what those capabilities are rather than trying to infer them from the browser identification? The technique known broadly as *object detection* allows code to branch based on whether certain objects, properties, or even methods exist.

Let's think back to our chapter on event handling as an example. We remember that there are two advanced event-handling models: the W3C standard DOM Level 2 Event Model and the proprietary Internet Explorer Model. Both models define methods on the DOM elements that allow listeners to be established, but each uses different method names. The standard model defines the method `addEventListener()`, whereas the IE model defines `attachEvent()`.

Using browser detection, assuming that we've gone through the pain and aggravation of (maybe even correctly) determining what browser is being used, we could write

```
...
complex code to set flags: isIE, isFirefox and isSafari
...
if (isIE) {
    element.attachEvent('onclick',someHandler);
}
else if (isFirefox || isSafari) {
    element.addEventListener('click',someHandler);
}
else {
    throw new Error('event handling not supported');
}
```

Aside from the fact that this example glosses over whatever necessarily complex code we are using to set the flags `isIE`, `isFirefox`, and `isSafari`, we can't be sure if these flags accurately represent the browser being used. Moreover, this code will throw an error if used in Opera, Camino, OmniWeb, or a host of other lesser-known browsers that might perfectly support the standard model.

Consider the following variation of this code:

```
if (element.attachEvent) {
    element.attachEvent('onclick', someHandler);
}
else if (element.addEventListener) {
    element.addEventListener('click', someHandler);
}
else {
    throw new Error('event handling not supported');
}
```

This code doesn't perform a lot of complex, and ultimately unreliable, browser detection; it automatically supports all browsers that support either of the two competing event models. Much better!

Object detection is vastly superior to browser detection. It's more reliable, and it doesn't accidentally block browsers that support the capability we are testing for simply because we don't know about the capabilities of that browser.

NOTE Even object detection is best avoided unless absolutely required. If we can come up with a cross-browser solution, it should be preferred over any type of branching.

But as superior to browser detection as object detection may be, it can't always come to our rescue. There are times when we'll need to make browser-specific decisions (we'll see an example in a moment) that can only be made using browser detection.

So without any further ado, let's get around to finally answering the question...

What are the blasted browser flags?

For those times when only browser detection will do, jQuery provides a set of flags that we can use for branching that are set up when the library is loaded, making them available even before any ready handlers have executed. They are defined as properties of an object instance with a reference of `$.browser`. The formal syntax for this flag set is as follows:

Flag syntax: \$.browser**\$.browser**

Defines a set of flags that can be used to discover to which broad set of browser families the current user agent belongs. These flags are as follows:

- `msie`—Set to `true` if the user agent header identifies the browser as Microsoft Internet Explorer.
- `mozilla`—Set to `true` if the user agent header identifies the browser as any Mozilla-based browser. This includes browsers such as Firefox, Camino, and Netscape.
- `safari`—Set to `true` if the user agent header identifies the browser as Safari or any Safari-based browser such as OmniWeb.
- `opera`—Set to `true` if the user agent header identifies the browser as Opera.
- `version`—Set to the version number of the rendering engine for the browser.

Note that these flags don't attempt to identify the specific browser that's being used; jQuery classifies a user agent based upon which *family* of browsers it belongs to. Browsers within each family will sport the same sets of characteristics; specific browser identification should not be necessary.

The vast majority of commonly used, modern browsers will fall into one of these four browser families.

The `version` property deserves special notice because it's not as handy as we might think. The value set into this property isn't the version of the browser (as we might initially believe) but the version of the browser's rendering engine. For example, when executed within Firefox 2.0.0.2, the reported version is 1.8.1.6—the version of the Gecko rendering engine. This value *is* handy for distinguishing between IE6 and IE7, containing 6.0 and 7.0 respectively.

We mentioned earlier that there are times when we can't fall back on object detection and must resort to browser detection. One example of such a situation is when the difference between browsers isn't that they present different object classes or different methods but that the parameters passed to a method are interpreted differently across the browser implementations. In such a case, there's no object to perform detection on.

Let's look at the `add()` method to the `<select>` elements. It's defined as the following:

```
selectElement.add(element,before)
```

In this method, the first parameter identifies an `<option>` or `<optgroup>` element to add to the `<select>` element and the second identifies the existing `<option>` (or `<optgroup>`) that the new element is to be placed before. In standards-compliant browsers, this second parameter is a *reference* to the existing element; in Internet Explorer, it's the *ordinal index* of the existing element.

Because there's no way to perform object detection to determine if we should pass an object reference or an integer value, we must resort to browser detection as shown in the example page of listing 6.1, which can be found in the file `chapter6/$.browser.html`.

Listing 6.1 Testing for browsers

```
<html>
<head>
  <title>$.browser Example</title>
  <link rel="stylesheet" type="text/css" href="../common.css">
  <script type="text/javascript"
    src="../scripts/jquery-1.2.1.js"></script>
  <script type="text/javascript">
    $(function(){
      $('#testButton').click(function(event){
        var select = $('#testSubject')[0];
        select.add(
          new Option('Two and \u00BD', '2.5'),
          $.browser.msie ? 2 : select.options
        );
      });
    });
  </script>
</head>

<body class="plain">
  <select id="testSubject" multiple="multiple" size="5">
    <option value="1">One</option>
    <option value="2">Two</option>
    <option value="3">Three</option>
    <option value="4">Four</option>
  </select>
  <div>
    <button type="button" id="testButton">Click me!</button>
  </div>
</body>
</html>
```

← Employs browser detection for second parameter

This example sets up a `<select>` element with four entries and a simple button. The button is instrumented to add a new `<option>` element between the second and third original options. Because Internet Explorer will expect the ordinal value 2, but W3C-standard browsers will expect a reference to the third existing option, we use browser detection to branch what gets passed as the second parameter to the `add()` method.

The before-and-after results for a sampling of browsers are shown in figure 6.1.



Figure 6.1 The option-adding functionality works flawlessly in a wide range of browsers.

The example code was executed in six modern browsers—Firefox, Safari, Opera, Camino, Internet Explorer 7, and OmniWeb—giving representation from each of the four browser families supported by the jQuery `$.browser` flag set. As can be seen, the addition of the option occurs correctly in each of the browsers.

Not to belabor the point, it's important to once again note that this method of branching (browser detection) is far from optimal. The code that we wrote assumes that all non-Internet Explorer browsers will follow the W3C standard for the `add()` method to the `<select>` element. With testing, we've determined that this is true of the five non-Internet Explorer browsers depicted in figure 6.1 but what of other browsers? Would you know how Konqueror works without testing?

The bottom line is that because browser detection is such a broad brush, it requires thorough investigation of any browsers and platforms we wish to support with our code to know which branch a particular browser should take.

Leaving the subject of browser detection behind, let's move on to another flag that helps us deal with browser differences.

6.1.2 Determining the box model

Which *box model* the current page is using is available via the Boolean `$.boxModel` flag, which is set to `true` if the page is using the W3C standard box model and `false` if the page is using the Internet Explorer box model (sometimes called the *traditional* box model).

“What's the difference?” you may ask.

The box model determines how the size of the content of an element is determined in conjunction with its padding and border (margins, although part of the box model, take no part in determining the content size). Most browsers, other than Internet Explorer, support only the W3C box model, whereas Internet Explorer can use either model depending upon whether the page is being rendered in *strict mode* or *quirks mode*. Which mode is used depends upon the DOCTYPE (or lack thereof) declared for the page being rendered.

It's beyond the scope of this book to go into a detailed examination of the various DOCTYPE issues, but some rules of thumb that work most of the time are

- Pages with a valid and recognized DOCTYPE declaration are rendered in strict mode.
- Pages lacking a DOCTYPE declaration or those containing an unrecognized declaration are rendered in quirks mode.

If you want to dig deeper into the DOCTYPE issues, an excellent resource to consult is <http://www.quirksmode.org/css/quirksmode.html>.

In a nutshell, the difference between the two box models centers on how the `width` and `height` styles are interpreted. In the W3C box model, these values determine the dimensions of the content of the element, not counting its padding and

border widths; in the traditional box model, the values *include* padding and border width.

Let's say that we have an element with the following styles applied:

```
{
  width: 180px;
  height: 72px;
  padding: 10px;
  border-width: 5px;
}
```

The different ways that the box models interpret the sizing of the element is diagrammed in figure 6.2.

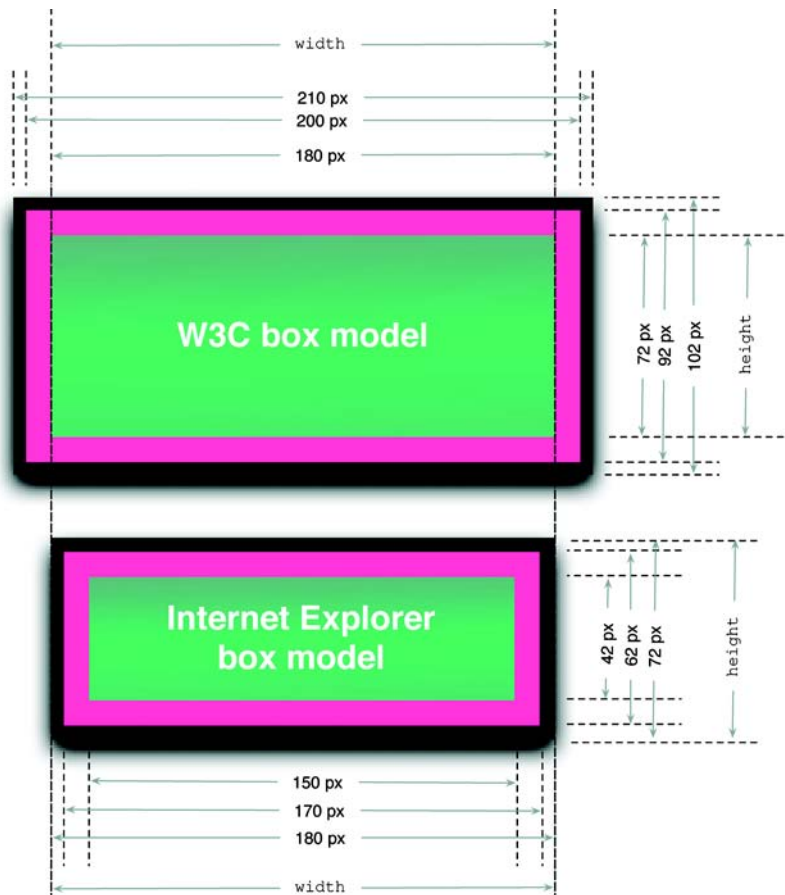


Figure 6.2 The W3C box model excludes padding and border measurements from the width of an element, whereas the Internet Explorer model doesn't.

Under the W3C box model, the size of the content of the element is 180 by 72 pixels exactly as specified by the `width` and `height` values. The padding and the border are applied *outside* this 180 by 72 pixel box, resulting in a total footprint of 210 by 102 pixels for the entire element.

When the traditional box model is used, the entire element is rendered in the 180 by 72 pixel box defined by the `width` and `height` attributes, reducing the size of the content to 150 by 42 pixels.

There are those on both sides of the fence that claim that one of these models is more intuitive or correct than the other, but the fact is we have to live with the difference.

If our code needs to account for these differences in that way that the elements will be rendered, the `$.boxModel` flag lets us know which is in force, and we can make our computations accordingly.

6.1.3 Detecting the correct float style to use

One other difference in browser capabilities that jQuery provides a flag for is the name used to represent the `float` CSS style in the element's `style` property. This flag, `$.styleFloat`, evaluates to the string that should be used for the property name.

For example, to set the `float` value for an element, we use

```
element.style[$.styleFloat] = 'left';
```

This will account for any browser differences between the naming of this property—this flag evaluates to `styleFloat` for Internet Explorer and to `cssFloat` for other browsers.

NOTE This flag is generally not required for on-page use. The `css()` wrapper method, when used with *float*, chooses the correct property (using this flag). The authors of plugins and other extensions are the target users of this flag for those cases where lower-level control is essential.

Now let's leave the world of flags and look at the utility functions that jQuery provides.

6.2 Using other libraries with jQuery

Back in section 1.3.6, we introduced a means, thoughtfully provided for us by the jQuery team, to easily use jQuery on the same page as other libraries. Usually,

the definition of the `$` variable is the largest point of contention and conflict when using other libraries on the same page as jQuery. As we know, jQuery uses `$` as an alias for the `jQuery` name, which is used for every feature that jQuery exhibits. But other libraries, most notably Prototype, use the `$` name as well.

jQuery provides the `$.noConflict()` utility function to relinquish control of the `$` name to whatever other library might wish to use it. The syntax of this function is as follows:

Function syntax: `$.noConflict`

`$.noConflict()`

Restores control of the `$` name back to another library, allowing mixed library use on pages using jQuery.

Once this function is executed, jQuery features will need to be invoked using the `jQuery` name rather than the `$` name.

Parameters

none

Returns

Undefined.

Because `$` is an alias for `jQuery`, all of jQuery's functionality is still available after the application of `$.noConflict()`, albeit by using the `jQuery` identifier. To compensate for the loss of the brief—yet beloved—`$`, we can define our own shorter, but non-conflicting, alias for `jQuery`, such as

```
var $j = jQuery;
```

Another idiom often employed is to create an environment where the `$` name is scoped to refer to the `jQuery` object. This technique is commonly used when extending jQuery, particularly by plugin authors who can't make any assumptions regarding whether page authors have called `$.noConflict()` and who, most certainly, can't subvert the wishes of the page authors by calling it themselves.

This idiom is as follows:

```
(function($) { /* function body here */ })(jQuery);
```

If this notation makes your head spin, don't worry! It's pretty straightforward if odd-looking to those encountering it for the first time.

Let's dissect the first part of this idiom:

```
(function($) { /* function body here */ })
```

This part declares a function and encloses it in parentheses to make an expression out of it, resulting in a reference to the anonymous function being returned as the value of the expression. The function expects a single parameter, which it names `$`; whatever is passed to this function can be referenced by the `$` identifier within the body of the function. And because parameter declarations have precedence over any similarly named identifiers in the global scope, any value defined for `$` outside of the function is superseded within the function by the passed argument.

The second part of the idiom

```
(jQuery)
```

performs a function call on the anonymous function passing the `jQuery` object as the argument.

As a result, the `$` name refers to the `jQuery` object within the body of the function regardless of whether it's already defined by Prototype or some other library *outside* of the function. Pretty nifty, isn't it?

When employing this technique, the external declaration of `$` isn't available within the body of the function.

A variant of this idiom is also frequently used to form a third syntax for declaring a ready handler in addition to the means that we already examined in section 1.3.3. Consider the following:

```
jQuery(function($) {  
    alert("I'm ready!");  
});
```

By passing a function as the parameter to the `jQuery` function, we declare it as a ready handler as we saw in section 1.3.3. But this time, we declare a single parameter to be passed to the ready handler using the `$` identifier. Because `jQuery` always passes a reference to `jQuery` to a ready handler as its first and only parameter, this guarantees that the `$` name refers to `jQuery` inside the ready handler regardless of the definition `$` might have outside the body of the handler.

Let's prove it to ourselves with a simple test. For the first part of the test, let's examine the HTML document of listing 6.2.

Listing 6.2 Ready handler test 1

```
<html>  
<head>  
  <title>Hi!</title>  
  <script type="text/javascript"  
    src="../scripts/jquery-1.2.1.js">  
</script>
```

```

<script type="text/javascript">
  var $ = 'Hi!';
  jQuery(function(){
    alert('$ = ' + $);
  });
</script>
</head>
<body>
</body>
</html>

```

① Overrides \$ name with custom value

② Declares the ready handler

In this document, we import jQuery, which (as we know) defines the global names jQuery and its alias \$. We then redefine the global \$ name to a string value ①, overriding the jQuery definition. We replace \$ with a simple string value for simplicity within this example, but it could be redefined by including another library such as Prototype.

We then define the ready handler ② whose only action is to display an alert showing the value of \$.

When we load this page, we see the alert displayed, as shown in figure 6.3.

Note that, within the ready handler, the *global* value of \$ is in scope and has the expected value resulting from our string assignment. How disappointing if we only wanted to use the jQuery definition of \$ within the handler.

Now let's make one change to this example document. Listing 6.3 shows only the portion of the document that has been modified; the minimal change is highlighted in bold.



Figure 6.3 The \$ says, “Hi!” as its redefinition takes effect within the ready handler.

Listing 6.3 Ready handler test 2

```
<script type="text/javascript">
  var $ = 'Hi!';
  jQuery(function($){
    alert('$ = '+ $);
  });
</script>
```

The only change we made was to add a parameter to the ready handler function named `$`. When we load this changed version, we see something completely different as shown in figure 6.4.



Figure 6.4 The alert now displays the jQuery version of `$` because its definition has been enforced within the function.

Well, that may not have been exactly what we might have predicted in advance. But a quick glance at the jQuery source code shows that, because we declare the first parameter of the ready handler to be `$` within that function, the `$` name refers to the jQuery function that jQuery passes as the sole parameter to all ready handlers (so the alert displays the definition of that function).

When writing reusable components, which might or might not be used in pages where `$.noConflict()` is used, it's best to take such precautions regarding the definition of `$`.

A good number of the remaining jQuery utility functions are used to manipulate JavaScript objects. Let's take a look at them.

6.3 Manipulating JavaScript objects and collections

The majority of jQuery features implemented as utility functions are designed to operate on JavaScript objects other than the DOM elements. Generally, anything

designed to operate on the DOM is provided as a jQuery command. Although some of these functions can be used to operate on DOM elements—which *are* JavaScript objects, after all—the focus of the utility functions isn’t DOM-centric.

Let’s start with one that’s basic.

6.3.1 Trimming strings

Almost inexplicably, the JavaScript `String` implementation doesn’t possess a method to remove whitespace characters from the beginning and end of a string instance. Such basic functionality is customarily part of a `String` class in most other languages, but JavaScript mysteriously lacks this useful feature.

Yet string trimming is a common need in many JavaScript applications; one prominent example is form data validation. Because whitespace is invisible on the screen (hence its name), it’s easy for users to accidentally enter extra space characters after (or sometimes even before) valid entries in text boxes or text areas. During validation, we want to silently trim such whitespace from the data rather than alerting the user to the fact that something that they can’t see is tripping them up.

To help us out, jQuery defines the `$.trim()` function as follows:

Function syntax: `$.trim`

`$.trim(value)`

Removes any leading or trailing whitespace characters from the passed string and returns the result.

Whitespace characters are defined by this function as any character matching the JavaScript regular expression `\s`, which matches not only the space character but also the form feed, new line, return, tab, and vertical tab characters, as well as the Unicode characters `\u00A0`, `\u2028`, and `\u2029`.

Parameters

`value` (String) The string value to be trimmed. This original value isn’t modified.

Returns

The trimmed string.

A small example of using this function to trim the value of a text field in-place is

```
$('#someField').val($.trim($('#someField').val()));
```

Be aware that this function doesn’t check the parameter we pass to ensure that it’s a `String` value, so we’ll likely get undefined and unfortunate results (probably a JavaScript error) if we pass any other value type to this function.

Now let’s look at some functions that operate on arrays and other objects.

6.3.2 Iterating through properties and collections

Oftentimes when we have non-scalar values composed of other components, we'll need to iterate over the contained items. Whether the container element is a JavaScript array (containing any number of other JavaScript values, including other arrays) or instances of JavaScript objects (containing properties), the JavaScript language gives us means to iterate over them. For arrays, we iterate over their elements using the `for` loop; for objects, we iterate over their properties using the `for-in` loop.

We can code examples of each as follows:

```
var anArray = ['one', 'two', 'three'];
for (var n = 0; n < anArray.length; n++) {
    //do something here
}

var anObject = {one:1, two:2, three:3};
for (var p in anObject) {
    //do something here
}
```

Pretty easy stuff, but some might think that the syntax is needlessly wordy and complex—a criticism frequently targeted at the `for` loop. We know that, for a wrapped set of DOM elements, jQuery defines the `each()` command, allowing us to easily iterate over the elements in the set without the need for messy `for` syntax. For general arrays and objects, jQuery provides an analogous utility function named `$.each()`.

The same syntax is used whether iterating over the items in an array or the properties of an object.

Function syntax: `$.each`

`$.each(container, callback)`

Iterates over the items in the passed container, invoking the passed callback function for each.

Parameters

- | | |
|------------------------|--|
| <code>container</code> | (Array Object) An array whose items or an object whose properties are to be iterated over. |
| <code>callback</code> | (Function) A function invoked for each element in the container. If the container is an array, this callback is invoked for each array item; if it's an object, the callback is invoked for each object property. The first parameter to this callback is the index of the array element or the name of the object property. The second parameter is the item or property value. The function context (<code>this</code>) of the invocation is also set to the value passed as the second parameter. |

Returns

The container object.

This unified syntax can be used to iterate over either arrays or objects using the same format. Using this function, we write the previous example as follows:

```
var anArray = ['one', 'two', 'three'];
$.each(anArray, function(n, value) {
    //do something here
});

var anObject = {one:1, two:2, three:3};
$.each(anObject, function(name, value) {
    //do something here
});
```

Although using `$.each()` with an inline function may seem like a six-of-one scenario in choosing syntax, this function makes it easy to write reusable iterator functions or to factor out the body of a loop into another function for purposes of code clarity as in the following:

```
$.each(anArray, someComplexFunction);
```

Note that when iterating over an array, we can break out of the loop by returning `false` from the iterator function. Doing so when iterating over object properties has no effect.

Sometimes we may iterate over arrays to pick and choose elements to become part of a new array. Let's see how jQuery makes that easy.

6.3.3 Filtering arrays

Traversing an array to find elements that match certain criteria is a frequent need of applications that handle lots of data. We might wish to filter through the data looking for items that fall above or below a particular threshold or, perhaps, that match a certain pattern. For any filtering operation of this type, jQuery provides the `$.grep()` utility function.

The name of the `$.grep()` function might lead us to believe that the function employs the use of regular expressions like its namesake, the UNIX `grep` command. But the filtering criteria used by the `$.grep()` utility function isn't a regular expression; it's a callback function provided by the caller that defines the criteria to determine if a data value should be included or excluded from the resulting set of values. Nothing prevents that callback from *using* regular expressions to accomplish its task, but the use of regular expressions is not automatic.

The syntax of the function is as follows:

Function syntax: \$.grep**\$.grep(array, callback, invert)**

Traverses the passed array invoking the callback function for each element. The return value of the callback function determines if the value is collected into a new array returned as the value of the `$.grep()` function. If the `invert` parameter is omitted or `false`, a callback value of `true` causes the data to be collected. If `invert` is `true`, a callback value of `false` causes the value to be collected.

The original array isn't modified.

Parameters

- array** (Array) The traversed array whose data values are examined for collection. This array isn't modified in any way by this operation.
- callback** (Function|String) A function whose return value determines if the current data value is to be collected. A return value of `true` causes the current value to be collected, unless the value of the `invert` parameter is `true` in which case the opposite occurs. This function is passed two parameters: the current data value and the index of that value within the original array. A string can also be passed as this parameter that's converted into the callback function. See the following discussion for details.
- invert** (Boolean) If specified as `true`, it inverts the normal operation of the function.

Returns

The array of collected values.

Let's say that we want to filter an array for all values that are greater than 100. We do that with a statement such as the following:

```
var bigNumbers = $.grep(originalArray, function(value) {
    return value > 100;
});
```

The callback function that we pass to `$.grep()` can use whatever processing it likes to determine if the value should be included. The decision could be as easy as this example or, perhaps, even as complex as making synchronous Ajax calls (with the requisite performance hit) to the server to determine if the value should be included or excluded.

When the decision making is as simple as this example, jQuery provides a shortcut that we can use to make the statement more compact—provide the expression as a string. For example, we can rewrite our code snippet as

```
var bigNumbers = $.grep(originalArray, 'a>100');
```


When the callback is specified as a string, jQuery automatically generates a callback function using the passed string as the value of the `return` statement and passing two parameters: `a` for the current value and `i` for the current index. So the generated function for this example is equivalent to

```
function(a,i){return a>100;}
```

Even though the `$.grep()` function doesn't directly use regular expressions (despite its name), JavaScript regular expressions can be powerful tools for us to use in our callback functions to determine whether to include or exclude values from the resultant array. Consider a situation in which we have an array of values and wish to identify any values that don't match the pattern for United States Postal Codes (also known as ZIP Codes).

US Postal Codes consist of five decimal digits optionally followed by a dash and four more decimal digits. A regular expression for such a pattern would be `/^\d{5}(-\d{4})?$/` , so we could filter a source array for non-conformant entries with the following:

```
var badZips = $.grep(
    originalArray,
    function(value) {
        return value.match(/^\d{5}(-\d{4})?$/) != null;
    },
    true);
```

Notable in this example is the use of the `String` class's `match()` method to determine whether a value matches the pattern or not and the specification of the `invert` parameter to `$.grep()` as `true` to *exclude* any values that match the pattern.

Collecting subsets of data from arrays isn't the only operation we might perform upon them. Let's look at another array-targeted function that jQuery provides.

6.3.4 Translating arrays

Data might not always be in the format that we need it to be. Another common operation that's frequently performed in data-centric web applications is the *translation* of a set of values to another set. Although it's a simple matter to write a `for` loop to create one array from another, jQuery makes it even easier with the `$.map` utility function.

Function syntax: \$.map**\$.map(array, callback)**

Iterates through the passed array, invoking the callback function for each array item and collecting the return values of the function invocations in a new array.

Parameters

- array** (Array) The array whose values are to be transformed to values in the new array.
- callback** (Function|String) A function whose return values are collected in the new array returned as the result of a call to the `$.map()` function. This function is passed two parameters: the current data value and the index of that value within the original array. A string can also be passed that's converted into the callback function. See the following discussion for details.

Returns

The wrapped set.

Let's look at a trivial example that shows the `$.map()` function in action.

```
var oneBased = $.map([0,1,2,3,4],function(value){return value+1;});
```

This statement converts an array of values, a zero-based set of indexes to a corresponding array of one-based indexes.

As with `$.grep()`, for such simple expressions, we can pass a string that represents the expression to make the statement more compact. The function generated on our behalf in such cases is passed the value as a parameter named `a` (unlike `$.grep()`, the index isn't passed to auto-generated functions). We can rewrite our example as

```
var oneBased = $.map([0,1,2,3,4], 'a+1');
```

Another important behavior to note is that if the function returns either `null` or `undefined`, the result isn't collected. In such cases, the resulting array will be smaller in length than the original, and one-to-one correspondence between items by order is lost.

Let's look at a slightly more involved example. Imagine that we have an array of strings, perhaps collected from form fields, that are expected to represent numeric values and that we want to convert this array to an array of corresponding `Number` instances. Because there's no guarantee against the presence of an invalid numeric string, we need to take some precautions. Consider the following code:

```
var strings = ['1','2','3','4','S','6'];

var values = $.map(strings,function(value){
    var result = new Number(value);
    return isNaN(result) ? null : result;
});
```

We start with an array of string values, each of which is expected to represent a numeric value. But a typo (or perhaps user entry error) resulted in *S* instead of the expected *5*. Our code handles this case by checking the `Number` instance created by the constructor to see if the conversion from string to numeric was successful or not. If the conversion fails, the value returned will be the constant `Number.NaN`. But the funny thing about `Number.NaN` is that by definition, it doesn't equal anything else, *including* itself! Therefore the value of the expression `Number.NaN==Number.NaN` is `false`!

Because we can't use a comparison operator to test for `NaN` (which stands for *Not a Number*, by the way), JavaScript provides the `isNaN()` method, which we employ to test the result of the string-to-numeric conversion.

In this example, we return `null` in the case of failure, ensuring that the resulting array contains only the valid numeric values with any error values elided. If we want to collect all the values, we can allow the transformation function to return `Number.NaN` for bad values.

Another useful behavior of `$.map()` is that it gracefully handles the case where an *array* is returned from the transformation function, merging the returned value into the resulting array. Consider the following statement:

```
var characters = $.map(
    ['this','that','other thing'],
    function(value){return value.split('');}
);
```

This statement transforms an array of strings into an array of all of the characters that make up the strings. After execution, the value of the variable `characters` is as follows:

```
['t','h','i','s','t','h','a','t','o','t','h','e','r','','t','h',
 ➡ 'i','n','g']
```

This is accomplished by use of the `String.split()` method, which returns an array of the string's characters when passed an empty string as its delimiter. This array is returned as the result of the transformation function and is merged into the resultant array.

jQuery's support for arrays doesn't stop there. There are a handful of minor functions that we might find handy.

6.3.5 More fun with JavaScript arrays

Have you ever needed to know if a JavaScript array contained a specific value and, perhaps, even the location of that value in the array?

If so, you'll appreciate the `$.inArray()` function

Function syntax: `$.inArray`

`$.inArray(value, array)`

Returns the index position of the first occurrence of the passed value

Parameters

`value` (Object) The value for which the array will be searched

`array` (Array) The array to be searched

Returns

The index of the first occurrence of the value within the array or -1 if the value is not found

A trivial but illustrative example of using this function is

```
var index = $.inArray(2,[1,2,3,4,5]);
```

This results in the index value of 1 being assigned to the `index` variable.

Another useful array-related function creates JavaScript arrays from other array-like objects.

“Other *array-like objects*? What on Earth is an array-like object?” you may ask.

jQuery considers other *array-like objects* to be any object that has a length and the concept of indexed entries. This capability is most useful for `NodeList` objects. Consider the following snippet:

```
var images = document.getElementsByTagName("img");
```

This populates the variable `images` with a `NodeList` of all the images on the page.

Dealing with a `NodeList` is a bit of a pain, so converting it to a JavaScript array makes things a lot nicer. The jQuery `$.makeArray` function makes converting the `NodeList` easy.

Function syntax: \$.makeArray**\$.makeArray(object)**

Converts the passed array-like object into a JavaScript array

Parameters

`object` (Object) The array-like object (such as a `NodeList`) to be converted

Returns

The resulting JavaScript array

This function is intended for use in code that doesn't make much use of jQuery, which internally handles this sort of thing on our behalves. This function also comes in handy when dealing with `NodeList` objects while traversing XML documents without jQuery.

Another seldom-used function that might come in handy when dealing with arrays built outside of jQuery is the `$.unique()` function.

Function syntax: \$.unique**\$.unique(array)**

Given an array of DOM elements, returns an array of the unique elements in the original array

Parameters

`array` (Array) The array of DOM elements to be examined

Returns

An array of DOM elements consisting of the unique elements in the passed array

Again, this is a function that jQuery uses internally to ensure that the lists of elements that we receive contain unique elements, and is intended for use on element arrays created outside the bounds of jQuery.

Now that we've seen how jQuery helps us to easily work with arrays, let's see a way that it helps us manipulate plain old JavaScript objects..

6.3.6 Extending objects

Although we all know that JavaScript provides some features that make it act in many ways like an object-oriented language, we know that JavaScript isn't what anyone would call purely object-oriented because of the features that it doesn't support. One of these important features is *inheritance*—the manner in which new classes are defined by extending the definitions of existing classes.

A pattern for mimicking inheritance in JavaScript is to extend an object by copying the properties of a base object into the new object, extending the new object with the capabilities of the base.

NOTE If you're an aficionado of object-oriented JavaScript, you'll no doubt be familiar with extending not only object instances but also their blueprints via the `prototype` property of object constructors. `$.extend()` can be used to effect such constructor-based inheritance by extending `prototype`, as well as object-based inheritance by extending existing object instances. Because understanding such advanced topics isn't a requirement in order to use jQuery effectively, this is a subject—albeit an important one—that's beyond the scope of this book.

It's fairly easy to write JavaScript code to perform this extension by copy, but as with so many other procedures, jQuery anticipates this need and provides a ready-made utility function to help us out. As we'll see in the next chapter, this function is useful for much more than extending an object, but even so its name is `$.extend()`. Its syntax is as follows:

Function syntax: `$.extend`

`$.extend(target, source1, source2, ... sourceN)`

Extends the object passed as `target` with the properties of the remaining passed objects.

Parameters

`target`

(Object) The object whose properties are augmented with the properties of the source objects. This object is directly modified with the new properties before being returned as the value of the function.

Any properties with the same name as properties in any of the source elements are overridden with the values from the source elements.

`source1 ... sourceN`

(Object) One or more objects whose properties are added to the `target` object.

When more than one source is provided and properties with the same name exist in the sources, sources later in the argument list take precedence over those earlier in the list.

Returns

The extended `target` object.

Let's take a look at this function doing its thing. Examine the code of listing 6.4, which can also be found in the file `chapter6/$.extend.html`.

Listing 6.4 Putting the \$.extend function to the test

```

<html>
<head>
  <title>$.extend Example</title>
  <link rel="stylesheet" type="text/css" href="../common.css">
  <script type="text/javascript"
    src="../scripts/jquery-1.2.1.js"></script>
  <script type="text/javascript"
    src="../scripts/support.labs.js"></script>
  <script type="text/javascript">
    var target = { a: 1, b: 2, c: 3 };
    var source1 = { c: 4, d: 5, e: 6 };
    var source2 = { e: 7, f: 8, g: 9 };

    $(function(){
      $('#targetBeforeDisplay').html($.toSource(target));
      $('#source1Display').html($.toSource(source1));
      $('#source2Display').html($.toSource(source2));
      $.extend(target,source1,source2);
      $('#targetAfterDisplay').html($.toSource(target));
    });
  </script>
  <style type="text/
css">
  label { float: left; width: 108px; text-align: right; }
  p { clear: both; }
  label + span { margin-left: 6px; }
</style>
</head>

<body>
  <fieldset>
    <legend>$.extend() Example</legend>
    <p>
      <label>target (before) =</label>
      <span id="targetBeforeDisplay"></span>
    </p>
    <p>
      <label>source1 =</label>
      <span id="source1Display"></span>
    </p>
    <p>
      <label>source2 =</label>
      <span id="source2Display"></span>
    </p>
  </fieldset>
</body>

```

1 Defines test objects
2 Displays before state of objects
3 Extends target object with source objects
4 Displays after state of target object
5 Defines HTML elements for display

```
<p>
  <label>target (after) =</label>
  <span id="targetAfterDisplay"></span>
</p>
</fieldset>
</body>
</html>
```

In this simple example, we define three objects: a target and two source objects **1**. We'll use these objects to demonstrate what happens to the target when `$.extend` is used to merge the two sources into it.

After declaring the objects, we define a ready handler in which we'll operate on them. Even though the objects are available immediately, we are going to display results on the page, so we need to wait until the HTML elements **5** have been rendered to start playing around.

Within the ready handler, we display the state of the three objects in `` elements defined to hold the results **2**. (If you're interested in how the `$.toSource()` function works, its definition can be found in the `support.labs.js` file. We'll address adding such utility functions to our repertoire in the next chapter.)

We extend the target object with the two source objects **3** using the following:

```
$.extend(target, source1, source2);
```

This merges the properties from objects `source1` and `source2` into the `target` object. The `target` object is returned as the value of the function; but, because the `target` is modified in place, we don't need to create a variable to hold its reference. The fact that the `target` is returned is significant when using this function as part of a statement chain.

Then, we display the values for the modified `target` **4**. The results are as shown in figure 6.5.

As we can see, all properties of the source objects have been merged into the target object. But note the following important nuances:

- Both the `target` and `source1` contain a property named `c`. The value of `c` in `source1` replaces the value in the original `target`.
- Both `source1` and `source2` contain a property named `e`. Note that the value of `e` within `source2` takes precedence over the value within `source1` when merged into `target`, demonstrating how objects later in the list of arguments take precedence over those earlier in the list.

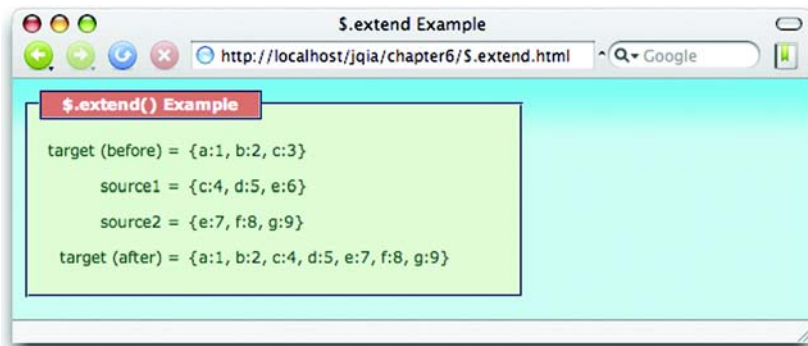


Figure 6.5 Using `$.extend` to merge object results in all source properties being copied into the target object.

Although it's evident that this utility function can be useful in many scenarios where one object must be extended with properties from another object (or set of objects), we'll see a concrete and common use of this feature when learning how to define utility functions of our own in the next chapter.

But before we get to that, let's wrap up our investigation of the utility functions with one that we can use to dynamically load new script into our pages.

6.4 Dynamically loading scripts

Most of the time—perhaps, almost always—we'll load the external scripts our page needs from script files when the page loads via `<script>` tags in the `<head>` of the page. But every now and again, we might want to load some script after the fact under script control.

We might do this because we don't know if the script will be needed until after some specific user activity has taken place but don't want to include the script unless absolutely needed, or perhaps, we might need to use some information not available at load time to make a conditional choice between various scripts.

Regardless of why we might want to dynamically load new script into the page, jQuery provides the `$.getScript()` utility function to make it easy.

Function syntax: \$.getScript**\$.getScript(url, callback)**

Fetches the script specified by the `url` parameter using a GET request to the specified server, optionally invoking a callback upon success.

Parameters

- `url` (String) The URL of the script file to fetch.
- `callback` (Function) An optional function invoked after the script resource has been loaded and evaluated.
- The following parameters are passed:
- The text loaded from the resource
 - The string *success*

Returns

The XHR instance used to fetch the script.

Under its covers, this function uses jQuery's built-in Ajax mechanisms to fetch the script file. We'll be covering these Ajax facilities in great detail in chapter 8, but we don't need to know anything about Ajax to use this function.

After fetching, the script in the file is evaluated; any inline script is executed, and any defined variables or functions become available.

WARNING In Safari, the script definitions loaded from the fetched file don't become available right away, even in the callback to the function. Any dynamically loaded script elements don't become available until after the script block within which it is loaded relinquishes control back to the browser. If your pages are going to support Safari, plan accordingly!

Let's see this in action. Consider the following script file (available in chapter6/new.stuff.js):

```
alert("I'm inline!");

var someVariable = 'Value of someVariable';

function someFunction(value) {
    alert(value);
}
```

This trivial script file contains an inline statement (which issues an alert that leaves no doubt as to when the statement gets executed), a variable declaration, and a declaration for a function that issues an alert containing whatever value is passed to it when executed. Now let's write a page to include this script

file dynamically. The page is shown in listing 6.5 and can be found in the file `chapter6/$.getScript.html`.

Listing 6.5 Dynamically loading a script file and examining the results

```

<html>
  <head>
    <title>$.getScript Example</title>
    <link rel="stylesheet" type="text/css" href="../common.css">
    <script type="text/javascript"
      src="../scripts/jquery-1.2.1.js"></script>
    <script type="text/javascript">
      $(function(){
        $('#loadButton').click(function(){
          $.getScript(
            'new.stuff.js',//,function(){$('#inspectButton').click()}
          );
        });
        $('#inspectButton').click(function(){
          someFunction(someVariable);
        });
      });
    </script>
  </head>
  <body>
    <button type="button" id="loadButton">Load</button>
    <button type="button" id="inspectButton">Inspect</button>
  </body>
</html>

```

❶ Fetches the script on clicking the Load button
❷ Displays result on clicking the Inspect button
❸ Defines the buttons

This page defines two buttons ❸ that we use to trigger the activity of the example. The first button, labeled Load, causes the `new.stuff.js` file to be dynamically loaded through use of the `$.getScript()` function ❶. Note that, initially, the second parameter (the callback) is commented out—we'll get to that in a moment.

On clicking that button, the `new.stuff.js` file is loaded, and its content is evaluated. As expected, the inline statement within the file triggers an alert message as shown in figure 6.6.

Clicking the Inspect button executes its `click` handler ❷, which executes the dynamically loaded `someFunction()` function passing the value of the dynamically loaded `someVariable` variable. If the alert appears as shown in figure 6.7, we know that both the variable and function are loaded correctly.

If you'd like to observe the behavior of Safari that we warned you about earlier, make a copy of the HTML file of listing 6.5, and uncomment the callback



Figure 6.6 The dynamic loading and evaluation of the script file results in the inline alert statement being executed.

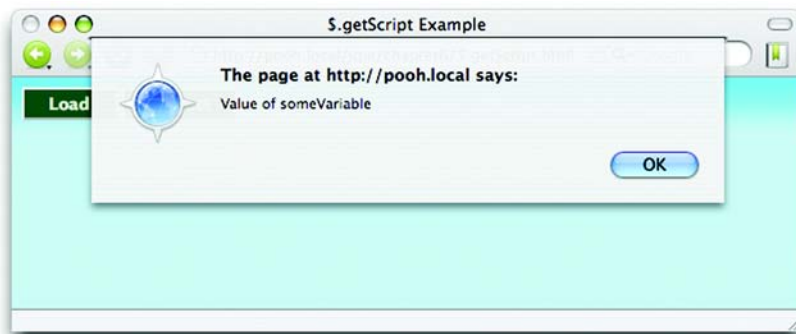


Figure 6.7 The appearance of the alert shows that the dynamic function is loaded correctly, and the correctly displayed value shows that the variable was dynamically loaded.

parameter to the `$.getScript()` function. This callback executes the `click` handler for the `Inspect` button, calling the dynamically loaded function with the loaded variable as its parameter.

In browsers other than Safari, the function and variable loaded dynamically from the script are available within the callback function. But when executed on Safari, nothing happens! We need to take heed of this divergence of functionality when using the `$.getScript()` function.

6.5 Summary

In this chapter we surveyed the features that jQuery provides outside of the methods that operate upon a wrapped set of matched DOM elements. These included an assortment of functions, as well as a set of flags, defined directly on the `jQuery` top-level name (as well as its `$` alias).

When we need to resort to browser detection to account for differences in browser capabilities and operation, the `$.browser` set of flags lets us determine within which browser family the page is being displayed. Browser detection should be used only as a last resort when it's impossible to write the code in a browser-independent fashion, and the preferred approach of object detection can't be employed.

The `$.boxModel` flag tells us which of the two box models is being used to render the page, and the `$.styleFloat` flag lets us reference the style property of the `float` style in a browser-independent manner.

Recognizing that page authors may sometimes wish to use other libraries in conjunction with jQuery, jQuery provides `$.noConflict()`, which allows other libraries to use the `$` alias. After calling this function, all jQuery operations must use the `jQuery` name rather than `$`.

`$.trim()` exists to fill the gap left by the native JavaScript `String` class for trimming whitespace from the beginning and end of string values.

jQuery also provides a set of functions that are useful for dealing with data sets in arrays. `$.each()` makes it easy to traverse through every item in an array; `$.grep()` allows us to create new arrays by filtering through the data of a source array using whatever filtering criteria we would like to use; and `$.map()` allows us to easily apply our own transformations to a source array to produce a corresponding new array with the transformed values.

To merge objects, perhaps even to mimic a sort of inheritance scheme, jQuery also provides the `$.extend()` function. This function allows us to unite the properties and any number of source objects into a target object.

And for those times when we want to load a script file dynamically, jQuery defines `$.getScript()`, which can load and evaluate a script file at any point in the execution of other page script.

With these additional tools safely tucked away in our toolbox, we're ready to tackle how to add our own extensions to jQuery. Let's get to it in the next chapter.



Extending jQuery with custom plugins

This chapter covers

- Why to extend jQuery with custom code
- The rules for effectively extending jQuery
- Writing custom utility functions
- Writing custom wrapper methods

Over the course of the previous chapters, we've seen that jQuery gives us a large toolset of useful commands and functions; we've also seen that we can easily tie these tools together to give our pages whatever behavior we choose. Sometimes that code follows common patterns we want to use again and again. When such patterns emerge, it makes sense to capture these repeated operations as reusable tools that we can add to our original toolset. In this chapter, we explore how to capture these reusable fragments of code as extensions to jQuery.

But before any of that, let's discuss *why* we'd want to pattern our own code as extensions to jQuery in the first place.

7.1 Why extend?

If you've been paying attention at all while reading through this book, as well as to the code examples presented within it, you undoubtedly have noted that adopting jQuery for use in our pages has a profound effect on how script is written within a page.

The use of jQuery promotes a certain style for a page's code, frequently in the guise of forming a wrapped set of elements and then applying a jQuery command, or chain of commands, to that set. When writing our own code, we can write it however we please, but most experienced developers agree that having all of the code on a site, or at least the great majority of it, adhere to a consistent style is a good practice.

So one good reason to pattern our code as jQuery extensions is to help maintain a consistent code style throughout the site.

Not reason enough? Need more? The whole point of jQuery is to provide a set of reusable tools and APIs. The creators of jQuery carefully planned the design of the library and the philosophy of how the tools are arranged to promote reusability. By following the precedent set by the design of these tools, we automatically reap the benefit of the planning that went into these designs—a compelling second reason to write our code as jQuery extensions.

Still not convinced? The final reason we'll consider (though it's quite possible others could list even more reasons) is that, by extending jQuery, we can leverage the existing code base that jQuery makes available to us. For example, by creating new jQuery commands (wrapper methods), we automatically inherit the use of jQuery's powerful selector mechanism. Why would we write everything from scratch when we can layer upon the powerful tools jQuery already provides?

Given these reasons, it's easy to see that writing our reusable components as jQuery extensions is a good practice and a smart way of working. In the remainder

of this chapter, we'll examine the guidelines and patterns that allow us to create jQuery plugins and we'll create a few of our own. In the following chapter, which covers a completely different subject (Ajax), we'll see even more evidence that creating our own reusable components as jQuery plugins in real-world scenarios helps to keep the code consistent and makes it a whole lot easier to write those components in the first place.

But first, the rules...

7.2 The jQuery plugin authoring guidelines

Sign! Sign! Everywhere a sign! Blocking out the scenery, breaking my mind. Do this! Don't do that! Can't you read the sign?

—Five Man Electric Band, 1971

Although the Five Man Electric Band may have lyrically asserted an anti-establishment stance against rules back in 1971, sometimes rules are a good thing. Without any, chaos would reign.

Such it is with the rules—more like common sensical guidelines—governing how to successfully extend jQuery with our own plugin code. These guidelines help us ensure, not only that our new code will plug into the jQuery architecture properly, but also that it will work and play well with other jQuery plugins and even other JavaScript libraries.

Extending jQuery takes one of two forms:

- Utility functions defined directly on `$` (an alias for jQuery)
- Methods to operate on a jQuery wrapped set (so-called jQuery *commands*)

In the remainder of this section, we'll go over some guidelines common to both types of extensions. Then in the following sections, we'll tackle the guidelines and techniques specific to writing each type of plugin element.

7.2.1 Naming files and functions

To Tell the Truth was an American game show, first airing in the 1950's, in which multiple contestants claimed to be the same person with the same name, and a panel of celebrities was tasked with determining which person was whom they claimed to be. Although fun for a television audience, *name collisions* are not fun at all when it comes to programming.

We'll discuss avoiding name collisions *within* our plugins, but first let's address naming the files within which we'll write our plugins so that they do not conflict with other files.

The guideline recommended by the jQuery team is simple but effective, advocating the following format:

- Prefix the filename with *jquery*.
- Follow that with the name of the plugin.
- Conclude with *.js*.

For example, if we write a plugin that we want to name Fred, our JavaScript filename for this plugin is

```
jquery.fred.js
```

The use of the *jquery.* prefix eliminates any possible name collisions with files intended for use with other libraries. After all, anyone writing non-jQuery plugins has no business using the *jquery.* prefix.

But that leaves the plugin name itself still open for contention *within* the jQuery community.

When we're writing plugins for our own use, all we need to do is avoid conflicts with any other plugins that we plan to use. But when writing plugins that we plan to publish for others to use, we need to avoid conflicts with any other plugin that's already published.

The best way to avoid conflicts is to stay in tune with the goings-on within the jQuery community. A good starting point is the page at <http://docs.jquery.com/Plugins>; but, beyond being aware of what's already out there, there are other precautions we can take.

One way to ensure that our plugin filenames are unlikely to conflict with others is to sub-prefix them with a name that's unique to us or our organization. For example, all of the plugins developed in this book use the filename prefix *jquery.jqia* (*jqia* being short for *jQuery in Action*) to help make sure that they won't conflict with anyone else's plugin filenames should anyone wish to use them in their own web applications. Likewise, the files for the jQuery Form Plugin begin with the prefix *jquery.form*. Not all plugins follow this convention, but as the number of plugins increases, it will become more and more important to follow such conventions.

Similar considerations need to be taken with the *names* we give to our functions, whether they're new utility functions or methods on the jQuery wrappers.

When creating plugins for our own use, we're usually aware of what other plugins we'll use; it's an easy matter to avoid any naming collisions. But what if we're creating our plugins for public consumption? Or what if our plugins, that we initially intended to use privately, turn out to be so useful that we want to share them with the rest of the community?

Once again, familiarity with the plugins that already exist will go a long way in avoiding API collisions, but we also encourage gathering collections of related functions under a common prefix (similar to the proposal for filenames) to avoid cluttering the namespace.

Now, what about conflicts with that \$?

7.2.2 Beware the \$

“Will the real \$ please stand up?”

Having written a fair amount of jQuery code, we've seen how handy it is to use the \$ alias in place of jQuery. But when writing plugins that may end up in other people's pages, we can't be quite so cavalier. As plugin authors, we have no way of knowing whether a page author intends to use the \$.noConflict() function to allow the \$ alias to be usurped by another library.

We could employ the sledgehammer approach and use the jQuery name in place of the \$ alias, but dang it, we *like* using \$ and are loath to give up on it so easily.

Section 6.2 discussed an idiom often used to make sure that the \$ alias referred to the jQuery name in a localized manner without affecting the remainder of the page, and this little trick can also be (and often is) employed when defining jQuery plugins as follows:

```
(function($){  
  //  
  // Plugin definition goes here  
  //  
})(jQuery);
```

By passing jQuery to a function that defines the parameter as \$, \$ is guaranteed to reference jQuery within the body of the function.

We can now happily use \$ to our heart's content in the definition of the plugin.

Before we dive into learning how to add new elements to jQuery, let's look at one more technique plugin authors are encouraged to use.

7.2.3 Taming complex parameter lists

Most plugins tend to be simple affairs that require few, if any, parameters. We've seen ample evidence of this in the vast majority of the core jQuery methods and functions, which either take a small handful of parameters or none at all. Intelligent defaults are supplied when optional parameters are omitted, and parameter order can even take on a different meaning when some optional parameters are omitted.

The `bind()` method is a good example; if the optional data parameter is omitted, the listener function, which is normally specified as the third parameter, can be supplied as the second. The dynamic and interpretive nature of JavaScript allows us to write such flexible code, but this sort of thing can start to break down and get complex (both for page authors and ourselves as the plugin authors) as the number of parameters grows larger. The possibility of a breakdown increases when many of the parameters are optional.

Consider a somewhat complex function whose signature is as follows:

```
function complex(p1,p2,p3,p4,p5,p6,p7) {
```

This function accepts seven arguments; let's say that all but the first are optional. There are too many optional arguments to make any intelligent guesses about the intention of the caller when optional parameters are omitted (as we saw with the `bind()` method). If a caller of this function is only omitting trailing parameters, this isn't much of a problem because the optional trailing arguments can be detected as nulls. But what if the caller wants to specify `p7` but let `p2` through `p6` default? Callers would need to use placeholders for any omitted parameters and write

```
complex(valueA,null,null,null,null,null,valueB);
```

Yuck! Even worse is a call such as

```
complex(valueA,null,valueC,valueD,null,null,valueB);
```

along with other variations of this nature. Page authors using this function are forced to carefully keep track of counting nulls and the order of the parameters; plus, the code is difficult to read and understand.

But short of not allowing so many options to the caller, what can we do?

Again, the flexible nature of JavaScript comes to the rescue; a pattern that allows us to tame this chaos has arisen among the page-authoring communities—the *options hash*.

Using this pattern, optional parameters are gathered into a *single* parameter in the guise of a JavaScript Object instance whose property name/value pairs serve as the optional parameters.

Using this technique, our first example could be written as

```
complex(valueA, {p7: valueB});
```

And the second as the following:

```
complex(valueA, {  
  p3: valueC,  
  p4: valueD,  
  p7: valueB  
});
```

Much better!

We don't have to account for omitted parameters with placeholder nulls, and we also don't need to count parameters; each optional parameter is conveniently labeled so that it's clear to see exactly what it represents (when we use better parameter names than p1 through p7, that is).

Although this is obviously a great advantage to the caller of our complex functions, what about the ramifications for *us* as the function authors? As it turns out, we've already seen a jQuery-supplied mechanism that makes it easy for us to gather these optional parameters together and to merge them with default values. Let's reconsider our complex example function with a required parameter and six options. The new, simplified signature is

```
complex(p1,options)
```

Within our function, we can merge those options with default values with the handy `$.extend()` utility function. Consider the following:

```
function complex(p1,options) {  
  var settings = $.extend({  
    option1: defaultValue1,  
    option2: defaultValue2,  
    option3: defaultValue3,  
    option4: defaultValue4,  
    option5: defaultValue5,  
    option6: defaultValue6  
  },options||{});  
  // remainder of function...  
}
```

By merging the values passed to us by the page author in the `options` parameter with an object containing all the available options with their default values, the

settings variable ends up with all possible option default values superseded by any explicit values specified by the page author.

Note that we guard against an `options` object that's `null` or `undefined` with `|| {}`, which supplies an empty object if `options` evaluates to `false` (as we know `null` and `undefined` do).

Easy, versatile, and caller-friendly!

We'll see examples of this pattern in use later in this chapter and in jQuery functions that will be introduced in chapter 8. But for now, let's finally look at how we extend jQuery with our own utility functions.

7.3 Writing custom utility functions

In this book, we use the term *utility function* to describe functions defined as properties of jQuery (and therefore `$`). These functions are not intended to operate on DOM elements—that's the job of methods defined to operate on a jQuery wrapped set—but to either operate on non-element JavaScript objects or perform some other nonobject-specific operation. Some examples we've seen of these types of function are `$.each()` and `$.noConflict()`.

In this section, we'll learn how to add our own similar functions.

Adding a function as a property to an object instance is as easy as declaring the function and assigning it to the object property. (If this seems like black magic to you and you have not yet read through the appendix, now would be a good time to do so.) Creating a trivial custom utility function should be as easy as

```
$.say = function(what) { alert('I say '+what); }
```

And in truth, it *is* that easy. But this manner of defining a utility function isn't without its pitfalls; remember our discussion in section 7.2.2 regarding the `$`? What if some page author is including this function on a page that uses Prototype and has called `$.noConflict()`? Rather than add a jQuery extension, we'd create a method on Prototype's `$()` function. (Get thee to the appendix if the concept of a *method* of a function makes your head hurt.)

This isn't a problem for a private function that we know will never be shared, but even then, what if some future changes to the pages usurp the `$`? It's a good idea to err on the side of caution.

One way to ensure that someone stomping on `$` doesn't also stomp on us is not using the `$` at all. We could write our trivial function as

```
jQuery.say = function(what) { alert('I say '+what); }
```

This seems like an easy way out but proves to be less than optimal for more complex functions. What if the function body utilizes lots of jQuery methods and functions internally to get its job done? We'd need to use jQuery rather than \$ throughout the function. That's rather wordy and inelegant; besides, once we use the \$, we don't want to let it go!

So looking back to the idiom we introduced in section 7.2.2, we can safely write our function as follows:

```
(function($){
  $.say = function(what) { alert('I say '+what); }
})(jQuery);
```

We highly encourage using this pattern (even though it may seem like overkill for such a trivial function) because it protects the use of \$ when declaring and defining the function. Should the function ever need to become more complex, we can extend and modify it without wondering whether it's safe to use the \$ or not.

With this pattern fresh in our minds, let's implement a non-trivial utility function of our own.

7.3.1 Creating a data manipulation utility function

Often, when emitting fixed-width output, it's necessary to take a numeric value and format it to fit into a fixed-width field (where *width* is defined as number of characters). Usually such operations will right-justify the value within the fixed-width field and prefix the value with enough *fill characters* to make up any difference between the length of the value and the length of the field.

Let's write such a utility function that's defined with the following syntax:

Function syntax: \$.toFixedWidth

\$.toFixedWidth(value, length, fill)

Formats the passed value as a fixed-width field of the specified length. An optional fill character can be supplied. If the numeric value exceeds the specified length, its higher order digits will be truncated to fit the length.

Parameters

- value** (Number) The value to be formatted.
- length** (Number) The length of the resulting field.
- fill** (String) The fill character used when front-padding the value. If omitted, 0 is used.

Returns

The fixed-width field.

The implementation of this function is shown in listing 7.1.

Listing 7.1 Implementation of the `$.toFixedWidth()` utility function

```
(function($){
  $.toFixedWidth = function(value,length,fill) {
    var result = value.toString();           ❶
    if (!fill) fill = '0';                  ❷
    var padding = length - result.length;
    if (padding < 0) {                       ❸
      result = result.substr(-padding);
    }
    else {
      for (var n = 0; n < padding; n++)      ❹
        result = fill + result;
    }
    return result;                           ❺
  };
})(jQuery);
```

This function is simple and straightforward. The passed value is converted to its string equivalent, and the fill character is determined either from the passed value or the default of 0 ❶. Then, we compute the amount of padding needed ❷.

If we end up with negative padding (the result is longer than the passed field length), we truncate from the beginning of the result to end up with the specified length ❸; otherwise, we pad the beginning of the result with the appropriate number of fill characters ❹ prior to returning it as the result of the function ❺.

Simple stuff, but it serves to show how easily we can add a utility function. And, as always, there's room for improvement. Consider the following exercises:

- As with most examples in books, the error checking is minimal to focus on the lesson at hand. How would you beef up the function to account for caller errors such as not passing numeric values for `value` and `length`? What if they don't pass them at all?
- We were careful to truncate numeric values that were too long in order to guarantee that the result was always the specified length. But, if the caller passes more than a single-character string for the fill character, all bets are off. How would you handle that?
- What if you don't want to truncate too-long values?

Now, let's tackle a more complex function in which we can make use of the `$.toFixedWidth()` function that we just wrote.

7.3.2 Writing a date formatter

If you've come to the world of client-side programming from the server, one of the things you may have longed for is a simple date formatter; something that the JavaScript `Date` type doesn't provide. Because such a function would operate on a `Date` instance, rather than any DOM element, it's a perfect candidate for a utility function. Let's write one that uses the following syntax:

Function syntax: `$.formatDate`

`$.formatDate(date, pattern)`

Formats the passed date according to the supplied pattern. The tokens that are substituted in the pattern are as follows:

- yyyy: the 4-digit year
- yy: the 2-digit year
- MMMM: the full name of the month
- MMM: the abbreviated name of the month
- MM: the month number as a 0-filled, 2-character field
- M: the month number
- dd: the day in the month as a 0-filled, 2-character field
- d: the day in the month
- EEEE: the full name of the day of the week
- EEE: the abbreviated name of the day of the week
- a: the meridium (AM or PM)
- HH: the 24-hour clock hour in the day as a 2-character, 0-filled field
- H: the 24-hour clock hour in the day
- hh: the 12-hour clock hour in the day as a 2-character, 0-filled field
- h: the 12-hour clock hour in the day
- mm: the minutes in the hour as a 2-character, 0-filled field
- m: the minutes in the hour
- ss: the seconds in the minute as a 2-character, 0-filled field
- s: the seconds in the minute
- S: the milliseconds in the second as a 3-character, 0-filled field

Parameters

- `date` (Date) The date to be formatted.
- `pattern` (String) The pattern to format the date into. Any characters not matching pattern tokens are copied as-is to the result.

Returns

The formatted date.

The implementation of this function is shown in listing 7.2. We're not going to go into great detail regarding the algorithm used to perform the formatting (after all, this isn't an algorithms book), but we're going to use this implementation to

point out some interesting tactics that we can use when creating a somewhat complex utility function.

Listing 7.2 Implementation of the \$.formatDate() utility function

```
(function($){
  $.formatDate = function(date,pattern) {
    var result = [];
    while (pattern.length > 0) {
      $.formatDate.patternParts.lastIndex = 0;
      var matched = $.formatDate.patternParts.exec(pattern);
      if (matched) {
        result.push(
          $.formatDate.patternValue[matched[0]].call(this,date)
        );
        pattern = pattern.slice(matched[0].length);
      } else {
        result.push(pattern.charAt(0));
        pattern = pattern.slice(1);
      }
    }
    return result.join('');
  };

  $.formatDate.patternParts =
    /^(\yy(\yy)?|M(M(M(M)?)?)?|d(d)?|EEE(E)?|a|H(H)?|h(h)?|m(m)?|s(s)?|S)/;

  $.formatDate.monthNames = [
    'January', 'February', 'March', 'April', 'May', 'June', 'July',
    'August', 'September', 'October', 'November', 'December'
  ];

  $.formatDate.dayNames = [
    'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
    'Saturday'
  ];

  $.formatDate.patternValue = {
    yy: function(date) {
      return $.toFixedWidth(date.getFullYear(), 2);
    },
    yyyy: function(date) {
      return date.getFullYear().toString();
    },
    MMMM: function(date) {
      return $.formatDate.monthNames[date.getMonth()];
    },
    MMM: function(date) {
      return $.formatDate.monthNames[date.getMonth()].substr(0,3);
    },
    MM: function(date) {

```

1 Implements the main body of the function

2 Defines the regular expression

3 Provides the name of the months

4 Provides the name of the days

5 Collects token-to-value translation functions

```
        return $.toFixedWidth(date.getMonth() + 1,2);
    },
    M: function(date) {
        return date.getMonth()+1;
    },
    dd: function(date) {
        return $.toFixedWidth(date.getDate(),2);
    },
    d: function(date) {
        return date.getDate();
    },
    EEEE: function(date) {
        return $.formatDate.dayNames[date.getDay()];
    },
    EEE: function(date) {
        return $.formatDate.dayNames[date.getDay()].substr(0,3);
    },
    HH: function(date) {
        return $.toFixedWidth(date.getHours(),2);
    },
    H: function(date) {
        return date.getHours();
    },
    hh: function(date) {
        var hours = date.getHours();
        return $.toFixedWidth(hours > 12 ? hours - 12 : hours,2);
    },
    h: function(date) {
        return date.getHours() % 12;
    },
    mm: function(date) {
        return $.toFixedWidth(date.getMinutes(),2);
    },
    m: function(date) {
        return date.getMinutes();
    },
    ss: function(date) {
        return $.toFixedWidth(date.getSeconds(),2);
    },
    s: function(date) {
        return date.getSeconds();
    },
    S: function(date) {
        return $.toFixedWidth(date.getMilliseconds(),3);
    },
    a: function(date) {
        return date.getHours() < 12 ? 'AM' : 'PM';
    }
    };
})(jQuery);
```

The most interesting aspect of this implementation, aside from a few JavaScript tricks used to keep the amount of code in check, is that the function ❶ needs some ancillary data to do its job—in particular:

- A regular expression used to match tokens in the pattern ❷
- A list of the English names of the months ❸
- A list of the English names of the days ❹
- A set of sub-functions designed to provide the value for each token type given a source date ❺

We could have included each of these as `var` definitions within the function body, but that would clutter an already somewhat involved algorithm; and because they're constants, it makes sense to segregate them from variable data.

We don't want to pollute the global namespace, or even the `$` namespace, with a bunch of names needed only by this function, so we make these declarations properties of our new function itself. Remember, JavaScript functions are first-class objects, and they can have their own properties like any other JavaScript object.

As for the formatting algorithm itself? In a nutshell, it operates as follows:

- Creates an array to hold portions of the result.
- Iterates over the pattern, consuming identified token and non-token characters until it has been completely inspected.
- Resets the regular expression (stored in `$.formatDate.patternParts`) on each iteration by setting its `lastIndex` property to 0.
- Tests the regular expression for a token match against the current beginning of the pattern.
- Calls the function in the `$.formatDate.patternValue` collection of conversion functions to obtain the appropriate value from the `Date` instance if a match occurs. This value is pushed onto the end of the results array, and the matched token is removed from the beginning of the pattern.
- Removes the first character from the pattern and adds it to the end of the results array if a token isn't matched at the current beginning of the pattern.
- Joins the results array into a string and returns it as the value of the function when the entire pattern has been consumed.

Note that the conversion functions in the `$.formatDate.patternValue` collection make use of the `$.toFixedWidth()` function that we created in the previous section.

You'll find both of these functions in the file `chapter7/jquery.jqia.dateFormat.js` and a rudimentary page to test it at `chapter7/test.dateFormat.html`.

Operating on run-of-the-mill JavaScript objects is all well and good, but the real power of jQuery lies in the wrapper methods that operate on a set of DOM elements collected via the power of jQuery selectors. Next, let's see how we can add our own powerful wrapper methods.

7.4 Adding new wrapper methods

The true power of jQuery lies in the ability to easily and quickly select and operate on DOM elements. Luckily, we can extend that power by adding wrapper methods of our own that manipulate selected DOM elements as we deem appropriate. By adding wrapper methods, we automatically gain the use of the powerful jQuery selectors to pick and choose which elements are to be operated on without having to do all the work ourselves.

Given what we know about JavaScript, we probably could have figured out on our own how to add utility functions to the `$` namespace, but that's not true of wrapper functions. There's a tidbit of jQuery-specific information that we need to know; to add wrapper methods to jQuery, we must assign them as properties to an object named `fn` in the `$` namespace.

The general pattern for creating a wrapper functions is

```
$.fn.wrapperFunctionName = function(params){function-body};
```

Let's concoct a trivial wrapper method to set the color of the matched DOM elements to blue.

```
(function($){
  $.fn.makeItBlue = function() {
    return this.css('color','blue');
  }
})(jQuery);
```

As with utility functions, we make the declaration within an outer function that guarantees that `$` is an alias to `jQuery`. But unlike utility functions, we create the new wrapper method as a property of `$.fn` rather than of `$`.

NOTE If you're familiar with object-oriented JavaScript and its prototype-based class declarations, you might be interested to know that `$.fn` is merely an alias for the `prototype` property of the `jQuery` constructor function.

Within the body of the method, the function context (`this`) refers to the wrapped set. We can use all of the predefined jQuery commands on it; as in this example, we call the `css()` command on the wrapped set to set the `color` to `blue` for all matched DOM elements.

WARNING The function context (`this`) within the main body of a wrapper method refers to the wrapped set, but when inline functions are declared within this function, they each have their own function contexts. You must take care when using `this` under such circumstances to make sure that it's referring to what you think it is! For example, if you use `each()` with its iterator function, `this` within the iterator function references the DOM element for the current iteration.

We can do almost anything we like to the DOM elements in the wrapped set, but there is one *very* important rule when defining new wrapper methods; unless the function is intended to return a specific value, it should always return the wrapped set as its return value. This allows our new command to take part in any jQuery command chains. In our example, because the `css()` command returns the wrapped set, we simply return the result of the call to `css()`.

In this example, we apply the jQuery `css()` command to all the elements in the wrapped set by applying it to `this`. If, for some reason, we need to deal with each wrapped element individually (perhaps because we need to make conditional processing decisions), the following pattern can be used:

```
(function($){
  $.fn.someNewMethod = function() {
    return this.each(function(){
      //
      // Function body goes here -- this refers to individual
      // elements
      //
    });
  }
})(jQuery);
```

In this pattern, the `each()` command is used to iterate over every individual element in the wrapped set. Note that, within the iterator function, `this` refers to the current DOM element rather than the entire wrapped set. The wrapped set returned by `each()` is returned as the new method's value so that this method can participate in chaining.

That's all there is to it, but (isn't there always a *but*?) there are some techniques we should be aware of when creating more involved jQuery wrapper methods. Let's define a couple more plugin methods of greater complexity to examine those techniques.

7.4.1 Applying multiple operations in a wrapper method

Let's develop another new plugin method that performs more than a single operation on the wrapped set. Imagine that we need to be able to flip the read-only status of text fields within a form and to simultaneously and consistently affect the appearance of the field. We could easily chain a couple of existing jQuery commands together to do this, but we want to be neat and tidy about it and bundle these operations together into a single method.

We'll name our new command `setReadOnly()`, and its syntax is as follows:

Command syntax: `setReadOnly`

`setReadOnly(state)`

Sets the read-only status of wrapped text fields to the state specified by `state`. The opacity of the fields will be adjusted: 100% if not read-only or 50% if read-only. Any elements in the wrapped set other than text fields are ignored.

Parameters

`state` (Boolean) The read-only state to set. If `true`, the text fields are made read-only; otherwise, the read-only status is cleared.

Returns

The wrapped set.

The implementation of this plugin is shown in listing 7.3 and can be found in the file `chapter7/jquery.jqia.setreadonly.js`.

Listing 7.3 Implementation of the `setReadOnly()` plugin method

```
(function($){
  $.fn.setReadOnly = function(readonly) {
    return this.filter('input:text')
      .attr('readonly',readonly)
      .css('opacity', readonly ? 0.5 : 1.0);
  }
})(jQuery);
```

This example is only slightly more complicated than our initial example, but exhibits the following additional key concepts:

- A parameter is passed that affects how the method operates.
- Three jQuery commands are applied to the wrapped set by use of jQuery chaining.
- The new command can participate in a jQuery chain because it returns the wrapped set as its value.
- The `filter()` command is used to ensure that, no matter what set of wrapped elements the page author applied this method to, only text fields are affected.

How might we put this method to use?

Often, when defining an online order form, we may need to allow the user to enter two sets of address information: one for where the order is to be shipped and one for the billing information. Much more often than not, these two addresses are going to be the same; making the user enter the same information twice decreases our user-friendliness factor to less than we'd want it to be.

We could write our server-side code to assume that the billing address is the same as the shipping address if the form is left blank, but let's assume that our product manager is a bit paranoid and would like something more overt on the part of the user.

We'll satisfy him by adding a check box to the billing address that indicates whether the billing address is the same as the shipping address. When this box is checked, the billing address fields will be copied from the shipping fields and then made read-only. Unchecking the box will clear the value and read-only status from the fields.

Figure 7.1 shows a test form in its before and after states.

The page for this test form is available in the file `chapter7/test.setreadonly.html` and is shown in listing 7.4.

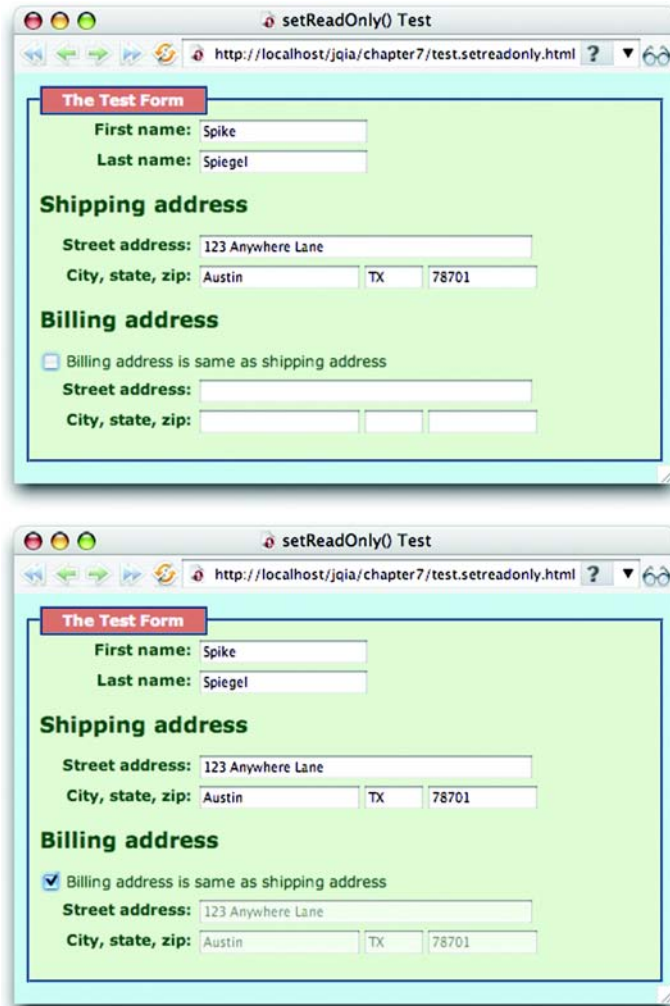


Figure 7.1
The Test Form prior to clicking the check box and after clicking the check box

Listing 7.4 The implementation of the form for testing the new `setReadOnly()` command

```
<html>
<head>
  <title>setReadOnly() Test</title>
  <link rel="stylesheet" type="text/css" href="../common.css">
  <link rel="stylesheet" type="text/css"
    href="test.setreadonly.css">
  <script type="text/javascript"
    src="../scripts/jquery-1.2.1.js"></script>
```



```

<script type="text/javascript"
  src="jquery.jqia.setreadonly.js"></script>
<script type="text/javascript">
  $(function(){
    $('#sameAddressControl').click(function(){
      var same = this.checked;
      $('#billAddress').val(same ? $('#shipAddress').val():'');
      $('#billCity').val(same ? $('#shipCity').val():'');
      $('#billState').val(same ? $('#shipState').val():'');
      $('#billZip').val(same ? $('#shipZip').val():'');
      $('#billingAddress input')
        .setReadOnly(same);
    });
  });
</script>
</head>

<body>
  <fieldset>
    <legend>The Test Form</legend>
    <div>
      <form name="testForm">
        <div>
          <label>First name:</label>
          <input type="text" name="firstName" id="firstName"/>
        </div>
        <div>
          <label>Last name:</label>
          <input type="text" name="lastName" id="lastName"/>
        </div>
        <div id="shippingAddress">
          <h2>Shipping address</h2>
          <div>
            <label>Street address:</label>
            <input type="text" name="shipAddress"
              id="shipAddress"/>
          </div>
          <div>
            <label>City, state, zip:</label>
            <input type="text" name="shipCity" id="shipCity"/>
            <input type="text" name="shipState" id="shipState"/>
            <input type="text" name="shipZip" id="shipZip"/>
          </div>
        </div>
        <div id="billingAddress">
          <h2>Billing address</h2>
          <div>
            <input type="checkbox" id="sameAddressControl"/>
            Billing address is same as shipping address
          </div>
          <div>
            <label>Street address:</label>

```

❶ Applies the new plugin

```
        <input type="text" name="billAddress"
            id="billAddress" />
    </div>
    <div>
        <label>City, state, zip:</label>
        <input type="text" name="billCity" id="billCity" />
        <input type="text" name="billState" id="billState" />
        <input type="text" name="billZip" id="billZip" />
    </div>
</div>
</form>
</div>
</fieldset>
</body>
</html>
```

We won't belabor the operation of this page, as it's relatively straightforward. The only truly interesting aspect of this page is the click handler attached to the check box in the ready handler. When the state of the check box is changed by a click we

- 1 Copy the checked state into variable `same` for easy reference in the remainder of the listener.
- 2 Set the values of the billing address fields. If they are to be the same, we set the values from the corresponding fields in the shipping address information. If not, we clear the fields.
- 3 Call the new `setReadOnly()` command **1** on all input fields in the billing address container.

But, oops! We were a little sloppy with that last step. The wrapped set that we create with `$('#billingAddress input')` contains not only the text fields in the billing address block but the check box too. The check box element doesn't have read-only semantics, but it can have its opacity changed—definitely not our intention!

Luckily, this sloppiness is countered by the fact that we were *not* sloppy when defining our plugin. Recall that we filtered out all but text fields before applying the remainder of the commands in that method. We highly recommend such attention to detail, particularly for plugins that are intended for public consumption.

What are some ways that this command could be improved? Consider making the following changes:

- We forgot about text areas! How would you modify the code to include text areas along with the text fields?

- The opacity levels applied to the fields in either state are hard-coded into the function. This is hardly caller-friendly. Modify the method to allow the levels to be caller-supplied.
- Oh heck, why force the page author to accept only the ability to affect opacity? How would you modify the method to allow the page author to determine what the renditions for the fields should be in either state?

Now let's take on an even more complex plugin.

7.4.2 Retaining state within a wrapper method

Everybody loves a slideshow!

At least on the web. Unlike hapless after-dinner guests forced to sit through a mind-numbingly endless display of badly focused vacation photos, visitors to a web slideshow can leave whenever they like without hurting anyone's feelings!

For our more complex plugin example, we're going to develop a jQuery command that will easily allow a page author to whip up a quick slideshow page. We'll create a jQuery plugin, which we'll name the *Photomatic*, and then we'll whip up a test page to put it through its paces. When complete, this test page will appear as shown in figure 7.2.

This page sports the following components:

- A set of thumbnail images
- A full-sized photo of one of the images available in the thumbnail list
- A set of buttons to move through the slideshow

The behaviors of the page are as follows:

- Clicking any thumbnail displays the corresponding full-sized image.
- Clicking the full-sized image displays the next image.
- Clicking any button performs the following operations:
 - First—Displays the first image
 - Previous—Displays the previous image
 - Next—Displays the next image
 - Last—Displays the last image
- Any operation that moves off the end of the list of images wraps back to the other end; clicking Next while on the last image displays the first image and vice versa.

We also want to grant the page authors as much freedom for layout and styling as possible; we'll define our plugin so that the page authors can set up the elements

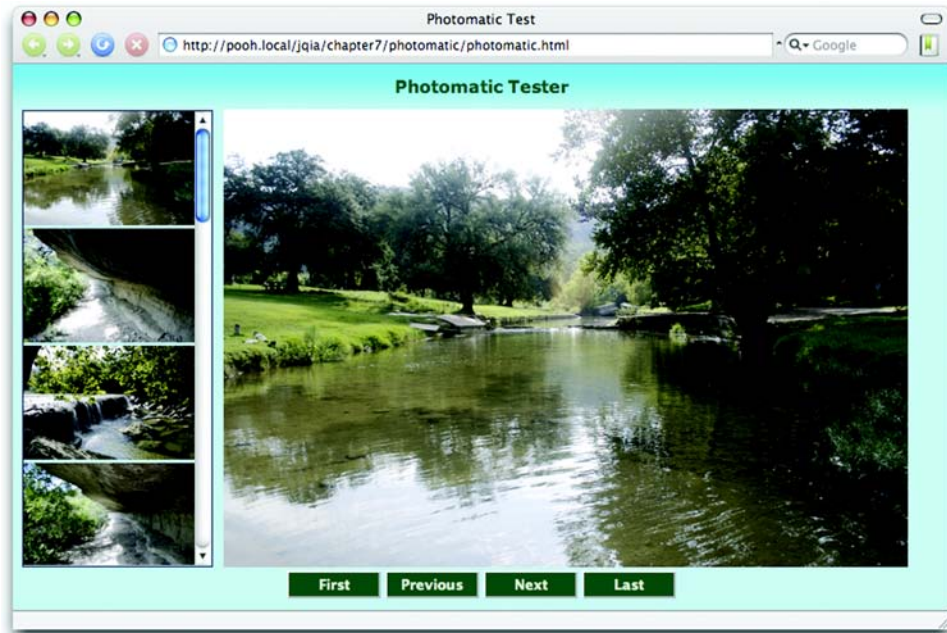


Figure 7.2 The Photomatic Tester that we'll use to put our plugin through its paces

in any manner that they would like and then tell us which page element should be used for each purpose. Furthermore, in order to give the page authors as much leeway as possible, we'll define our plugin so that the authors can provide any wrapped set of images to serve as thumbnails. Usually, thumbnails will be gathered together as in our test page, but page authors are free to identify any image on the page as a thumbnail.

To start, let's introduce the syntax for the Photomatic Plugin.

Command syntax: photomatic

`photomatic(settings)`

Instruments the wrapped set of thumbnails, as well as page elements identified in the settings hash, to operate as Photomatic controls.

Parameters

`settings` (Object) An object hash that specifies the settings for the Photomatic. See table 7.1 for details.

Returns

The wrapped set.

Because we have a non-trivial number of parameters to control the operation of the Photomatic (many of which can be defaulted), we utilize an object hash to pass them in as outlined in section 7.2.3. The possible settings that we can specify are shown in table 7.1.

Table 7.1 The settings properties for the `Photomatic()` plugin command

Setting name	Description
<code>firstControl</code>	(String Object) Either a reference to or jQuery selector that identifies the DOM element(s) to serve as a <i>first</i> control. If omitted, no control is instrumented.
<code>lastControl</code>	(String Object) Either a reference to or jQuery selector that identifies the DOM element(s) to serve as a <i>last</i> control. If omitted, no control is instrumented.
<code>nextControl</code>	(String Object) Either a reference to or jQuery selector that identifies the DOM element(s) to serve as a <i>next</i> control. If omitted, no control is instrumented.
<code>photoElement</code>	(String Object) Either a reference to or jQuery selector that identifies the <code></code> element that's to serve as the full-sized photo display. If omitted, defaults to the jQuery selector <code>'#photomaticPhoto'</code> .
<code>previousControl</code>	(String Object) Either a reference to or jQuery selector that identifies the DOM element(s) to serve as a <i>previous</i> control. If omitted, no control is instrumented.
<code>transformer</code>	(Function) A function used to transform the URL of a thumbnail image into the URL of its corresponding full-sized photo image. If omitted, the default transformation substitutes all instances of <i>thumbnail</i> with <i>photo</i> in the URL.

With a nod to the notion of *test-driven development*, let's create the test page for this plugin *before* we dive into creating the Photomatic Plugin itself. The code for this page, available in the file `chapter7/photomatic/photomatic.html`, is shown in listing 7.5.

Listing 7.5 The test page that creates the Photomatic display shown in figure 7.2

```

<html>
  <head>
    <title>Photomatic Test</title>
    <link rel="stylesheet" type="text/css" href="../../common.css">
    <link rel="stylesheet" type="text/css" href="photomatic.css">
    <script type="text/javascript"
      src="../../scripts/jquery-1.2.1.js"></script>
    <script type="text/javascript"
      src="jquery.jqia.photomatic.js"></script>
    <script type="text/javascript">
      $(function(){
        $('#thumbnails img').photomatic({

```

1 Invokes the Photomatic Plugin

```

        photoElement: '#photo',
        previousControl: '#previousButton',
        nextControl: '#nextButton',
        firstControl: '#firstButton',
        lastControl: '#lastButton'
    });
    });
</script>
</head>

<body>
<h1>Photomatic Tester</h1>
<div id="thumbnails">
    
    
    
    
    
    
</div>
<div id="photoContainer">
    <img id="photo" src="" />
</div>
<div id="buttonBar">
    <button type="button" id="firstButton">First</button>
    <button type="button" id="previousButton">Previous</button>
    <button type="button" id="nextButton">Next</button>
    <button type="button" id="lastButton">Last</button>
</div>
</body>
</html>

```

② Contains thumbnail images

③ Defines the image element for full-sized photos

④ Contains elements to serve as controls

If that looks simpler than you thought it would, you shouldn't be surprised at this point. By applying the principles of Unobtrusive JavaScript and by keeping all style information in an external style sheet, our markup is tidy and simple. In fact, even the on-page script has a tiny footprint, consisting of a single statement that invokes our plugin ①.

The HTML markup consists of a container that holds the thumbnail images ②, an image element (initially sourceless) to hold the full-sized photo ③, and a collection of buttons ④ that will control the slideshow. Everything else is handled by our new plugin.

Let's develop that now.

To start, let's set out a skeleton (we'll fill in the details as we go along). Our starting point should look rather familiar by now because it follows the same patterns we've been using so far.

```
(function($){
  $.fn.photomatic = function(callerSettings) {
  };
})(jQuery);
```

This defines our initially empty wrapper function, which (as expected from our syntax description) accepts a single hash parameter named `callerSettings`. First, within the body of the function, we merge these caller settings with the default settings as described by table 7.1. This will give us a single settings object that we can refer to throughout the remainder of the method.

We perform this merge operation using the following idiom (that we've already seen a few times):

```
var settings = $.extend({
  photoElement: '#photomaticPhoto',
  transformer: function(name) {
    return name.replace(/thumbnail/, 'photo');
  },
  nextControl: null,
  previousControl: null,
  firstControl: null,
  lastControl: null
}, callerSettings||{});
```

After the execution of this statement, the `settings` variable will contain the values supplied by the caller, with defaults supplied by the inline hash object. But we're not done with `settings` yet. Consider the `photoElement` property; it might contain a string specifying a jQuery selector (either the default or one supplied by the page authors), or it could be an object reference. We want to normalize that to something we know how to deal with. By adding the statement

```
settings.photoElement = $(settings.photoElement);
```

we create a wrapped set containing the photo element (or possibly multiple elements if the page authors so chose). Now, we have something consistent that we know how to deal with.

We're also going to need to keep track of a few things. In order to know what concepts like *next* image and *previous* image mean, we need not only a list of the thumbnail images but also an indicator that identifies the *current* image being displayed.

The list of thumbnail images is the wrapped set that this method is operating on—or, at least, it should be. We don't know what the page authors collected in the wrapped set, so we want to filter it down to only image elements; we can easily do this with a jQuery selector. But where should we store the list?

We could easily create another variable to hold it, but there's a lot to be said for keeping things corralled. Let's store the list as another property on `settings` as follows:

```
settings.thumbnails = this.filter('img');
```

Filtering the wrapped set (available via `this` in the method) for only image elements results in a new wrapped set (containing only `` elements) that we store in a property of `settings` that we name `thumbnails`.

Another piece of state that we need to keep track of is the *current* image. We'll keep track of that by maintaining an index into the list of thumbnails by adding another property to `settings` named `current` as follows:

```
settings.current = 0;
```

There is one more setup step that we need to take. If we're going to keep track of which photo is *current* by keeping track of its index, there will be at least one case where, given a reference to a thumbnail element, we'll need to know its index. The easiest way to handle that is to anticipate this need and to add an *expando* (custom property) to the thumbnail elements, recording their respective indexes. We do that with the following statement:

```
settings.thumbnails.each(function(n){this.index = n;});
```

This statement iterates through each of the thumbnail images, adding an `index` property to it that records its order in the list. Now that our initial state is set up, we're ready to move on to the meat of the plugin.

Wait a minute! *State?* How can we expect to keep track of state in a local variable within a function that's about to finish executing? Won't the variable and all our settings go out of scope when the function returns?

In general that might be true, but there is one case where such a variable sticks around for longer than its usual scope—when it's part of a *closure*. We've seen closures before, but if you're still shaky on them, please review the appendix. You must understand closures not only for completing the implementation of the Photomatic Plugin but also when creating anything but the most trivial of plugins.

When we think about the job remaining, we realize that we need to attach a number of event listeners to the controls and elements that we've taken such great pains to identify to this point. And each listener we define will create a closure that includes the `settings` variable, so we can rest assured that, even though `settings` may appear to be transient, the state that it represents will stick around and be available to all the listeners that we define.

Speaking of those listeners, here's the list of `click` event listeners that we need to attach to the various elements:

- Clicking a thumbnail photo will cause its full-sized version to be displayed.
- Clicking the full-sized photo will cause the next photo to be displayed.
- Clicking the element defined as the previous control will cause the previous image to be displayed.
- Clicking the next control will cause the next image to be displayed.
- Clicking the first control will cause the first image in the list to be displayed.
- Clicking the last control will cause the last image in the list to be displayed.

Looking over this list, we immediately note that all of these listeners have something in common; they all need to cause the full-sized photo of one of the thumbnail images to be displayed. And being the good and clever coders that we are, we want to factor out that common processing into a function so that we don't need to repeat code over and over again.

But how?

If we were writing normal on-page JavaScript, we could define a top-level function. If we were writing object-oriented JavaScript, we might define a method on a JavaScript object. But we're writing a jQuery plugin; where should we define implementation functions?

We don't want to infringe on either the global or the `$` namespace for a function that should only be called internally from our plugin code, so what can we do? We *could* define the function as a method of the plugin function itself (similar to what we did in the date formatter of listing 7.2); as first-class JavaScript objects, even functions can possess methods. But there's an even easier way.

Recall that our plugin function is defined within the body of another function—the function that we use to ensure that the `$` means jQuery. Therefore any local variables defined within that outer function become part of the closure defined by our plugin function. What if we define the implementation function, which we'll name `showPhoto()`, as a local variable in the outer function?

That solves our issue nicely! Our `showPhoto()` function will be available to the plugin function as part of its closure, but because it's declared locally to the outer function, it can't be seen from outside that function and has no opportunity to pollute any other namespace.

Outside of the plugin function, but inside the outer function, we define the `showPhoto()` function as follows:

```
var showPhoto = function(index) {
  settings.photoElement
    .attr('src',
          settings.transformer(settings.thumbnails[index].src));
  settings.current = index;
};
```

This function, when passed the `index` of the thumbnail whose full-sized photo is to be displayed, uses the values in the `settings` object to do the following:

- 1 Look up the `src` attribute of the thumbnail identified by `index`
- 2 Pass that value through the `transformer` function to convert it from a thumbnail URL to a photo URL
- 3 Assign the result of the transformation to the `src` attribute of the full-sized image element
- 4 Record the index of the displayed photo as the new current index

But before we break our arms patting ourselves on the back for our cleverness, we should know that there's still a problem. The `showPhoto()` function needs access to the `settings`, but that variable is defined *inside* the plugin function and isn't available outside of it.

“Holy scoped closures, Batman! How will we get out of this one?”

We could be inelegant about it and pass `settings` to the function as another parameter, but we're slyer than that. Because the problem is that `settings` isn't defined in the outer closure containing the implementation and plugin functions the simplest solution is to *move it there*. Doing so won't affect its availability to all of the inner closures, but it will ensure that it's available to both the plugin function and the implementation function or any other implementation functions that we might want to define. (Be aware that this technique assumes that we're only going to have one Photomatic instance on a page—a restriction that makes sense in this case. For more general cases, passing `settings` as a parameter would not impose this restriction.)

So we'll add the following to the outer function:

```
var settings;
```

And remove the `var` from the statement within the plugin function as follows:

```
settings = $.extend({
```

The settings variable is now available to both functions, and we're finally ready to complete the implementation of the plugin function. We define the listeners that we listed earlier with the following code:

```
settings.thumbnails.click(function(){ showPhoto(this.index); });
settings.photoElement.click(function(){
  showPhoto((settings.current + 1) % settings.thumbnails.length);
});
$(settings.nextControl).click(function(){
  showPhoto((settings.current + 1) % settings.thumbnails.length);
});
$(settings.previousControl).click(function(){
  showPhoto((settings.thumbnails.length + settings.current - 1) %
    settings.thumbnails.length);
});
$(settings.firstControl).click(function(){
  showPhoto(0);
});
$(settings.lastControl).click(function(){
  showPhoto(settings.thumbnails.length - 1);
});
```

Each of these listeners calls the `showPhoto()` function with a thumbnail index determined either by the index of the clicked thumbnail, the length of the list, or computed relative to the current index. (Note how the modulus operator is used to wrap the index values when they fall off either end of the list.)

We have two final tasks before we can declare success; we need to display the first photo in advance of any user action, and we need to return the original wrapped set so that our plugin can participate in jQuery command chains. We achieve these with

```
showPhoto(0);
return this;
```

Take a moment to do a short Victory Dance; we're finally done!

The completed plugin code, which you'll find in the file `chapter7/photomatic/jquery.jqia.photomatic.js`, is shown in listing 7.6.

Listing 7.6 The complete Photomatic Plugin implementation

```
(function($){
  var settings;

  $.fn.photomatic = function(callerSettings) {
    settings = $.extend({
      photoElement: '#photomaticPhoto',
      transformer: function(name) {
```

```

        return name.replace(/thumbnail/, 'photo');
    },
    nextControl: null,
    previousControl: null,
    firstControl: null,
    lastControl: null
}, callerSettings || {});
settings.photoElement = $(settings.photoElement);
settings.thumbnails = this.filter('img');
settings.thumbnails.each(function(n){this.index = n;});
settings.current = 0;
settings.thumbnails.click(function(){ showPhoto(this.index); });
settings.photoElement.click(function(){
    showPhoto((settings.current + 1) % settings.thumbnails.length);
});
$(settings.nextControl).click(function(){
    showPhoto((settings.current + 1) % settings.thumbnails.length);
});
$(settings.previousControl).click(function(){
    showPhoto((settings.thumbnails.length + settings.current - 1) %
        settings.thumbnails.length);
});
$(settings.firstControl).click(function(){
    showPhoto(0);
});
$(settings.lastControl).click(function(){
    showPhoto(settings.thumbnails.length - 1);
});
showPhoto(0);
return this;
};

var showPhoto = function(index) {
    settings.photoElement
        .attr('src',
            settings.transformer(settings.thumbnails[index].src));
    settings.current = index;
};

})(jQuery);

```

Boy, it seemed longer than 45 lines when we were working through it, didn't it?

This plugin is typical of jQuery-enabled code; it packs a big wallop in some compact code. But it serves to demonstrate an important set of techniques—using closures to maintain state across the scope of a jQuery plugin and to enable the creation of private implementation functions that plugins can define and use without resorting to any namespace infringements.

You're now primed and ready to write your own jQuery plugins. When you come up with some useful ones, consider sharing them with the rest of the jQuery community. Visit <http://jquery.com/plugins/> for more information.

7.5 Summary

This chapter introduced us to how we can write code that extends jQuery.

Writing our own code as extensions to jQuery has a number of advantages. Not only does it keep our code consistent across our web application regardless of whether it's employing jQuery APIs or our own, but it also makes all of the power of jQuery available to our own code.

Following a few naming rules helps avoid naming collisions between filenames, as well as problems that might be encountered when the `$` name is reassigned by a page that will use our plugin.

Creating new utility functions is as easy as creating new function properties on `$`, and new wrapper methods are as easily created as properties of `$.fn`.

If plugin authoring intrigues you, we highly recommend that you download and comb through the code of existing plugins to see how their authors implemented their own features. You'll see how the techniques presented in this chapter are used in a wide range of code and learn new techniques that are beyond the scope of this book.

Having yet more jQuery knowledge at our disposal, let's move on to learning how jQuery makes incorporating Ajax into our Rich Internet Applications a no-brainer.

Talk to the server with Ajax

This chapter covers

- A brief overview of Ajax
- Loading pre-formatted HTML from the server
- Making general GET and POST requests
- Making requests with fine-grained control
- Setting default Ajax properties
- A comprehensive example

It can be successfully argued that no one technology has transformed the landscape of the web more in recent years than the adoption of Ajax. The ability to make asynchronous requests back to the server without the need to reload pages has enabled a whole new set of user interaction paradigms and made Rich Internet Applications possible.

Ajax is a less recent addition to the web toolbox than many people may realize. In 1998, Microsoft introduced the ability to perform asynchronous requests under script control (discounting the use of `<iframe>` elements for such activity) as an ActiveX control to enable the creation of Outlook Web Access (OWA). Although OWA was a moderate success, few people seemed to take notice of the underlying technology.

A few years passed, and a handful of events launched Ajax into the collective consciousness of the web development community. The non-Microsoft browsers implemented a standardized version of the technology as the XMLHttpRequest (XHR) object; Google began using XHR; and, in 2005, Jesse James Garrett of Adaptive Path coined the term *Ajax* (for Asynchronous JavaScript and XML).

As if they were only waiting for the technology to be given a catchy name, the web development masses suddenly took note of Ajax in a *big* way, and it has become one of the primary tools by which we can enable Rich Internet Applications.

In this chapter, we'll take a brief tour of Ajax (if you're already an Ajax guru, you might want to skip ahead to section 8.2) and then look at how jQuery makes using Ajax a snap.

Let's start off with a refresher on what Ajax technology is all about.

8.1 *Brushing up on Ajax*

Although we'll take a quick look at Ajax in this section, please note that it's not intended as a complete Ajax tutorial or an Ajax primer. If you're completely unfamiliar with Ajax (or worse, still think that we're talking about a dishwashing liquid or a mythological Greek hero), we encourage you to familiarize yourself with the technology through resources that are geared towards teaching you *all* about Ajax; the Manning books *Ajax in Action* and *Ajax in Practice* are both excellent examples.

Some people may argue that the term Ajax applies to any means to make server requests without the need to refresh the user-facing page (such as by submitting a request to a hidden `<iframe>` element), but most people associate the term with the use of XHR or the Microsoft XMLHttpRequest ActiveX control.

Let's take a look at how those objects are used to generate requests to the server, beginning with creating one.

8.1.1 Creating an XHR instance

In a perfect world, code written for one browser would work in all commonly used browsers. We've already learned that we don't live in that world; things don't change with Ajax. There is a standard means to make asynchronous requests via the JavaScript XHR object, and an Internet Explorer proprietary means that uses an ActiveX control. With IE7, a wrapper that emulates the standard interface is available, but IE6 requires divergent code.

Once created (thankfully) the code to set up, initiate, and respond to the request is relatively browser-independent, and creating an instance of XHR is easy for any particular browser. The problem is that different browsers implement XHR in different ways, and we need to create the instance in the manner appropriate for the current browser.

But rather than relying on detecting which browser a user is running to determine which path to take, we'll use the preferred technique known as *object detection*. In this technique, we try to figure out what the browser's capabilities are, not which browser is being used. Object detection results in more robust code because it can work in any browser that supports the tested capability.

The code of listing 8.1 shows a typical idiom used to instantiate an instance of XHR using this technique.

Listing 8.1 Object detection resulting in code that can deal with many browsers

```
var xhr;
if (window.XMLHttpRequest) {
    xhr = new XMLHttpRequest();
}
else if (window.ActiveXObject) {
    xhr = new ActiveXObject("Msxml2.XMLHTTP");
}
else {
    throw new Error("Ajax is not supported by this browser");
}
```

Tests to see if XHR is defined

Tests to see if ActiveX is present

Throws error if there's no XHR support

After creation, the XHR instance sports a conveniently consistent set of properties and methods across all supporting browser instances. These properties and methods are shown in table 8.1, and the most commonly used of these will be discussed in the sections that follow.

With an instance created, let's look at what it takes to set up and fire off the request to the server.

Table 8.1 XHR methods and properties

Methods	Description
<code>abort()</code>	Causes the currently executing request to be cancelled.
<code>getAllResponseHeaders()</code>	Returns a single string containing the names and values of all response headers.
<code>getResponseHeader(name)</code>	Returns the value of the named response header.
<code>open(method,url,async,username,password)</code>	Sets the method and destination URL of the request. Optionally, the request can be declared synchronous, and a username and password can be supplied for requests requiring container-based authentication.
<code>send(content)</code>	Initiates the request with the specified (optional) body content.
<code>setRequestHeader(name,value)</code>	Sets a request header using the specified name and value.
Properties	Description
<code>onreadystatechange</code>	Assigns the event handler used when the state of the request changes.
<code>readyState</code>	An integer value that indicates the state of the request as follows: <ul style="list-style-type: none"> ■ 0—Uninitialized ■ 1—Loading ■ 2—Loaded ■ 3—Interactive ■ 4—Complete
<code>responseText</code>	The body content returned in the response.
<code>responseXML</code>	If the body content is XML, the XML DOM created from the body content.
<code>status</code>	Response status code returned from the server. For example: 200 for <i>success</i> or 404 for <i>not found</i> . See the HTTP Specification ¹ for the full set of codes.
<code>statusText</code>	The status text message returned by the response.

¹ <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10>

8.1.2 Initiating the request

Before we can send a request to the server, we need to do the following setup steps:

- 1 Specify the HTTP method such as (POST or GET)
- 2 Provide the URL of the server-side resource to be contacted
- 3 Let the XHR instance know how it can inform us of its progress
- 4 Provide any body content for POST requests

We set up the first two items by calling the `open()` method of XHR as follows:

```
xhr.open('GET', '/some/resource/url');
```

Note that this method does *not* cause the request to be sent to the server. It merely sets up the URL and HTTP method to be used. The `open()` method can also be passed a third Boolean parameter that specifies if the request is to be asynchronous (if `true`, which is the default) or synchronous (if `false`). There's seldom a good reason not to make the request asynchronous (even if it means we don't have to deal with callback functions); after all, the asynchronous nature of the request is usually the whole point of making a request in this fashion.

Third, we provide a means for the XHR instance to tap us on the shoulder to let us know what's going on; we accomplish this by assigning a callback function to the `onreadystatechange` property of the XHR object. This function, known as the *ready state handler*, is invoked by the XHR instance at various stages of its processing. By looking at the settings of the various other properties of XHR, we can find out exactly what's going on with the request. We'll take a look at how a typical ready state handler operates in the next section.

The last steps to initiating the request are to provide any body content for POST requests and send it off to the server. Both of these are accomplished via the `send()` method. For GET requests, which typically have no body, no body content parameter is passed as follows:

```
xhr.send(null);
```

When request parameters are passed to POST requests, the string passed to the `send()` method must be in the proper format (which we might think of as *query string* format) in which the names and values must be properly URI-encoded. URI encoding is beyond the scope of this section (and as it turns out, jQuery is going to handle all of that for us), but if you're curious, do a web search for the term `encodeURIComponent`, and you'll be suitably rewarded.

An example of such a call is as follows:

```
xhr.send('a=1&b=2&c=3');
```

Now let's see what the ready handler is all about.

8.1.3 Keeping track of progress

An XHR instance informs us of its progress through the *ready state handler*. This handler is established by assigning a reference to the function to serve as the ready handler to the `onreadystatechange` property of the XHR instance.

Once the request is initiated via the `send()` method, this callback will be invoked numerous times as the request makes transitions through its various states. The current state of the request is available as a numeric code in the `readyState` property (see the description of this property in table 8.1).

That's nice, but more times than not, we're only interested in when the request completes and whether it was successful or not. So frequently, we'll see ready handlers implemented using the pattern shown in listing 8.2.

Listing 8.2 Writing the ready state handler to ignore all but the completed state

```
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    if (xhr.status >= 200 &&
        xhr.status < 300) {
      //success
    }
    else {
      //error
    }
  }
}
```

← Ignores all but completed state

← Branches on response status

← Executes on success

← Executes on failure

This pattern ignores all but the completed state and, once complete, examines the value of the `status` property to determine if the request succeeded or not. The HTTP Specification defines all status codes in the 200 to 299 range as success and those with values of 300 or above as various types of failures.

We should note one thing about this ready handler; it referenced the XHR instance through a top-level variable. But shouldn't we expect the instance to be passed to the handler as a parameter?

Well, we could have *expected* that, but that's not what happens. The instance must be located by some other means, and that's usually a top-level (global)

variable. This could be a problem when we want to have more than one request firing simultaneously. Luckily, we shall see that the jQuery Ajax API handily solves this problem for us.

Let's explore how to deal with the response from a completed request.

8.1.4 Getting the response

Once the ready handler has determined that the `readyState` is complete and that the request completed successfully, the body of the response can be retrieved from the XHR instance.

Despite the moniker Ajax (where the *X* stands for XML), the format of the response body can be any text format; it's not limited to XML. In fact, most of the time, the response to Ajax requests is a format *other* than XML. It could be plain text or, perhaps, an HTML fragment; it could even be a text representation of a JavaScript object or array in JavaScript Object Notation (JSON) format.

Regardless of its format, the body of the response is available via the `responseText` property of the XHR instance (assuming that the request completes successfully). If the response indicates that the format of its body is XML by including a content-type header specifying a MIME type of `text/xml` (or any XML MIME type), the response body will be parsed as XML. The resulting DOM will be available in the `responseXML` property. JavaScript (and jQuery itself, using its selector API) can then be used to process the XML DOM.

Processing XML on the client isn't rocket science, but—even with jQuery's help—it can still be a pain. Although there are times when nothing but XML will do for returning complex hierarchical data, frequently page authors will use other formats when the full power (and corresponding headache) of XML isn't absolutely necessary.

But some of those other formats aren't without their own pain. When JSON is returned, it must be converted into its runtime equivalent. When HTML is returned, it must be loaded into the appropriate destination element. And what if the HTML markup returned contains `<script>` blocks that need evaluation? We're not going to deal with these issues in this section because it isn't meant to be a complete Ajax reference and, more importantly, because we're going to find out that jQuery handles most of these issues on our behalf.

A diagram of this whole process is shown in figure 8.1.

In this short overview of Ajax, we've identified the following pain points that page authors using Ajax need to deal with:

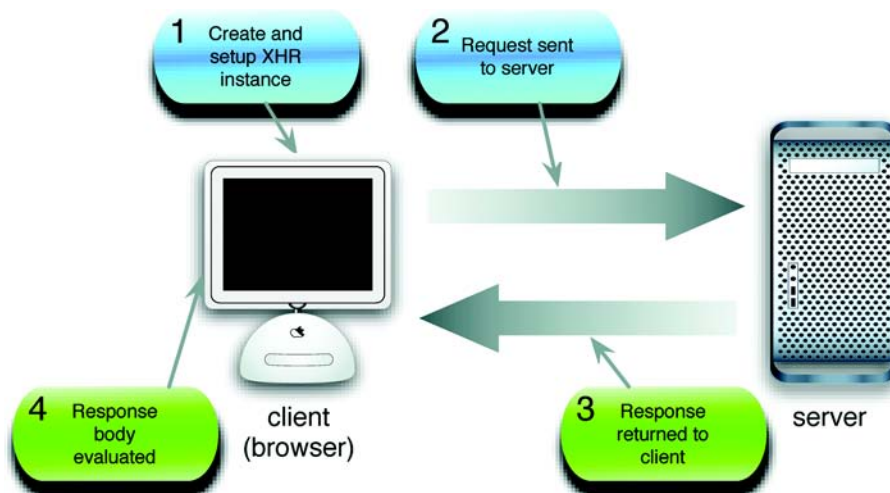


Figure 8.1 The life cycle of an Ajax request as it makes its way from the client to the server and back again

- Instantiating an XHR object requires browser-specific code.
- Ready handlers need to sift through a lot of uninteresting state changes.
- Ready handlers don't automatically get a reference to invoking XHR instances.
- The response body needs to be dealt with in numerous ways depending upon its format.

The remainder of this chapter will describe how the jQuery Ajax commands and utility functions make Ajax a lot easier (and cleaner) to use on our pages. There are a lot of choices in the jQuery Ajax API, and we'll start with some of the simplest and most often-used tools.

8.2 Loading content into elements

Perhaps one of the most common uses of Ajax is to grab a chunk of content from the server and stuff it into the DOM at some strategic location. The content could be an HTML fragment that's to become the child content of a target container element, or it could be plain text that will become the content of the target element.

Let's imagine that, on page load, we want to grab a chunk of HTML from the server using a resource named `/serverResource` and make it the content of a

Setting up for the examples

Unlike any of the example code that we've examined so far in this book, the code examples for this chapter require the services of a web server to receive the requests to server-side resources. Because it's well beyond the scope of this book to discuss the operation of server-side mechanisms, we're going to set up some minimal server-side resources that send data back to the client without worrying about doing it for real. We'll treat the server as a black box; we don't need or want to know how it's doing its job.

To enable the serving of these smoke and mirrors resources, you'll need to set up a web server of some type. For your convenience, the server-side resources have been set up in two formats: Java Server Pages (JSP) and some in PHP. The JSP resources can be used if you're running (or wish to run) a servlet/JSP engine; if you want to enable PHP for your web server of choice, you can use the PHP resources.

If you want to use the JSP resources but aren't already running a suitable server, instructions on setting up the free Tomcat web server are included with the sample code for this chapter. You'll find these instructions in the file `chapter8/tomcat.pdf`. And don't be concerned; it's easier than you might think!

The examples found in the downloaded code are set up to use the JSP resources. If you want to switch the examples to use PHP, do a search-and-replace of all instances of the string `.jsp` with `.php`. Note that not all server-side resources have been translated from JSP to PHP, but the existing PHP resources should be enough to let the PHP-savvy fill in the rest of the resources.

Once you have the server of your choice set up, you can hit the URL `http://localhost:8080/chapter8/test.jsp` (to check your Tomcat installation) or `http://localhost/chapter8/test.php` (to check your PHP installation). The latter assumes that you have set up your web server (Apache or any other you have chosen) to use the example code root folder as a document base.

When you can successfully view the appropriate test page, you'll be ready to run the examples in this chapter.

`<div>` element with an id of `someContainer`. For the final time in this chapter, let's look at how we do this without jQuery assistance. Using the patterns we set out earlier in this chapter, the body of the `onload` handler is as shown in listing 8.3. The full HTML file for this example can be found in the file `chapter8/listing.8.3.html`.

Listing 8.3 Using native XHR to include an HTML fragment

```
var xhr;

if (window.XMLHttpRequest) {
    xhr = new XMLHttpRequest();
}
else if (window.ActiveXObject) {
    xhr = new ActiveXObject("Msxml2.XMLHTTP");
}
else {
    throw new Error("Ajax is not supported by this browser");
}

xhr.onreadystatechange = function() {
    if (xhr.readyState == 4) {
        if (xhr.status >= 200 && xhr.status < 300) {
            document.getElementById('someContainer')
                .innerHTML = xhr.responseText;
        }
    }
}

xhr.open('GET', '/serverResource');
xhr.send();
```

Although there's nothing tricky going on here, that's a non-trivial amount of code; 19 lines or so—even accounting for blank lines that we added for readability and one line that we artificially broke in two so that it would fit on the page.

The equivalent code we'd write as the body of a ready handler using jQuery is as follows:

```
$('#someContainer').load('/serverResource');
```

We're betting that we know which code you'd rather write! Let's look at the jQuery command that we used in this statement.

8.2.1 Loading content with jQuery

The simple jQuery statement from the previous section easily loads content from the server-side resource using one of the most basic, but useful, jQuery Ajax commands: `load()`. The full syntax description of this command is as follows:

Command syntax: load**load(url, parameters, callback)**

Initiates an Ajax request to the specified URL with optional parameters. A callback function can be specified that's invoked when the request completes. The response text replaces the content of all matched elements.

Parameters

<code>url</code>	(String) The URL of the server-side resource to which the request is sent.
<code>parameters</code>	(Object) An object whose properties are serialized into properly encoded parameters to be passed to the request. If specified, the request is made using the POST method. If omitted, the GET method is used.
<code>callback</code>	(Function) A callback function invoked after the response data has been loaded into the elements of the matched set. The parameters passed to this function are the response text, the status code, and the XHR instance.

Returns

The wrapped set.

Though simple to use, this command has some important nuances. For example, when the `parameters` parameter is used to supply the request parameters, the request is made using the POST HTTP method; otherwise, a GET request is initiated. If we want to make a GET request with parameters, we can include them as a query string on the URL. But be aware that when we do so, we're responsible for ensuring that the query string is properly formatted and that the names and values of the request parameters are URI-encoded.

Most of the time, we'll use the `load()` command to inject the complete response into whatever elements are contained within the wrapped set, but sometimes we may want to filter elements coming back as the response. If we want to filter response elements, jQuery allows us to specify a selector on the URL that will be used to limit which response elements are injected into the wrapped elements by suffixing the URL with a space and pound sign character (`#`) followed by the selector.

For example, to filter response elements so that only `<div>` instances are injected, we write

```
$('.injectMe').load('/someResource #div');
```

If the request parameters come from form controls, a helpful command in building a query string is `serialize()`, whose syntax is as follows:

Command syntax: serialize**serialize()**

Creates a properly formatted and encoded query string from all successful form elements in the wrapped set

Parameters

none

Returns

The formatted query string

The `serialize()` command is smart enough to only collect information from form control elements in the wrapped set, and only from those qualifying elements that are deemed *successful*. A successful control is one that would be included as part of a form submission according to the rules of the HTML Specification.² Controls such as unchecked check boxes and radio buttons, dropdowns with no selections, and any disabled controls are not considered successful and do not participate in the submission. They are also ignored by `serialize()`.

If we'd rather get the form data in a JavaScript array (as opposed to a query string), jQuery provides the `serializeArray()` method.

Command syntax: serializeArray**serializeArray()**

Collects the values of all successful form controls into an array of objects containing the names and values of the controls

Parameters

none

Returns

The array of form data

The array returned by `serializeArray()` is composed of anonymous object instances, each of which contains a `name` property and a `value` property that contain the name and value of each successful form control.

With the `load()` command at our disposal, let's put it to work solving a common real-world problem that many web developers encounter.

² <http://www.w3.org/TR/html401/interact/forms.html#h-17.13.2>

8.2.2 Loading dynamic inventory data

Often in business applications, particularly for retail web sites, we want to grab real-time data from the server to present our users with the most up-to-date information. After all, we wouldn't want to mislead customers into thinking that they can buy something that's not available, would we?

In this section, we'll begin to develop a page that we'll add onto throughout the course of the chapter. This page is part of a web site for a fictitious firm named The Boot Closet, an online retailer of overstock and closeout motorcycle boots. Unlike the fixed product catalogs of other online retailers, this inventory of overstock and closeouts is fluid, depending on what deals the proprietor was able to make that day and what's already been sold from the inventory. So it will be important for us to always make sure that we're displaying the latest info!

To begin our page (which will ignore site navigation and other boilerplate to concentrate on the lesson at hand), we want to present our customers with a dropdown containing the styles that are currently available and, on a selection, display detailed information regarding that style to the customer. On initial display, the page will look as shown in figure 8.2.

After the page first loads, a pre-loaded dropdown with the list of the styles currently available in the inventory and labeled fields to show the item data when a

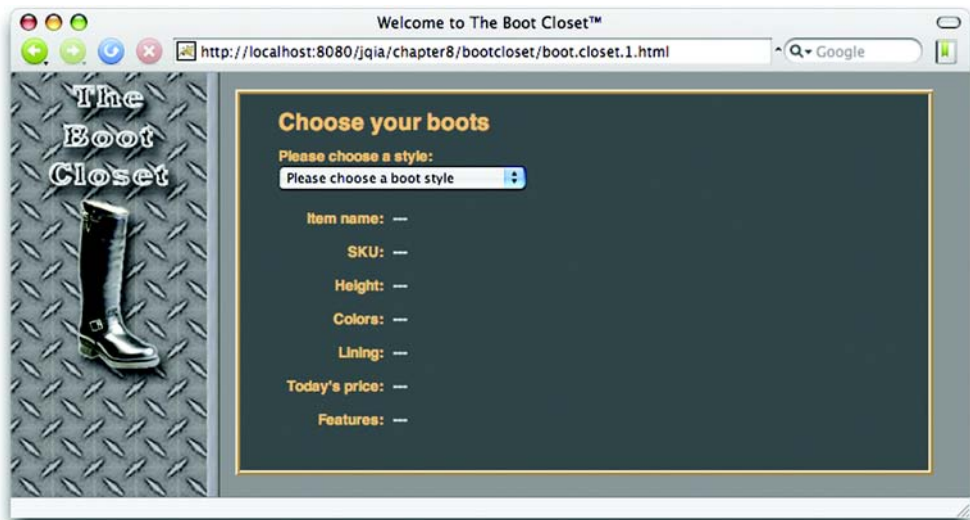


Figure 8.2 The initial display of our item information page

style is selected will be displayed. When no style is selected, we'll display dashes as a placeholder for the data.

Let's start by defining the HTML markup that will create this page structure as follows:

```
<body id="bootCloset1">
  
  <form action="" id="orderForm">
    <div id="detailFormContainer">
      <h1>Choose your boots</h1>
      <div>
        <label>Please choose a style:</label><br/>
        <select id="styleDropdown">
          <option value="">Please choose a boot style</option>
          <option value="7177382">Caterpillar Tradesman Work Boot</option>
          <option value="7269643">Caterpillar Logger Boot</option>
          <option value="7141832">Chippewa 17" Engineer Boot</option>
          <option value="7141833">Chippewa 17" Snakeproof Boot</option>
          <option value="7173656">Chippewa 11" Engineer Boot</option>
          <option value="7141922">Chippewa Harness Boot</option>
          <option value="7141730">Danner Foreman Pro Work Boot</option>
          <option value="7257914">Danner Grouse GTX Boot</option>
        </select>
      </div>
      <div id="detailsDisplay"></div>
    </div>
  </form>
</body>
```

Not much to it, is there?

We've defined all the visual rendition information in an external stylesheet, and we've included no behavioral aspects in the HTML markup in order to adhere to the precepts of Unobtrusive JavaScript.

The options for the styles dropdown have been pre-populated. In all the examples in this chapter, we assume that we're using server-side resources to power our web application; communicating with these resources is, after all, the whole point of Ajax. So even though the example uses a simple HTML file, we assume that it was originally generated by some server-side templating resources such as a JSP or PHP page and that the product data was dynamically included from the inventory database (or wherever it's stored).

Also, the `<div>` container defined (with an `id` of `detailsDisplay`) to hold the details display is completely empty! We're going to rely on the server-side templating resource to provide the dynamic content, so we don't want to specify it

here *and* in the JSP (or PHP) page; having the structure defined in two places would necessitate keeping them in sync. Bad idea!

On page load, we grab the empty version of the content from the server so that the structure only needs to be defined in one place. Let's look at our ready handler now.

```
$(function(){
  $('#styleDropdown')
    .change(function(){
      var styleValue = $(this).val();
      $('#detailsDisplay').load(
        'getDetails.jsp',
        { style: styleValue }
      );
    })
    .change();
});
```

① Wraps style dropdown and binds change handler

② Loads data for selected style

③ Triggers change handler

In this ready handler, we wrap the boot style dropdown and bind a change handler ① to it. In the callback for the change handler, which will be invoked whenever a customer changes the selection, we obtain the current value of the selection by applying the `val()` command to `this`, which in the handler is the `<select>` element that triggered the event. We then apply the `load()` command ② to the `detailsDisplay` element to initiate an Ajax callback to a server-side resource, `getDetails.jsp`, passing the style value as a parameter named `style`.

As the final act of the ready handler, we call the `change()` command ③ to invoke the change handler. This issues a request with the default style selection of "" (the empty string), causing the server-side resource to return the construct that results in the display that was shown in figure 8.2.

After the customer chooses an available boot style, the page will appear as shown in figure 8.3.

The most notable operation performed in the ready handler is the use of the `load()` command to quickly and easily fetch a snippet of HTML from the server and place it within the DOM as the child of an existing element. This command is extremely handy and well suited to web applications powered by servers capable of server-side templating such as JSP and PHP.

Listing 8.4 shows the complete code for our Boot Closet page, which can be found in the file `chapter8/bootcloset/boot.closet.1.html`. We'll be revisiting this page to add further capabilities to it as we progress through this chapter.

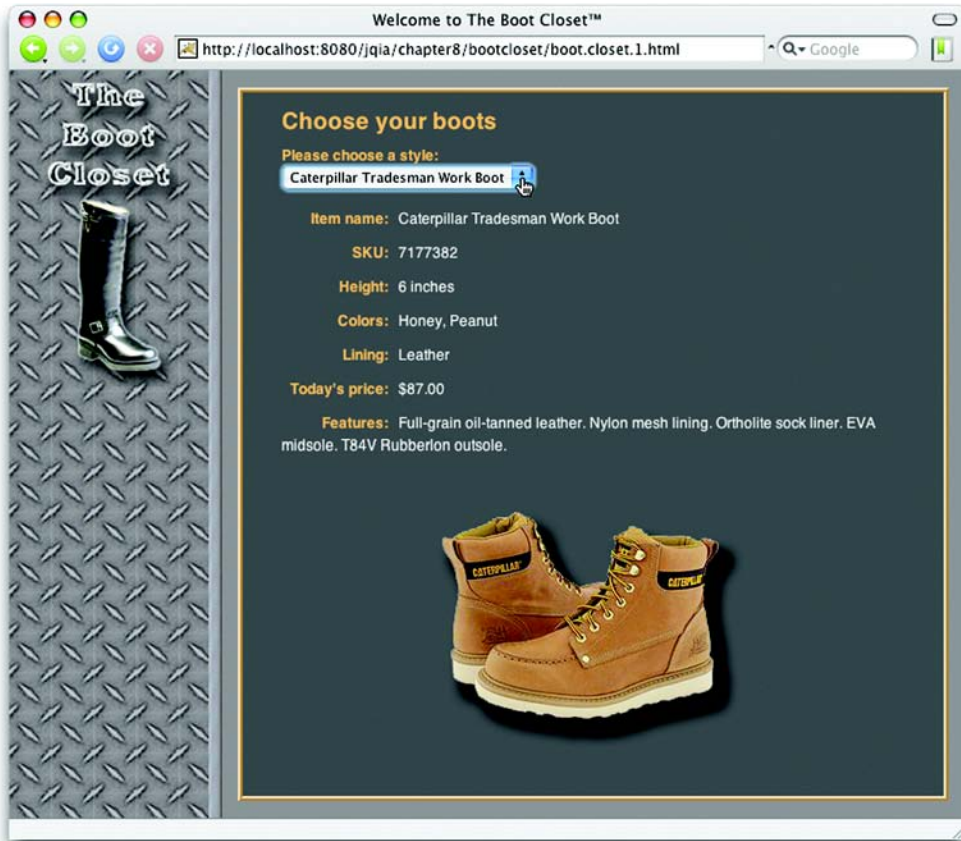


Figure 8.3 The server-side resource returns a pre-formatted fragment of HTML to display the boot information.

Listing 8.4 The first phase of our Boot Closet retailer page

```
<html>
<head>
  <title>Welcome to The Boot Closet™</title>
  <link rel="stylesheet" type="text/css" href="boot.closet.css">
  <script type="text/javascript"
    src="../../scripts/jquery-1.2.1.js"></script>
  <script type="text/javascript">
    $(function(){
      $('#styleDropdown')
        .change(function(){
          var styleValue = $(this).val();
          $('#detailsDisplay').load(
            'getDetails.jsp',

```

```

        { style: styleValue }
      );
    })
    .change();
  });
</script>
</head>

<body id="bootCloset1">
  
  <form action="" id="orderForm">
    <div id="detailFormContainer">
      <h1>Choose your boots</h1>
      <div>
        <label>Please choose a style:</label><br/>
        <select id="styleDropdown">
          <option value="">Please choose a boot style</option>
          <option value="7177382">
            Caterpillar Tradesman Work Boot</option>
          <option value="7269643">Caterpillar Logger Boot</option>
          <option value="7141832">Chippewa 17" Engineer Boot</option>
          <option value="7141833">Chippewa 17" Snakeproof Boot</option>
          <option value="7173656">Chippewa 11" Engineer Boot</option>
          <option value="7141922">Chippewa Harness Boot</option>
          <option value="7141730">Danner Foreman Pro Work Boot</option>
          <option value="7257914">Danner Grouse GTX Boot</option>
        </select>
      </div>
      <div id="detailsDisplay"></div>
    </div>
  </form>
</body>
</html>

```

The `load()` command is tremendously useful when we want to grab a fragment of HTML to stuff into the content of an element (or set of elements). But there may be times when we either want more control over how the Ajax request gets made, or we need to do something more esoteric with the returned data in the response body.

Let's continue our investigation of what jQuery has to offer for these more complex situations.

8.3 Making GET and POST requests

The `load()` command makes either a GET or a POST request, depending on whether it's called with request data, but sometimes we want to have a bit more

control over which HTTP method gets used. Why should we care? Because, maybe, our *servers* care.

Web authors have traditionally played fast and loose with the GET and POST methods, using one or the other without heeding how the HTTP protocol intends for these methods to be used. The intentions for each method are as follows:

- *GET requests*—Intended to be *idempotent*; the state of the server and the model data for the application should be unaffected by a GET operation. The same GET operation, made again and again and again, should return exactly the same results (assuming no other force is at work changing the server state).
- *POST requests*—Can be *non-idempotent*; the data they send to the server can be used to change the model state of the application; for example, adding records to a database or removing information from the server.

A GET request should, therefore, be used for getting data (as its name implies). It may be required to *send* some data to the server for the GET; for example, to identify a style number to retrieve color information. But when data is being sent to the server in order to effect a change, POST should be used.

WARNING This is more than theoretical. Browsers make decisions about caching based upon the HTTP method used; GET requests are highly subject to caching. Using the proper HTTP method ensures that you don't get cross-ways with the browser's expectations regarding the intentions of the requests.

All that being said, jQuery gives us a few means to make GET requests, which unlike `load()`, aren't implemented as jQuery commands for a wrapped set. *Utility functions* are provided to make various types of GET requests. As we pointed out in chapter 1, jQuery utility functions are top-level functions that are namespaced with the `jQuery` global name or its `$` alias.

Let's look at each of these functions.

8.3.1 Getting data with jQuery

When we want to fetch some data from the server and decide what to do with it ourselves (rather than letting the `load()` command set it as the content of an HTML element), we can use the `$.get()` utility function. Its syntax is as follows:

Command syntax: \$.get**\$.get(url, parameters, callback)**

Initiates a GET request to the server using the specified URL with any passed parameters as the query string.

Parameters

<code>url</code>	(String) The URL of the server-side resource to contact via the GET method.
<code>parameters</code>	(Object String) An object whose properties serve as the name/value pairs used to construct a query string to be appended to the URL, or a preformatted and encoded query string.
<code>callback</code>	(Function) A function invoked when the request completes. The response body is passed as the first parameter to this callback, and the status as the second.

Returns

The XHR instance.

Let's look at a simple use of this function as shown in listing 8.5 (which can be found in the file `chapter8/$.get.html`).

Listing 8.5 Using \$.get() utility function to fetch data from the server

```
<html>
  <head>
    <title>$.get() Example</title>
    <link rel="stylesheet" type="text/css" href="../common.css">
    <script type="text/javascript"
      src="../scripts/jquery-1.2.1.js"></script>
    <script type="text/javascript">
      $(function(){
        $('#testButton').click(function(){
          $.get(
            'reflectData.jsp',
            {a:1, b:2, c:3},
            function(data) { alert(data); }
          );
        });
      });
    </script>
  </head>

  <body>
    <button type="button" id="testButton">Click me!</button>
  </body>
</html>
```

1 Gets data from the server

In this simple page, we create a button and instrument it to initiate a call to `$.get()` ❶ once it's clicked. The GET request is made to the server resource at `reflectData.jsp` (which returns a text snippet showing the values that were passed to it as request parameters), specifying values for request parameters `a`, `b`, and `c`. The callback is passed the fetched data and can do whatever it wants with it. In this case, it merely issues an alert displaying that data.

When this HTML page is loaded into a browser and the button is clicked, we see the display of figure 8.4.

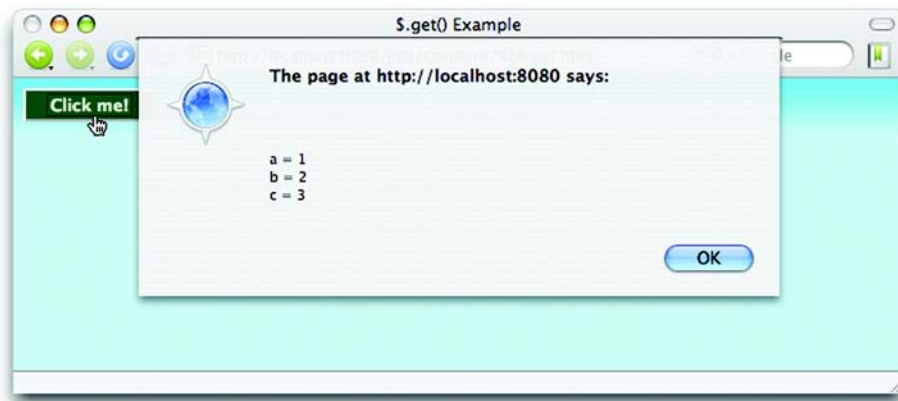


Figure 8.4 The `$.get()` utility function fetches data from the server that we can manipulate as we please, including only showing it in an alert.

If the response contains an XML document, the document will be parsed, and the data parameter passed to the callback will be the resulting DOM.

XML is great when we need its flexibility and our data is hierarchical in nature, but it can be painful to digest. Let's see another jQuery utility function that's quite useful when our data needs are more basic.

8.3.2 Getting JSON data

As stated in the previous section, when an XML document is returned from the server, the XML document is automatically parsed, and the resulting DOM is made available to the callback function. When XML is overkill or otherwise unsuitable as a data transfer mechanism, JSON is often used in its place; one reason is that JSON is easy to digest in client-side script. Well, jQuery makes it even easier.

For times when we know that the response will be JSON, the `$.getJSON()` utility function automatically parses the returned JSON string and makes the resulting

JavaScript data item available to its callback. The syntax of this utility function is as follows:

Command syntax: \$.getJSON

`$.getJSON(url, parameters, callback)`

Initiates a GET request to the server using the specified URL with any passed parameters as the query string. The response is interpreted as a JSON string, and the resulting data is passed to the callback function.

Parameters

<code>url</code>	(String) The URL of the server-side resource contacted via the GET method.
<code>parameters</code>	(Object String) An object whose properties serve as the name/value pairs used to construct a query string to be appended to the URL, or a preformatted and encoded query string.
<code>callback</code>	(Function) A function invoked when the request completes. The data value resulting from digesting the response body as a JSON string is passed as the first parameter to this callback, and the status as the second.

Returns

The XHR instance.

This function is great for those times when we want to get data from the server without the overhead of dealing with XML. Let's see an example that puts it to work for us.

Loading cascading dropdowns

When creating Rich Internet Applications, we often encounter the need to set the options in a dropdown control to values that are dependent on the setting of some other control, frequently another dropdown. A common example is choosing a state or province from one dropdown that causes a subsequent dropdown to be loaded with the list of cities in that state or province.

Such a set of controls has come to be known by the term *dependent dropdowns*—or, sometimes, *cascading dropdowns*—and has become a poster child for Ajax; it's used as an example in almost every Ajax book in existence, as well as all over the Internet. In this section, we'll look at how to solve this canonical problem and create a set of elements by leveraging the jQuery `$.getJSON()` utility function.

For this example, we'll return to The Boot Closet page that we set up in section 8.2.2 and extend its capabilities. As originally written, the page allows our customers to find out which boots are available and the detailed information associated with the available boots, but they have no way to pick a pair for

purchase. We like purchases, so our next step is to add controls that let them pick color and size.

Remember that we're a closeout business—that means we don't always have a full line of any one style available. Only certain colors are available from day to day, and only certain sizes are available in those colors. So we can't hard-code lists of colors and sizes; we need to obtain these lists dynamically from our real-time inventory database.

To allow the customer to select colors and sizes, we'll add two dropdowns to our form: one for color and one for size. The initial appearance of our enhanced form is shown in figure 8.5, and you can load the HTML page for this example from the file `chapter8/bootcloset/boot.closet.2.html`.

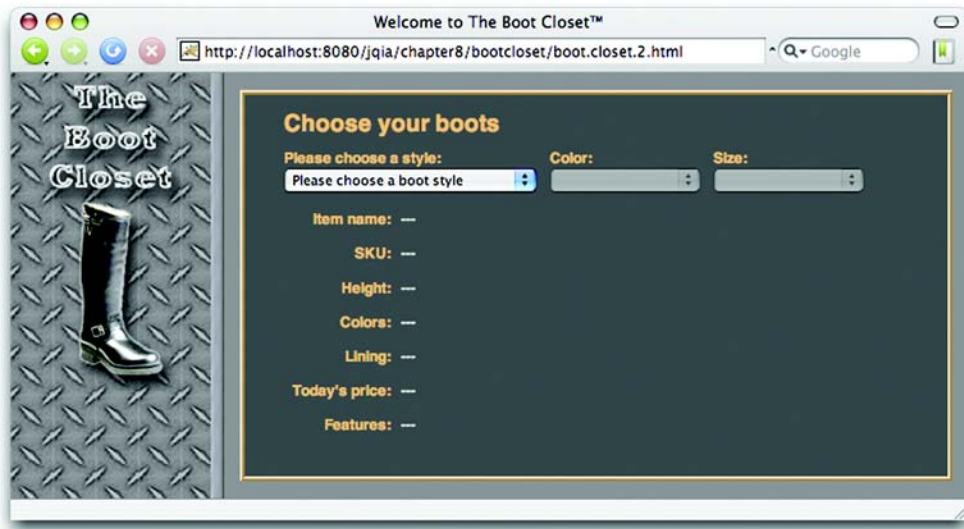


Figure 8.5 The initial state of the order form with the dependent dropdowns in an empty and disabled state

The dropdown element for choosing the boot style is enabled (and pre-filled with the available styles as we discussed earlier), but the color and size controls are disabled and empty. We can't pre-fill these dropdowns because we don't know what colors to display until a style is picked, and we don't know what sizes to display until we know both the style and the color.

Given that, here are the things that we wish to accomplish for these controls:

- When a style is selected, the color dropdown should be enabled and filled with the colors available for the selected style.

- When both a style and a color are selected, the size dropdown should be enabled and display the sizes available for the combination of style and color.
- The dropdowns always need to be consistent with each other. We need to ensure that, regardless of the order in which the user goes about manipulating the controls, a combination that's not available in inventory never be shown.

Let's roll up our sleeves and get to work.

To begin, let's lay out the additional HTML markup for the page body. Here's the HTML for the new page with significant changes and additions to the code highlighted in bold:

```
<body id="bootCloset2">
  
  <form action="" id="orderForm">
    <div id="detailFormContainer">
      <h1>Choose your boots</h1>
      <div id="cascadingDropdowns">
        <div>
          <label>Please choose a style:</label><br/>
          <select id="styleDropdown">
            <option value="">Please choose a boot style</option>
            <option value="7177382">
              Caterpillar Tradesman Work Boot</option>
            <option value="7269643">Caterpillar Logger Boot</option>
            <option value="7141832">Chippewa 17" Engineer Boot</option>
            <option value="7141833">Chippewa 17" Snakeproof Boot</option>
            <option value="7173656">Chippewa 11" Engineer Boot</option>
            <option value="7141922">Chippewa Harness Boot</option>
            <option value="7141730">Danner Foreman Pro Work Boot</option>
            <option value="7257914">Danner Grouse GTX Boot</option>
          </select>
        </div>
        <div>
          <label>Color:</label><br/>
          <select id="colorDropdown" disabled="disabled"></select>
        </div>
        <div>
          <label>Size:</label><br/>
          <select id="sizeDropdown" disabled="disabled"></select>
        </div>
      </div>
      <div id="detailsDisplay"></div>
    </div>
  </form>
</body>
```

We've changed the id of the <body> tag (primarily so that we can use it as a switch within the CSS stylesheet used by multiple versions of the page) and added the two empty and disabled dropdowns.

To add the new behaviors associated with these controls to the page, we also need to enhance our ready handler. The new ready handler, once again with changes highlighted in bold, is as follows:

```
$(function(){
  $('#styleDropdown')
    .change(function(){
      var styleValue = $(this).val();
      $('#detailsDisplay').load(
        'getDetails.jsp',
        { style: styleValue }
      );
      adjustColorDropdown();
    })
    .change();
  $('#colorDropdown')
    .change(adjustSizeDropdown);
});
```

1 Triggers state adjustment of color dropdown

2 Binds change listener to color dropdown

These changes are minor but significant. First, we make a call to a function named `adjustColorDropdown()` within the change listener for the styles dropdown **1**. This will trigger any change required for the state and content based upon the value of the style selected.

We then add a listener to the new colors dropdown **2** to adjust the state and content of the size dropdown when the color dropdown changes value. This listener is set to be a function named `adjustSizeDropdown()`.

That's all simple enough, but we still haven't written the functions that effect the changes to the state and content of the dependent dropdowns. Let's tackle the color dropdown first with the definition of the `adjustColorDropdown()` function.

```
function adjustColorDropdown() {
  var styleValue = $('#styleDropdown').val();
  var dropdownSet = $('#colorDropdown');
  if (styleValue.length == 0) {
    dropdownSet.attr("disabled", true);
    dropdownSet.emptySelect();
    adjustSizeDropdown();
  }
  else {
    dropdownSet.attr("disabled", false);
    $.getJSON(
      'getColor.jsp',
      {style:styleValue},

```

1 Enables or disables the color dropdown

2 Empties disabled dropdown and clears dependent dropdown

3 Gets color values based on style

```

function(data){
  $(dropdownSet).loadSelect(data);
  adjustSizeDropdown(); ← 4 Triggers adjustment of
                          dependent dropdown
}
);
}
}

```

After obtaining the value of the style dropdown and tucking it away in variable `styleValue` for later reference, we form a wrapped set consisting of the color dropdown and store it in variable `dropdownSet`. We're going to reference this set repeatedly throughout the remainder of the function, and we don't want to incur the overhead of re-creating it every time we need it.

The decision whether the color dropdown should be enabled or disabled is made next ❶, depending on the value of the style dropdown—disabled if the value is empty, enabled if otherwise. If the color dropdown is disabled, it's also emptied through use of the `emptySelect()` command ❷.

Wait a minute! What `emptySelect()` command?

Before you start feverishly thumbing through the previous chapters searching for this command, don't bother; it doesn't exist—at least not yet. This command is one that we'll create ourselves in the next section. For now, be aware that this command will cause all the options in the color dropdown to be removed.

After we disable and empty the color dropdown, we call the `adjustSizeDropdown()` function ❸ to make sure that any appropriate adjustments are made to its dependent size dropdown.

If the color dropdown is enabled, it needs to be filled with the values that are appropriate for the value that is selected from the styles dropdown. To obtain those values, we use the services of the `$.getJSON()` function, ❹ specifying a server-side resource of `getColors.jsp` and passing it the selected style value via a request parameter named `style`.

When the callback function is invoked (after the Ajax request returns its response), the parameter passed to the callback is the JavaScript value resulting from the evaluation of the response as a JSON construct. A typical JSON construct returned from `getColors.jsp` is as follows:

```

[
  {value:'',caption:'choose color'},
  {value:'bk',caption:'Black Oil-tanned'},
  {value:'br',caption:'Black Polishable'}
]

```

This notation defines an array of objects, each with two properties: value and caption. These objects define the value and display text of the options to be added to the color dropdown. We populate the dropdown by passing the evaluated JSON value to the `loadSelect()` command applied to the dropdown. Yes, `loadSelect()` is another custom command that we'll define ourselves.

Finally, whenever the value of the color dropdown changes, we need to make sure that the size dropdown reflects the new value; we call the `adjustSizeDropdown()` function ④ to apply a series of operations to the size dropdown similar to the routine for the color dropdown.

The definition of that `adjustSizeDropdown()` function is as follows:

```
function adjustSizeDropdown() {
  var styleValue = $('#styleDropdown').val();
  var colorValue = $('#colorDropdown').val();
  var dropdownSet = $('#sizeDropdown');
  if ((styleValue.length == 0) || (colorValue.length == 0) ) {
    dropdownSet.attr("disabled",true);
    dropdownSet.emptySelect();
  }else {
    dropdownSet.attr("disabled",false);
    $.getJSON(
      'getSizes.jsp',
      {style:styleValue,color:colorValue},
      function(data){dropdownSet.loadSelect(data)}
    );
  }
}
```

It should be no surprise that the structure of this function is strikingly similar to that of the `adjustColorDropdown()` function. Aside from operating upon the size dropdown rather than the color dropdown, this function looks at *both* the values of the style and color dropdowns to decide whether the size dropdown is to be enabled or disabled. In this case, both the style and color values must not be empty for the size dropdown to become enabled.

If enabled, the size dropdown is loaded—once again using `$.getJSON()`—from a server-side resource named `getSizes.jsp`, which is passed both the style and color values.

With these functions in place, our cascading dropdowns are now operational. Figure 8.5 showed us what the page looks like after its initial display, and figure 8.6 shows the relevant portion of the page after a style has been selected (top part of figure) and after a color has been selected (bottom part of figure).

The full code for this page is shown in listing 8.6 and can be found in the file `chapter8/bootcloset/boot.closet.2.html`.

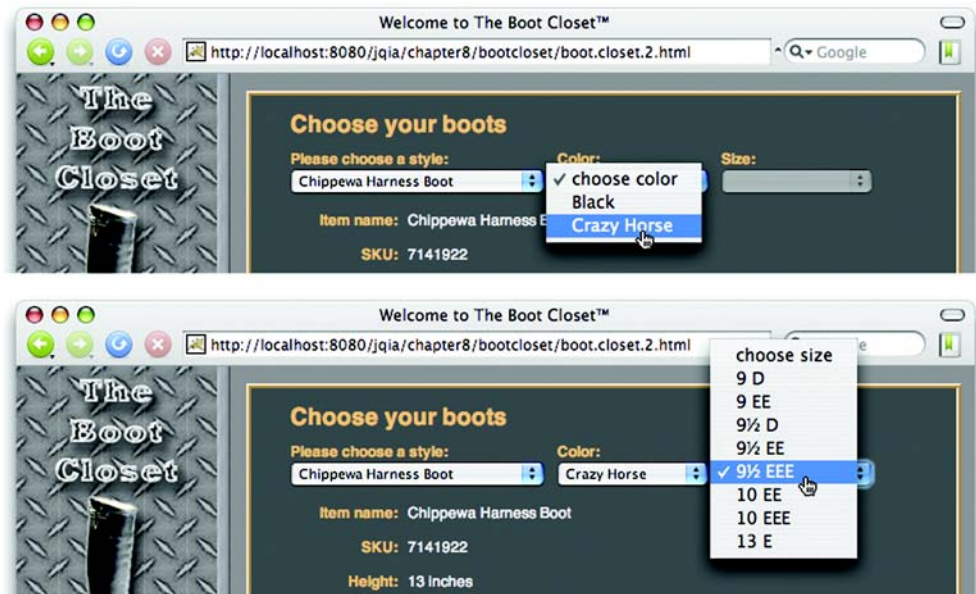


Figure 8.6 Selecting a style enables the color dropdown (top), and selecting a color enables the size dropdown (bottom).

Listing 8.6 The Boot Closet page augmented with cascading dropdowns

```

<html>
<head>
  <title>Welcome to The Boot Closet™</title>
  <link rel="stylesheet" type="text/css" href="boot.closet.css">
  <script type="text/javascript"
    src="../../scripts/jquery-1.2.1.js"></script>
  <script type="text/javascript"
    src="jquery.jqia.selects.js"></script>
  <script type="text/javascript">
    $(function(){
      $('#styleDropdown')
        .change(function(){
          var styleValue = $(this).val();
          $('#detailsDisplay').load(
            'getDetails.jsp',
            { style: styleValue }
          );
          adjustColorDropdown();
        })
        .change();
      $('#colorDropdown')

```



```

        .change(adjustSizeDropdown);
    });

    function adjustColorDropdown() {
        var styleValue = $('#styleDropdown').val();
        var dropdownSet = $('#colorDropdown');
        if (styleValue.length == 0) {
            dropdownSet.attr("disabled",true);
            dropdownSet.emptySelect();
            adjustSizeDropdown();
        } else {
            dropdownSet.attr("disabled",false);
            $.getJSON(
                'getColors.jsp',
                {style:styleValue},
                function(data){
                    dropdownSet.loadSelect(data);
                    adjustSizeDropdown();
                }
            );
        }
    }

    function adjustSizeDropdown() {
        var styleValue = $('#styleDropdown').val();
        var colorValue = $('#colorDropdown').val();
        var dropdownSet = $('#sizeDropdown');
        if ((styleValue.length == 0)|| (colorValue.length == 0) ) {
            dropdownSet.attr("disabled",true);
            dropdownSet.emptySelect();
        }else {
            dropdownSet.attr("disabled",false);
            $.getJSON(
                'getSizes.jsp',
                {style:styleValue,color:colorValue},
                function(data){dropdownSet.loadSelect(data)}
            );
        }
    }
</script>
</head>

<body id="bootCloset2">

<form action="" id="orderForm">
<div id="detailFormContainer">
<h1>Choose your boots</h1>
<div id="cascadingDropdowns">
<div>
<label>Please choose a style:</label><br/>
<select id="styleDropdown">

```

```
<option value="">Please choose a boot style</option>
<option value="7177382">
  Caterpillar Tradesman Work Boot</option>
<option value="7269643">Caterpillar Logger Boot</option>
<option value="7141832">Chippewa 17" Engineer Boot</option>
<option value="7141833">Chippewa 17" Snakeproof Boot</option>
<option value="7173656">Chippewa 11" Engineer Boot</option>
<option value="7141922">Chippewa Harness Boot</option>
<option value="7141730">Danner Foreman Pro Work Boot</option>
<option value="7257914">Danner Grouse GTX Boot</option>
</select>
</div>
<div>
  <label>Color:</label><br/>
  <select id="colorDropdown" disabled="disabled"></select>
</div>
<div>
  <label>Size:</label><br/>
  <select id="sizeDropdown" disabled="disabled"></select>
</div>
</div>
<div id="detailsDisplay"></div>
</div>
</form>
</body>
</html>
```

Before we pat ourselves on the back too hard, we should know that we're not done yet. We used custom commands in our functions that we haven't written yet! Let's get back to work...

Writing the custom commands

Our cascading dropdowns example needed to perform two operations on the select elements (dropdowns) on our page: removing all options from a dropdown and loading it with options defined within a JavaScript data construct.

We made the decision to implement these operations as jQuery commands for two major reasons:

- We knew that we'd need to perform these operations in multiple places throughout our page, so we should, at minimum, separate these operations out into individual functions rather than repeat them in inline code.
- These operations are general enough to be useful elsewhere on the site and even in other web applications. Defining and structuring them as jQuery commands makes a lot of sense.

Let's tackle the `emptySelect()` command first.

```
$.fn.emptySelect = function() {
  return this.each(function(){
    if (this.tagName=='SELECT') this.options.length = 0;
  });
}
```

We learned how to add new jQuery commands in chapter 7, and we apply those techniques here to augment `$.fn` with a new function named `emptySelect`. Remember that, when such a function is invoked, the function context (`this`) is the matched set. By applying `each()` to the matched set, we iterate through all the elements in the set, calling the iterator function specified as the parameter to `each()`.

Within *that* function, the function context is the individual element for the current round of the iteration. We check this element to ensure that it's a `<select>` element, ignoring any other element type, and set the length of the `options` array for the element to 0. This is a supported, cross-browser way to remove all options from the dropdown.

Note that we return the wrapped set being operated on as the value of the function, ensuring that this command can participate in any jQuery command chain.

Easy enough! Now let's tackle the `loadSelect()` command.

We add the following function to the `$.fn` namespace:

```
$.fn.loadSelect = function(optionsDataArray) {
  return this.emptySelect().each(function(){
    if (this.tagName=='SELECT') {
      var selectElement = this;
      $.each(optionsDataArray, function(index, optionData){
        var option = new Option(optionData.caption,
                               optionData.value);

        if ($.browser.msie) {
          selectElement.add(option);
        }
        else {
          selectElement.add(option, null);
        }
      });
    }
  });
}
```

This command is slightly more involved.

As the lone parameter to this command, we expect a JavaScript construct as defined in the previous section—an array of objects, each of which possesses a `value` and a `caption` property that define the options to be added to the `<select>` element.

We'll, once again, iterate through all the elements in the matched set; before we do that, we empty all `<select>` elements in the matched set by calling the `emptySelect()` command that we defined. This removes any options that might be in the element prior to our adding the new options.

Within the iterator function, we check the tag name of the elements and ignore all but `<select>` elements. For the elements that survive this test, we iterate through the data array passed to the command, creating a new `Option` instance for each array item and adding it to the `<select>`.

This addition is problematic because it must be performed in a browser-specific fashion. The W3C standard defines the `add()` method so that, in order to add an option to the end of the `<select>`, a `null` must be passed as the second parameter to `add()`. We'd think that this would be easily accomplished by omitting the second parameter as in

```
selectElement.add(option);
```

But life can't be that simple. The Safari and Opera browsers work equally well whether the option is omitted or explicit, but Mozilla-based browsers throw a `too-few-arguments` exception when the second parameter is omitted. Adding an explicit `null` as the second parameter doesn't help matters because doing so causes Internet Explorer to no longer add the new option.

Catch 22!

Because there is no single cross-browser way to perform the operation and no object to perform detection upon to use the preferred technique of *object detection*, we're forced to resort to browser detection. Sigh. At least we have the `$.browser` utility flags to make the detection easy.

Once again, note that the wrapped set is returned as the function's result.

The full implementation of these commands can be found in the file `chapter8/bootcloset/jquery.jqia.selects.js` and is shown in listing 8.7.

Listing 8.7 The implementation of our custom select commands

```
(function($) {  
  $.fn.emptySelect = function() {  
    return this.each(function(){  
      if (this.tagName=='SELECT') this.options.length = 0;  
    });  
  }  
  
  $.fn.loadSelect = function(optionsDataArray) {  
    return this.emptySelect().each(function(){  
      if (this.tagName=='SELECT') {
```

```
var selectElement = this;
$.each(optionsdataArray, function(index, optionData) {
    var option = new Option(optionData.caption,
                            optionData.value);

    if ($.browser.msie) {
        selectElement.add(option);
    }
    else {
        selectElement.add(option, null);
    }
});
});
})(jQuery);
```

Between `$.get()` and `$.getJSON()`, jQuery gives us some powerful tools when it comes to making GET requests, but man does not live by GETs alone!

8.3.3 Making POST requests

“Sometimes you feel like a nut, sometimes you don’t.” What’s true of choosing between an Almond Joy or a Mounds candy bar is also true of making requests to the server. Sometimes we want to make a GET, but at other times we want (or need) to make a POST request.

There are any number of reasons why we might choose a POST over a GET. First, the intention of the HTTP protocol is that POST will be used for any non-idempotent requests. Therefore, if our request has the potential to cause a change in the server-side state, it should be a POST (at least according to HTTP purists). Accepted practices and conventions aside, a POST operation must sometimes be used when the data to be passed to the server exceeds the small amount that can be passed by URL in a query string; that limit is a browser-dependent value. And sometimes, the server-side resource we contact may only support POST operations, or it might even perform different functions depending upon whether our request uses the GET or POST method.

For those occasions when a POST is desired or mandated, jQuery offers the `$.post()` utility function, which operates in the exact same fashion as `$.get()` except for the HTTP method used. Its syntax is as follows:

Command syntax: \$.post**\$.post(url, parameters, callback)**

Initiates a POST request to the server using the specified URL with any parameters passed within the body of the request.

Parameters

<code>url</code>	(String) The URL of the server-side resource to contact via the POST method.
<code>parameters</code>	(Object String) An object whose properties serve as the name/value pairs used to construct the body of the request, or a preformatted and encoded query string.
<code>callback</code>	(Function) A function invoked when the request completes. The response body is passed as the single parameter to this callback, and the status as the second.

Returns

The XHR instance.

Between the `load()` command and the various GET and POST jQuery Ajax functions, we can exert some measure of control over how our request is initiated and how we're notified of its completion. But for those times when we need *full* control over an Ajax request, jQuery has a means for us to get as picky as we want.

8.4 Taking full control of an Ajax request

The functions and commands that we've seen so far are convenient for many cases, but there may be times when we want to take control of the nitty-gritty details into our own hands.

In this section, we'll explore how jQuery lets us exert such dominion.

8.4.1 Making Ajax requests with all the trimmings

For those times when we want or need to exert a fine-grained level of control over how we make Ajax requests, jQuery provides a general utility function for making Ajax requests named `$.ajax()`. Under the covers, all other jQuery features that make Ajax requests eventually use this function to initiate the request. Its syntax is as follows:

Command syntax: \$.ajax

\$.ajax(options)

Initiates an Ajax request using the passed options to control how the request is made and callbacks notified.

Parameters

`options` (Object) An object instance whose properties define the parameters to this operation. See table 8.2 for details.

Returns

The XHR instance.

Looks simple, doesn't it? But don't be deceived. The `options` parameter can specify a large range of values that can be used to tune the operation of this function. These options (in order of the likelihood of their use) are defined in table 8.2.

Table 8.2 Options for the `$.ajax()` utility function

Name	Type	Description
<code>url</code>	String	The URL for the request.
<code>type</code>	String	The HTTP method to use. Usually either POST or GET. If omitted, the default is GET.
<code>data</code>	Object	An object whose properties serve as the query parameters to be passed to the request. If the request is a GET, this data is passed as the query string. If a POST, the data is passed as the request body. In either case, the encoding of the values is handled by the <code>\$.ajax()</code> utility function.
<code>dataType</code>	String	A keyword that identifies the type of data that's expected to be returned by the response. This value determines what, if any, post-processing occurs upon the data before being passed to callback functions. The valid values are as follows: <ul style="list-style-type: none"> ■ <code>xml</code>—The response text is parsed as an XML document and the resulting XML DOM is passed to the callbacks. ■ <code>html</code>—The response text is passed unprocessed to the callbacks functions. Any <code><script></code> blocks within the returned HTML fragment are evaluated. ■ <code>json</code>—The response text is evaluated as a JSON string, and the resulting object is passed to the callbacks. ■ <code>jsonp</code>—Similar to <code>json</code> except that remote scripting is allowed, assuming the remote server supports it. ■ <code>script</code>—The response text is passed to the callbacks. Prior to any callbacks being invoked, the response is processed as a JavaScript statement or statements. ■ <code>text</code>—The response text is assumed to be plain text.

continued on next page

Table 8.2 Options for the `$.ajax()` utility function (continued)

Name	Type	Description
		The server resource is responsible for setting the appropriate content-type response header. If this property is omitted, the response text is passed to the callbacks without any processing or evaluation.
<code>timeout</code>	Number	Sets a timeout for the Ajax request in milliseconds. If the request does not complete before the timeout expires, the request is aborted and the error callback (if defined) is called.
<code>global</code>	Boolean	Enables (if <code>true</code>) or disables (if <code>false</code>) the triggering of so-called global functions. These are functions that can be attached to elements that trigger at various points or conditions during an Ajax call. We'll be discussing them in detail in section 8.8. If omitted, the default is to enable the triggering of global functions.
<code>contentType</code>	String	The content type to be specified on the request. If omitted, the default is <code>application/x-www-form-urlencoded</code> , the same type used as the default for form submissions.
<code>success</code>	Function	A function invoked if the response to the request indicates a success status code. The response body is returned as the first parameter to this function and formatted according to the specification of the <code>dataType</code> property. The second parameter is a string containing a status value—in this case, always <code>success</code> .
<code>error</code>	Function	A function invoked if the response to the request returns an error status code. Three arguments are passed to this function: the XHR instance, a status message string (in this case, always <code>error</code>), and an optional exception object returned from the XHR instance.
<code>complete</code>	Function	A function called upon completion of the request. Two arguments are passed: the XHR instance and a status message string of either <code>success</code> or <code>error</code> . If either a success or error callback is also specified, this function is invoked after the callback is called.
<code>beforeSend</code>	Function	A function invoked prior to initiating the request. This function is passed the XHR instance and can be used to set custom headers or to perform other pre-request operations.
<code>async</code>	Boolean	If specified as <code>false</code> , the request is submitted as a synchronous request. By default, the request is asynchronous.
<code>processData</code>	Boolean	If set to <code>false</code> , prevents the data passed from being processed into URL-encoded format. By default, the data is URL-encoded into a format suitable for use with requests of type <code>application/x-www-form-urlencoded</code> .
<code>ifModified</code>	Boolean	If <code>true</code> , allows a request to succeed only if the response content has not changed since the last request according to the <code>Last-Modified</code> header. If omitted, no header check is performed.

That's a lot of options to keep track of, but it's unlikely that more than a few of them will be used for any one request. Even so, wouldn't it be convenient if we could set default values for these options for pages where we're planning to make a large number of requests?

8.4.2 Setting request defaults

Obviously the last question in the previous section was a setup. As you might have suspected, jQuery provides a way for us to set up a default set of Ajax properties that will be used when we don't override their values. This can make pages that initiate lots of similar Ajax calls much simpler.

The function to set up the list of Ajax defaults is `$.ajaxSetup()`, and its syntax is as follows:

Command syntax: `$.ajaxSetup`

`$.ajaxSetup(properties)`

Establishes the passed set of properties as the defaults for subsequent calls to `$.ajax()`.

Parameters

`properties` (Object) An object instance whose properties define the set of default Ajax properties. These are the same properties described for the `$.ajax()` function in table 8.2.

Returns

Undefined.

At any point in script processing, usually at page load (but can be at any point of the page authors' choosing), this function can be used to set up defaults to be used for all subsequent calls to `$.ajax()`.

NOTE Defaults set with this function aren't applied to the `load()` command. For utility functions such as `$.get()` and `$.post()`, the HTTP method can't be overridden by use of these defaults. Setting a default `type` of GET won't cause `$.post()` to use the GET HTTP method.

Let's say that we are setting up a page where, for the majority of Ajax requests (made with the utility function rather than the `load()` command), we want to set up some defaults so that we don't need to specify them on every call. We can, as the first statement in the header `<script>` element, write

```
$.ajaxSetup({
  type: 'POST',
  timeout: 5000,
```

```
    dataType: 'html',
    error: function(xhr) {
        $('#errorDisplay')
            .html('Error: ' + xhr.status + ' ' + xhr.statusText);
    }
})
```

This would ensure that every subsequent Ajax call (again, except via `load()`) would use these defaults, unless explicitly overridden in the properties passed to the Ajax utility function being used. Note the defaulting of an error callback. It's quite common for error, complete, and even `beforeSend` callbacks that should be applied to all Ajax invocations to be specified in this way.

Now, what about those *global functions* that were controlled by the `global` property?

8.4.3 Global functions

In addition to the ability to specify default functions to be executed for all Ajax requests by establishing them as defaults with `$.ajaxSetup()`, jQuery also allows us to attach functions to specific DOM elements. These functions will be triggered during the various phases of Ajax request processing or when a request ultimately succeeds or fails.

For example, to attach a function to an element with an `id` of `errorConsole` whose purpose is to display error messages, we write

```
$('#errorConsole').ajaxError(reportError);
```

The function `reportError` will be called in the event that any Ajax request fails.

When this, or any other of these global functions, is invoked, the first parameter passed to the callback function consists of a JavaScript Object instance with the following two properties:

- `type`—A string that contains the type of global function invoked—*ajaxError*, for example.
- `target`—A reference to the DOM element to which the global function was attached. In the case of the previous example, it's the element with the `id` of `errorConsole`.

We'll call this construct the *Global Callback Info* object. Some global function types are passed additional parameters (as we'll see shortly), but this common first parameter can be used to identify what global function type triggered the callback and to which element the function was attached.

The commands that can be used to attach these global functions are `ajaxStart()`, `ajaxSend()`, `ajaxSuccess()`, `ajaxError()`, `ajaxComplete()`, and `ajaxStop()`.

As the syntax for the commands that attach each of these function types are identical, they are presented in the following single syntax description:

Command syntax: Ajax global functions

```
ajaxStart(callback)
ajaxSend(callback)
ajaxSuccess(callback)
ajaxError(callback)
ajaxComplete(callback)
ajaxStop(callback)
```

Attaches the passed function to all matched elements invoked when the specified point in the processing of an Ajax request takes place.

Parameters

`callback` (Function) The callback function to be attached. See table 8.3 for information on when the callback is invoked and what parameters it will be passed.

Returns

The wrapped set.

Each of these global callbacks is invoked at a certain point during the processing of an Ajax request or conditionally depending upon the response status, assuming that global functions were enabled for the Ajax request. Table 8.3 describes when each global callback type is invoked and what parameters are passed to it.

Table 8.3 Global Ajax callbacks, listed in order of firing

Global callback type	When invoked	Parameters
<code>ajaxStart</code>	When a jQuery Ajax function or command is started, but before the XHR instance is created	<ul style="list-style-type: none"> ■ A Global Callback Info object with type set to <code>ajaxStart</code>
<code>ajaxSend</code>	After the XHR instance has been created, but before it's sent to the server	<ul style="list-style-type: none"> ■ A Global Callback Info object with type set to <code>ajaxSend</code> ■ The XHR instance ■ The properties used by the <code>\$.ajax()</code> function
<code>ajaxSuccess</code>	After the request has returned from the server and the response contains a success status code	<ul style="list-style-type: none"> ■ A Global Callback Info object with type set to <code>ajaxSuccess</code> ■ The XHR instance ■ The properties used by the <code>\$.ajax()</code> function

continued on next page

Table 8.3 Global Ajax callbacks, listed in order of firing (continued)

Global callback type	When invoked	Parameters
<code>ajaxError</code>	After the request has returned from the server and the response contains a failure status code	<ul style="list-style-type: none"> ■ A Global Callback Info object with type set to <code>ajaxError</code> ■ The XHR instance ■ The properties used by the <code>\$.ajax()</code> function ■ An exception object returned by the XHR instance, if any
<code>ajaxComplete</code>	After the request has returned from the server and after any declared <code>ajaxSuccess</code> or <code>ajaxError</code> callbacks have been invoked	<ul style="list-style-type: none"> ■ A Global Callback Info object with type set to <code>ajaxComplete</code> ■ The XHR instance ■ The properties used by the <code>\$.ajax()</code> function
<code>ajaxStop</code>	After all other Ajax processing is complete and any other applicable global callbacks have been invoked	<ul style="list-style-type: none"> ■ A Global Callback Info object with type set to <code>ajaxStop</code>

Let's put together a simple example of how some of these commands can easily be used to report the success or failure of an Ajax request. The layout of our test page (it's too simple to be called a lab) is as shown in figure 8.7 and is available in the file `chapter8/listing.8.6.html`.

On this page we have defined three fieldsets: one that contains buttons that will initiate Ajax requests, one that contains an area for success messages, and one that contains an area for error messages. The HTML markup to set up this structure is suitably simple.

**Figure 8.7** The initial layout of the page we'll use to examine the use of the Ajax global callbacks

```

<body>
  <fieldset>
    <legend>Initiate Ajax Requests</legend>
    <div>
      <button type="button" id="goodButton">
        Initiate successful request
      </button>
      <button type="button" id="badButton">
        Initiate failed request
      </button>
    </div>
  </fieldset>

  <fieldset>
    <legend>Success display</legend>
    <div id="successDisplay"></div>
  </fieldset>

  <fieldset>
    <legend>Error display</legend>
    <div id="errorDisplay"></div>
  </fieldset>
</body>

```

The ready handler for the page has three tasks:

- 1 Set up the click handlers for the buttons
- 2 Establish a global function as a success listener attached to the *success area*
- 3 Establish a global function as a failure listener attached to the *error area*

Setting up the click handlers for the buttons is straightforward.

```

$('#goodButton').click(function(){
  $.get('reflectData.jsp');
});
$('#badButton').click(function(){
  $.get('returnError.jsp');
});

```

The *good* button is set up to initiate an Ajax request to a resource that will return a success status, and the *bad* button initiates a request to a resource that always returns an error status.

Now we use the `ajaxSuccess()` command to establish a success listener attached to the `<div>` element with the id of `successDisplay` as follows:

```

$('#successDisplay').ajaxSuccess(function(info){
  $(info.target)
    .append('<div>Success at '+new Date()+ '</div>');
});

```

This establishes a function that will be called when an Ajax request completes successfully. This callback is passed a Global Callback Info instance whose target identifies the bound element—in this case, `successDisplay`. We use that reference to construct a success message and display it in the success display area.

The binding of the error callback is similar.

```
$('#errorDisplay').ajaxError(function(info,xhr){
    $(info.target)
        .append('<div>Failed at '+new Date()+'</div>')
        .append('<div>Status: ' + xhr.status + ' ' +
            xhr.statusText+'</div>');
});
```

Using the `ajaxError()` command, we bind a callback function that will be invoked when an Ajax request fails. In this case, the XHR instance is passed to the function, and we use it to give the user information about the nature of the error.

Because each function is bound to only one global function type, the use of the `type` field of the Global Callback Info instance isn't needed. But notice how the two functions perform some similar processing? How can we combine these functions into a single function instance that uses the `type` field to operate more efficiently?

With the page displayed in the browser, click each button a number of times (the messages have timestamps, so it's easy to discern the order in which they are displayed). You might end up with a display such as shown in figure 8.8.

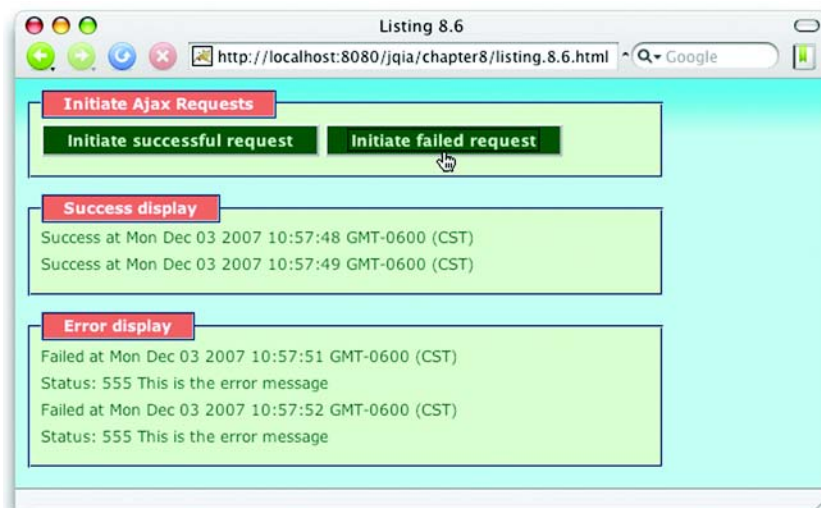


Figure 8.8 Clicking the buttons reveals how the callback functions get information and how they know which element they are attached to.

Before we move on to the next chapter, let's put all this grand knowledge to use, shall we?

8.5 Putting it all together

It's time for another comprehensive example. Let's put a little of everything we've learned so far to work: selectors, DOM manipulation, advanced JavaScript, events, effects, and Ajax. And to top it all off, we'll implement another jQuery command!

For this example, we'll once again return to The Boot Closet page. To review, look back at figures 8.2, 8.3, 8.5, and 8.6 because we're going to continue to enhance this page.

In the detailed information of the boots listed for sale (evident in figure 8.3), terms are used that our customers may not be familiar with—terms like *Goodyear welt* and *stitch-down construction*. We'd like to make it easy for customers to find out what these terms mean because an informed customer is usually a happy customer. And happy customers buy things!

We could be all 1998 about it and provide a glossary page that customers navigate to for reference, but that would move the focus away from where we want it—the pages where they can buy our stuff! We could be a *little* more modern about it and open a pop-up window to show the glossary or even the definition of the term in question. But even that's being a tad old-fashioned.

If you're thinking ahead, you might be wondering if we could use the `title` attribute of DOM elements to display a *tooltip* (sometimes called a *flyout*) containing the definition when customers hover over the term with the mouse cursor. Good thinking! That allows the definition to be shown in-place without the need for the customers to have to move their focus elsewhere.

But the `title` attribute approach presents some problems for us. First, the flyout only appears if the mouse cursor hovers over the element for a few seconds—and we'd like to be a bit more overt about it, displaying the information immediately after clicking a term—but, more importantly, some browsers will truncate the text of a `title` flyout to a length far too short for our purposes.

So we'll build our own!

We'll somehow identify terms that have definitions, change their appearance to allow the user to easily identify such terms, and instrument them so that a mouse click will display a flyout containing a description of the term. Subsequently clicking the flyout will remove it from the display.

Figure 8.9 displays a portion of our page showing the behavior we wish to add.

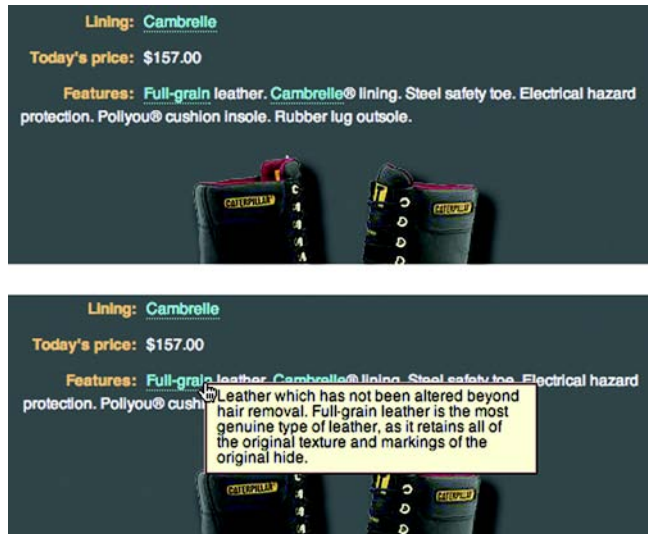


Figure 8.9
The before and after shots
of the behavior that we'll be
adding to the page

In the top part of the figure we see the Features description of the item with the terms *Full-grain* and *Cambrelle* highlighted. Clicking *Full-grain* causes the flyout containing its definition to be displayed as shown in the bottom part of the figure.

We could hard-code everything to make this happen right on the page, but we're smarter than that. As with our extensions to manipulate the select elements, we want to create a reusable component to use elsewhere on this, or any other, site. So once again, being jQuery-savvy, we'll implement it as a jQuery command.

8.5.1 Implementing the flyout behavior

As we recall, adding a jQuery command is accomplished by use of the `$.fn` property. We'll call our new plugin that looks up the definitions of terms *The Termifier*, and the command will be named `termifier()`.

The `termifier()` command will be responsible for instrumenting each element in its matched set to achieve the following goals:

- Establish a `click` handler on each matched element that initiates the display of *The Termifier* flyout.
- Once clicked, the term defined by the current element will be looked up using a server-side resource.
- Once received, the definition of the term will be displayed in a flyout using a fade-in effect.

- The flyout will be instrumented to fade out once clicked within its boundaries.
- The URL of the server-side resource, as well as the CSS class assigned to the flyout element, can be assignable by the page author but will have reasonable defaults.

The code that creates a jQuery command that meets these goals is shown in listing 8.8 and can be found in the file `chapter8/bootcloset/jquery.jqia.termifier.js`.

Listing 8.8 The implementation of the `termifier()` command

```

(function($) {
  $.fn.termifier = function(options) {
    options = $.extend({
      lookupResource: 'getTerm',
      flyoutClass: 'lookerUpperFlyout'
    }, options || {});
    this.attr('title', 'Click me for my definition!');
    return this.click(function(event) {
      $.ajax({
        url: options.lookupResource,
        type: 'GET',
        data: {term: this.innerHTML},
        dataType: 'html',
        success: function(data) {
          $('<div></div>')
            .css({
              position: 'absolute',
              left: event.pageX,
              top: event.pageY,
              cursor: 'pointer',
              display: 'none'
            })
            .html(data)
            .addClass(options.flyoutClass)
            .click(function() {
              $(this).fadeOut(1500, function() {$(this).remove();});
            })
            .appendTo('body')
            .fadeIn();
        }
      });
    });
    return false;
  };
})(jQuery);

```

1 Defines command API
2 Merges options with defaults
3 Establishes click handler on terms
4 Initiates request for term definition
5 Acts upon response data
6 Establishes click handler on flyout
7 Attaches flyout to DOM and fade in

Perhaps not as much code as expected, but there's a lot going on in there! Let's take it one step at a time.

First, we use the pattern that we learned in chapter 7 to establish the API for the `termifier()` command ❶. The only parameter expected is an object whose properties serve as our options. To be friendly, we provide a default set that we merge into the passed options using the services of the `$.extend()` utility function ❷. The defined options are as follows:

- `lookupResource`—Specifies the URL of the server-side resource to be used
- `flyoutClass`—The CSS class name applied to newly created flyout elements

As a helpful tip to our customers, we add a `title` attribute to the target element so that if they hover the mouse cursor over the highlighted term, they will see a message letting them know that clicking the term will do something wonderful.

We establish a `click` handler on every element in the matched set ❸. Remember that the function context (`this`) for a jQuery command is the matched set, so applying other jQuery commands to the matched set is as easy as calling the commands on `this`.

In the listener for the `click` event, we initiate the Ajax call that will retrieve the term definition. For maximum control, we use the `$.ajax()` function ❹ and pass it an object that defines the following options:

- The URL specified by the command options (either the default or one provided by the page author)
- An HTTP method of GET (because the request is clearly idempotent)
- A request parameter named `term` that's set to the content of the event target (the function context within the listener)
- Identification of the expected response data as HTML
- A `success` callback ❺ that uses the response data to create the flyout

A lot of the more interesting things happen in the `success` callback for the Ajax request. First, a new and empty `<div>` element is created, and then the following operations are performed on it (using the magic of jQuery chaining again):

- CSS styles are added to the `<div>` element that absolutely position it at the point of the mouse click event, change the mouse cursor to the hand shape, and hide the element from view.
- The response data, passed as the first parameter to the success callback and which we know contains the term definition, is inserted as the content of the `<div>` element.

- The CSS class identified by the `flyoutClass` option is added to the `<div>`.
- A click handler is established on the flyout `<div>` **6** that will cause it to slowly fade when clicked and then to be removed from the DOM tree once the fade effect has concluded.
- The newly created flyout `<div>` is added to the DOM by appending it to the `<body>` element.
- And finally, the flyout `<div>` is displayed by fading it in using the default rate **7**.

The implementation of the `termifier()` command makes sure to return the wrapped set as the result of the command (by returning the wrapped set as returned by the `click()` command) so that our new command can participate in any jQuery command chain.

Now, let's see what it takes to apply this command to our Boot Closet page.

8.5.2 Using The Termifier

Because we rolled all the complex logic of creating and manipulating The Termifier flyout into the `termifier()` command, using this new jQuery command on the Boot Closet page is relatively simple. But first we have some interesting decisions to make.

We need to decide how to identify the terms on the page. Remember, we need to construct a wrapped set of elements whose content contains the term elements for the command to operate on. We *could* use a `` element with a specific class name; perhaps something like

```
<span class="term">Goodyear welt</span>
```

Creating a wrapped set of these elements would be as easy as `$('.span.term')`.

But some might feel that the `` markup is a bit wordy. Instead, we'll leverage the little-used HTML tag `<abbr>`. The `<abbr>` tag was added to HTML 4 in order to help identify abbreviations in the document. Because the tag is intended purely for identifying document elements, none of the browsers do much with these tags either in the way of semantics or visual rendition, so it's perfect for our use.

NOTE HTML 4³ defines a few more of these document-centric tags such as `<cite>`, `<dfn>`, and `<acronym>`. The HTML 5 Draft Specification⁴

³ <http://www.w3.org/TR/html4/>

⁴ <http://www.w3.org/html/wg/html5/>

proposal adds even more of these document-centric tags whose purpose is to provide semantics rather than provide layout or visual rendition directives. Among such tags are `<section>`, `<article>`, and `<aside>`.

Therefore, the first thing that we need to do is modify the server-side resource that returns the item details to enclose terms that have glossary definitions in `<abbr>` tags. Well, as it turns out, the `getDetails.jsp` resource already does that. But because the browsers don't do anything with the `<abbr>` tag, we might not have even noticed unless we'd already taken a look inside the JSP or PHP file. This resource returns JSON data such as the following for an example item:

```
{
  name: 'Chippewa Harness Boot',
  sku: '7141922',
  height: '13"',
  lining: 'leather',
  colors: 'Black, Crazy Horse',
  price: '$188.00',
  features: '<abbr>Full-grain</abbr> leather uppers. Leather
  ➡ lining. <abbr>Vibram</abbr> sole. <abbr>Goodyear welt</abbr>.'
}
```

Note how the terms *Full-grain*, *Vibram* and *Goodyear welt* are identified using the `<abbr>` tag.

Now, on to the page itself. Starting with the code of listing 8.6 as a starting point, let's see what we need to add to the page in order to use The Termifier. We need to bring the new command into the page, so we add the following statement to the `<head>` section (after jQuery itself has loaded):

```
<script type="text/javascript"
  src="jquery.jqia.termifier.js"></script>
```

We need to apply the `termifier()` command to any `<abbr>` tags added to the page when item information is loaded, so we add a callback to the `load()` command that fetched the item information. That callback uses The Termifier to instrument all `<abbr>` elements. The augmented `load()` command (with changes in bold) is as follows:

```
$('#detailsDisplay').load(
  'getDetails.jsp',
  { style: styleValue },
  function(){
    $('#abbr').termifier({
      lookupResource: 'getTerm.jsp'
    });
  }
);
```

The added callback creates a wrapped set of all `<abbr>` elements and applies the `termifier()` command to them, specifying a server-side resource of `getTerm.jsp` that overrides the command's default.

And that's it.

Because we wisely encapsulated all the heavy lifting in our reusable jQuery command, using it on the page is even easier than pie! And we can as easily use it on any other page or any other site. Now that's what *engineering* is all about!

The only remaining task is to alter the appearance of the text elements so that users know which are clickable terms. To the CSS file, we add the following CSS properties for the `<abbr>` tag:

```
color: aqua;
cursor: pointer;
border-bottom: 1px aqua dotted;
```

These styles give the terms a link-ish appearance but with the subtle difference of using a dotted underline. This invites the users to click the terms, yet keeps them distinct from any true links on the remainder of the page.

The new page can be found in the file `chapter8/bootcloset/boot.closet.3.html`. Because the changes we made to the code of listing 8.6 are minimal (as we discussed), we'll spare some paper and not include the entire page listing here.

The updated page with our new functionality in action is shown in figure 8.10.

Our new command is useful and powerful, but there's always...

8.5.3 Room for improvement

Our brand-spankin'-new jQuery command is useful as is, but it does have some minor issues and the potential for some major improvements. To hone your skills, here's a list of possible changes you could make to this command or to the Boot Closet page:

- The server-side resource is passed the term in a request parameter named `term`. Add an option to the command giving the page author the ability to specify the name of the query parameter. Our client-side command shouldn't dictate how the server-side code is written.
- Add an option (or options) that allows the page author to control the fade durations or, perhaps, even to use alternate effects.
- The `Termifier` flyout stays around until the customer clicks it or until the page is unloaded. Add a `timeout` option to the command that automatically makes the flyout go away if it's still displayed after the timeout has expired.

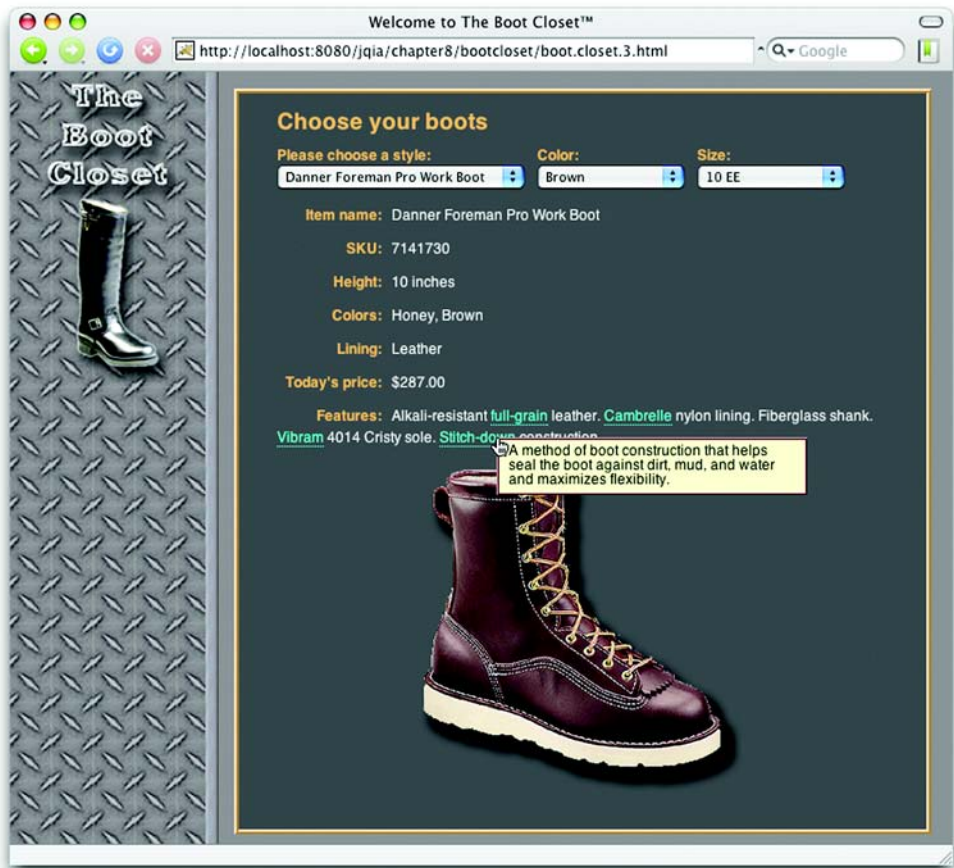


Figure 8.10 Our customer learns what *Stitch-down construction* is all about.

- Clicking the flyout to close it introduces a usability issue because the text of the flyout can't be selected for cut-and-paste. Modify the code so that it closes the flyout if the user clicks anywhere on the page *except* on the flyout.
- It's possible for multiple flyouts to be displayed if the user doesn't dismiss one flyout before clicking another term, even when a new style is selected. Add code to remove any previous flyout before displaying a new flyout and when a new style is picked.
- We don't do any error handling in our command. How would you enhance the command to gracefully deal with server-side errors?

- We achieved the appealing drop shadows in our images by using PNG files with partial transparencies. Although most browsers handle this file format well, IE6 does not and displays the PNG files with white backgrounds. To deal with this we could also supply GIF formats for the images without the drop shadows. How would you enhance the page to detect when IE6 is being used and to replace all the PNG references with their corresponding GIFs?
- While we're talking about the images, we only have one photo per boot style, even when multiple colors are available. Assuming that we have photo images for each possible color, how would you enhance the page to show the appropriate image when the color is changed?

Can you think of other improvements to make to this page or the `termifier()` command? Share your ideas and solutions at this book's discussion forum, which you can find at <http://www.manning.com/bibeault>.

8.6 Summary

Not surprisingly, this is one of the longest chapters in this book. Ajax is a key part of Rich Internet Applications, and jQuery is no slouch in providing a rich set of tools for us to work with.

For loading HTML content into DOM elements, the `load()` command provides an easy way to grab the content from the server and make it the content of any wrapped set of elements. Whether a GET or POST method is used is determined by whether data needs to be passed to the server or not.

When a GET is required, jQuery provides the utility functions `$.get()` and `$.getJSON()`; the latter is useful when JSON data is returned from the server. To force a POST, the `$.post()` utility function can be used.

When maximum flexibility is required, the `$.ajax()` utility function, with its ample assortment of options, lets us control most aspects of an Ajax request. All other Ajax features in jQuery use the services of this function to provide their functionality.

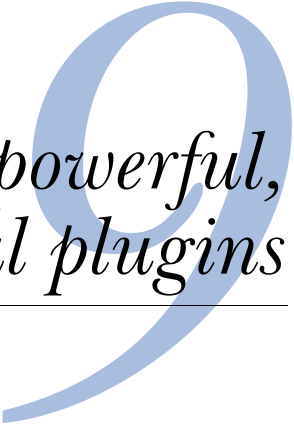
To make managing the bevy of options less of a chore, jQuery provides the `$.ajaxSetup()` utility function that allows us to set default values for any frequently used options to the `$.ajax()` function (and to all of the other Ajax functions that use the services of `$.ajax()`).

To round out the Ajax toolset, jQuery also allows us to monitor the progress of Ajax requests and associate these events with DOM elements via the

`ajaxStart()`, `ajaxSend()`, `ajaxSuccess()`, `ajaxError()`, `ajaxComplete()`, and `ajaxStop()` commands.

With this impressive collection of Ajax tools under our belts, it's easy to enable Rich Internet Application functionality in our web applications. And remember, if there's something that jQuery doesn't provide, we've seen that it's easy to extend jQuery by leveraging its existing features. Or, perhaps, there's already a plugin—official or otherwise—that adds exactly what you need!

Which is the subject of our next chapter...



*Prominent, powerful,
and practical plugins*

This chapter covers

- An overview of the jQuery plugins
- The official Form Plugin
- The official Dimensions Plugin
- The Live Query Plugin
- The UI Plugin

In the first eight chapters of this book, we focused on the capabilities that the core jQuery library makes available to us as page authors. But that's the tip of the iceberg! The immense collection of available jQuery plugins is impressive and gives us even more tools, all based on the jQuery core, to work with.

The creators of the core jQuery library carefully chose the functions needed by the vast majority of page authors and created a framework on which plugins can readily be built. This keeps the core footprint as small as possible and lets us, the page authors, decide how we want to spend the rest of our bandwidth allowance by picking and choosing what additional functionality is important enough to add to our pages.

It'd be an impossible task to try to cover all of the jQuery plugins in the space of a chapter, perhaps even in the space of a single book, so we had to choose which plugins to talk about here. It was a tough call, and the plugins we included are those that we felt were either important enough or useful enough to the majority of web application developers to warrant coverage.

Non-inclusion of a plugin in this chapter is most certainly not an indictment of a plugin's usefulness or quality! We just had to make some hard decisions.

You can find information on all the available plugins by visiting <http://docs.jquery.com/Plugins> or http://jquery.com/plugins/most_popular.

We also can't completely cover the plugins that we *will* discuss, but this chapter should give you a good basic understanding of the plugins and when they can be applied. Consult the official documentation for each plugin to fill in any gaps in the coverage here.

Let's start by looking at a plugin that we previously mentioned on a number of occasions.

9.1 The Form Plugin

Dealing with forms can be a hassle. Each control type has its particular quirks, and form submission can often take unintended paths. Core jQuery has a number of methods to help tame forms, but there's only so much that it can do for us. The purpose of the official Form Plugin is to help fill these gaps and help us take control of form controls.

This plugin can be found at <http://jquery.com/plugins/project/form> and resides in the file `jquery.form.js`.

It augments the form functionalities in three areas:

- Getting the values of form controls
- Clearing and resetting form controls
- Submitting forms (including file uploads) via Ajax

Let's start with getting form control values.

9.1.1 Getting form control values

The Form Plugin gives us two ways to get the values of form controls: as an array of values or as a serialized string. There are three methods that the Form Plugin provides to obtain control values: `fieldValue()`, `formSerialize()`, and `fieldSerialize()`.

Let's look at grabbing field values first.

Getting control values

We can get the values of form controls using the `fieldValue()` command. At first glance, you might think that `fieldValue()` and `val()` are redundant. But prior to jQuery 1.2, the `val()` command was considerably less capable, and `fieldValue()` was designed to make up for its deficiencies.

The first major difference is that `fieldValue()` returns an array of all the values for form controls in its wrapped set, whereas `val()` only returns the value of the first element (and only if that element is a form control). In fact, `fieldValue()` always returns an array, even if it only has one value or no values to return.

Another difference is that `fieldValue()` ignores any non-control elements in the wrapped set. If we create a set containing all elements on a page, an array that contains only as many control values as `fieldValue()` finds will be returned. But not all controls have values in this returned array: like the `val()` command, `fieldValue()`, by default, returns values only for controls that are deemed *successful*.

So what's a *successful* control? It's not a control that can afford a stable of fancy sports cars, but a formal definition in the HTML Specification¹ that determines whether a control's value is significant or not and whether it should be submitted as part of the form.

We won't go into exhaustive detail here; but, in a nutshell, successful controls are those that have name attributes, aren't disabled, and are checked (for checkable controls like check boxes and radio buttons). Some controls, like reset and

¹ <http://www.w3.org/TR/REC-html40/>

button controls, are always considered unsuccessful and never participate in a form submission. Others, like `<select>` controls, must have a selected value to be considered successful.

The `fieldValue()` command gives us the choice whether to include unsuccessful values or not; its syntax is as follows:

Command syntax: `fieldValue`

`fieldValue(excludeUnsuccessful)`

Collects the values of all successful form controls in the wrapped set and returns them as an array of strings. If no values are found, an empty array is returned.

Parameters

`excludeUnsuccessful` (Boolean) If `true` or omitted, specifies that any unsuccessful controls in the wrapped set be ignored.

Returns

A String array of the collected values.

We've set up another handy lab page to demonstrate the workings of this command. You'll find this page in the file `chapter9/form/lab.get.values.html`; when displayed in your browser, it will appear as shown in figure 9.1.

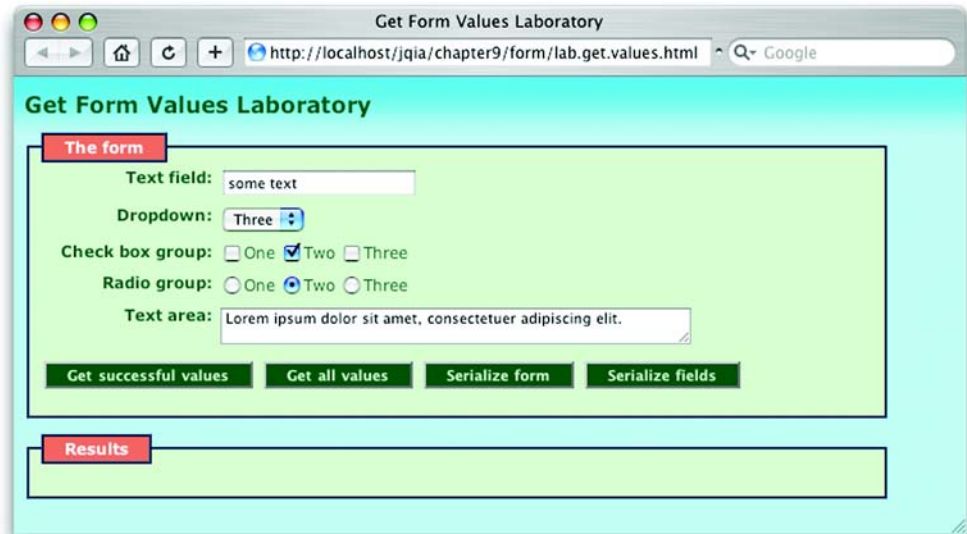


Figure 9.1 The Get Form Values Laboratory helps us to understand the operation of the `fieldValue()` and the `serialize()` commands.

Bring up this page, and leaving the controls in their initial state, click the Get Successful Values button. This causes the following command to be executed:

```
$('#testForm *').fieldValue()
```

This creates a wrapped set of all children of the test form, including all the labels and `<div>` elements, and executes the `fieldValue()` command on it. Because this command ignores all but form controls and, without a parameter, only includes successful controls, the results displayed on the page are

```
['some text','Three','cb.2','radio.2','Lorem ipsum dolor sit amet,
➡ consectetuer adipiscing elit.']
```

As expected, the values for the text field, the dropdown, the *checked* check box, the *checked* radio button, and the text area are collected into an array.

Now go ahead and click the Get All Values button, which executes the command

```
$('#testForm *').fieldValue(false)
```

The `false` parameter to this command instructs it not to exclude unsuccessful controls, and we can see the more inclusive results as follow:

```
['some text','Three','One','Two','Three','Four','Five','cb.1',
➡ 'cb.2','cb.3','radio.1','radio.2','radio.3','Lorem ipsum dolor
➡ sit amet, consectetuer adipiscing elit.',' ',' ',' ',' ']
```

Note that not only have the values for the unchecked check boxes and radio buttons been included but also empty strings for the values for the four buttons.

Now, have some fun playing around with the values of the controls and observing the behavior of the two forms of the `fieldValue()` command until you feel you've got it down.

Getting the values of the controls in an array can be useful when we want to process the data in some way; if we want to create a query string from the data, the `serialize` commands will do that for us. Let's see how.

Serializing control values

When we want to construct properly formatted and encoded query strings from the values of form controls, we turn to the `formSerialize()` and `fieldSerialize()` commands. Both of these wrapper methods collect values from the wrapped set and return a formatted query string with the names and values properly URL-encoded. The `formSerialize()` method accepts a form in the wrapped set and serializes all of the *successful* child controls. The `fieldSerialize()` command serializes all of the controls in the wrapped set and is useful for serializing only a portion of a form.

The syntaxes of these commands are as follow:

Command syntax: formSerialize**formSerialize(semantic)**

Creates and returns a properly formatted and encoded query string from the values of all successful controls in the wrapped form.

Parameters

`semantic` (Boolean) Specifies that the order of the values in the query string follows the semantic order of the elements—the order in which the elements are declared in the form. This option can be much slower than allowing random order.

Returns

The generated query string.

Command syntax: fieldSerialize**fieldSerialize(excludeUnsuccessful)**

Creates and returns a properly formatted and encoded query string from the values of controls in the wrapped form.

Parameters

`excludeUnsuccessful` (Boolean) If `true` or omitted, specifies that any unsuccessful controls in the wrapped set be ignored.

Returns

The generated query string.

The `semantic` parameter to `formSerialize()` deserves special note. When specified as `true`, the serialized values will be in the order that they would be in if the form were submitted through the conventional means, making any submission of these values exactly emulate a browser submission. We should only use this when absolutely necessary (it's usually not) because there's a performance penalty to be paid.

WARNING The `semantic` flag will cause the order of the parameters to be specified in the submitted data in semantic order, but what the server-side code does with this order isn't under the control of the client-side code. For example, when using servlets, a call to the `getParameterMap()` method of the request instance won't preserve the submitted order.

We can use the [Get Form Values Laboratory](#) page to observe the behavior of these commands. Load the page, and leaving the controls be, click the `Serialize Form` button. This will execute a `formSerialize()` command on the test form as follows:

```
$('#testForm').formSerialize()
```

This results in

```
text=some%20text&dropdown=Three&cb=cb.2&radio=radio.2&
➤ textarea=Lorem%20ipsum%20dolor%20sit%20amet%2C%20conse
➤ ctetuer%20adipiscing%20elit.
```

Notice that all successful form controls have their names and values collected, and the query string created using this data has been URL-encoded.

Clicking the Serialize Fields button executes the command

```
$('#testForm input').fieldSerialize()
```

The wrapped set created by this selector includes only a subset of the form’s controls: those of type `input`. The resulting query string, which includes only the wrapped control elements that are successful, is as follows:

```
text=some%20text&cb=cb.2&radio=radio.2
```

One reason that we might want to serialize form controls into a query string is to use as the submission data for an Ajax request. But wait! If we want to submit a form via Ajax rather than through the normal process, we can turn to yet more features of the Form Plugin. But before we get to that, let’s examine a few commands that allow us to manipulate the form controls’ values.

9.1.2 *Clearing and resetting form controls*

The Form Plugin provides two commands to affect the values of a form’s controls. The `clearForm()` command clears all fields in a wrapped form, whereas the `resetForm()` command resets the fields.

“Ummm, what’s the difference?” you ask.

When `clearForm()` is called to *clear* form controls, they are affected as follows:

- Text, password, and text area controls are set to empty values.
- `<select>` elements have their selection unset.
- Check boxes and radio buttons are unchecked.

When `resetForm()` is called to *reset* controls, the form’s native `reset()` method is invoked. This reverts the value of the controls to that specified in the original HTML markup. Controls like text fields revert to the value specified in their `value` attribute, and other control types revert to settings specified by `checked` or `selected` attributes.

Once again, we’ve set up a lab page to demonstrate this difference. Locate the file `chapter9/form/lab.reset.and.clear.html`, and display it in your browser. You should see the display shown in figure 9.2.

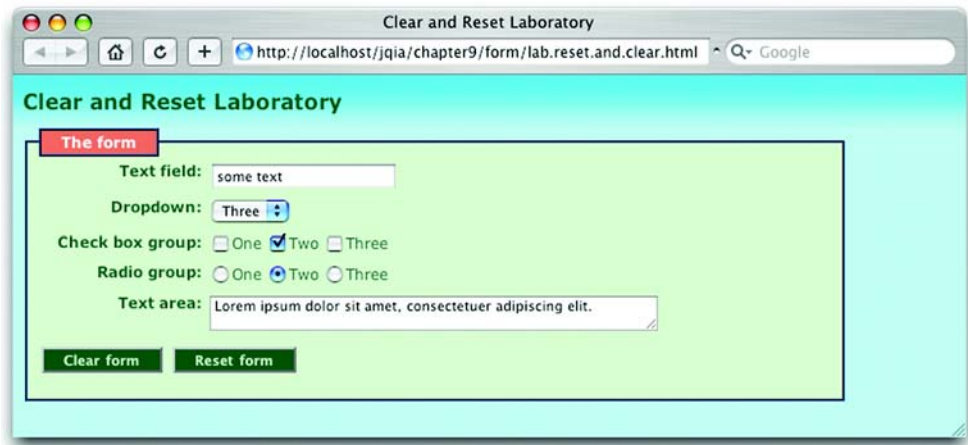


Figure 9.2 The Clear and Reset Laboratory shows us the difference between a reset and a clear.

Note that this familiar form has been initialized with values via its HTML markup. The text field and text area have been initialized via their `value` attributes, the dropdown has had one of its options selected, and one check box and one radio button have been checked.

Click the Clear Form button, and watch what happens. The text field and text area are cleared, the dropdown has no selection, and all check boxes and radio buttons are unchecked.

Now click the Reset Form button, and note how the controls all revert to their original values. Change the values of each control, and click Reset Form, noting how, once again, the original values are restored.

The syntaxes for these commands are as follow:

Command syntax: `clearForm`

`clearForm()`

Clears the value of any controls in the wrapped set or that are descendants of elements in the wrapped set

Parameters

none

Returns

The wrapped set

Command syntax: resetForm**resetForm()**Calls the native `reset()` method of forms in the wrapped set**Parameters**

none

Returns

The wrapped set

Now let's see how the Form Plugin helps us to submit forms via Ajax requests.

9.1.3 Submitting forms through Ajax

Back in chapter 8, we saw how easy jQuery makes initiating Ajax requests, but the Form Plugin makes things even easier. We could use the serialization commands introduced in section 9.1.1, but wait! There's more! The Form Plugin makes it even easier to hijack form requests.

The Form Plugin introduces two new commands for submitting forms via Ajax: one that initiates an Ajax request under script control, passing data in a target form as the request's parameters, and another that instruments any form to reroute its submission as an Ajax request.

Both approaches use the jQuery Ajax core functions to perform the Ajax request, so all the global jQuery hooks continue to be applied even when using these methods in place of the core jQuery Ajax API.

Let's start by examining the first approach.

Grabbing form data for an Ajax request

When we developed the e-commerce examples of chapter 8, we encountered a number of situations in which we needed to grab values from form controls to send them to the server via an Ajax request—a common real-world requirement. We saw that the core Ajax function made this a simple affair, particularly when we only needed to grab a handful of form values.

The combination of the Form Plugin's `serializeForm()` method and the core Ajax functions makes submitting *all* the controls in a form even easier. But even easier than *that*, the Form Plugin makes submitting an entire form through Ajax almost trivial with the `ajaxSubmit()` command.

This command, when applied to a wrapped set containing a form, grabs the names and values of all the successful controls in the target form and submits them as an Ajax request. We can supply information on how to make the request

to the method, or we can allow the request to default from the settings on the target form.

Let's look at its syntax.

Command syntax: `ajaxSubmit`

`ajaxSubmit(options)`

Generates an Ajax request using the *successful* controls within the form in the wrapped set. The options parameter can be used to specify optional settings, or these settings can be defaulted as described in the following table.

Parameters

`options` (Object|Function) An optional object hash containing properties as described in table 9.1. If the only desired option is a success callback, it can be passed in place of the options hash.

Returns

The wrapped set.

The `options` parameter can be used to specify exactly how the request is to be made. The optional properties are described in table 9.1, and all properties have defaults designed to make it easy to generate requests with the minimum of fuss and bother. It's common to call this method with no options and let all the defaults apply.

Table 9.1 The optional properties for the `ajaxSubmit()` command, listed according to likelihood of use

Name	Description
<code>url</code>	(String) The URL to which the Ajax request will be submitted. If omitted, the URL will be taken from the <code>action</code> attribute of the target form.
<code>type</code>	(String) The HTTP method to use when submitting the request, such as GET or POST. If omitted, the value specified by the target form's <code>method</code> attribute is used. If not specified and the form has no <code>method</code> attribute, GET is used.
<code>dataType</code>	(String) The expected data type of the response, which determines how the response body will be post-processed. If specified, it must be one of the following: <ul style="list-style-type: none"> ■ <code>xml</code>—Treated as XML data. Any success callback will be passed the <code>responseXML</code> document. ■ <code>json</code>—Treated as a JSON construct. The JSON is evaluated, and the result is passed to any success callback. ■ <code>script</code>—Treated as JavaScript. The script will be evaluated in the global context. If omitted, no post-processing of the data (except as specified by other options such as <code>target</code>) takes place.

continued on next page

Table 9.1 The optional properties for the `ajaxSubmit()` command, listed according to likelihood of use (continued)

Name	Description
target	(String Object Element) Specifies a DOM element or elements to receive the response body as content. This can be a string depicting a jQuery selector, a jQuery wrapper containing the target elements, or a direct element reference. If omitted, no element receives the response body.
beforeSubmit	(Function) Specifies a callback function invoked prior to initiating the Ajax request. This callback is useful for performing any pre-processing operations including the validation of form data. If this callback returns the value <code>false</code> , the form submission is cancelled. This callback is passed the following three parameters: <ul style="list-style-type: none"> ■ An array of the data values passed to the request as parameters. This is an array of objects; each contains two properties, <code>name</code> and <code>value</code>, containing the name and value of a request parameter. ■ The jQuery matched set that the command was applied to. ■ The <code>options</code> object that was passed to the command. If omitted, no pre-processing callback is invoked.
success	(Function) Specifies a callback invoked after the request has completed and returned as response with successful status. This callback is passed the following three parameters: <ul style="list-style-type: none"> ■ The response body as interpreted according to the <code>dataType</code> option. ■ A string containing <code>success</code>. ■ The jQuery matched set that the command was applied to. If omitted, no success callback is invoked. If this is the only option to be specified, this function can be passed directly to the command in place of the <code>options</code> hash. Note that no provisions have been made for a callback upon error conditions.
clearForm	(Boolean) If specified and <code>true</code> , the form is cleared after a successful submission. See <code>clearForm()</code> for semantics.
resetForm	(Boolean) If specified and <code>true</code> , the form is reset after a successful submission. See <code>resetForm()</code> for semantics.
semantic	(Boolean) If specified and <code>true</code> , the form parameters are arranged in semantic order. The only difference this makes is in the location of the parameters submitted for input element of type <i>image</i> when the form is submitted by clicking that element. Because there's overhead associated with this processing, this option should be enabled only if parameter order is important to the server-side processing and image input elements are used in the form.
other options	Any options that are available for the core jQuery <code>\$.ajax()</code> function, as described in table 8.2, can be specified and will pass through to the lower-level call.

Despite the number of options, calls to `ajaxSubmit()` are frequently quite simple. If all we need to do is submit the form to the server (and don't have anything to do when it completes), the call is as Spartan as

```
$('#targetForm').ajaxSubmit();
```

If we want to load the response into a target element or elements:

```
$('#targetForm').ajaxSubmit( { target: '.target' } );
```

If we want to handle the response on our own in a callback:

```
$('#targetForm').ajaxSubmit(function(response){
    /* do something with the response */
});
```

And so on. Because there are sensible defaults for all options, we only need to specify as much information as needed to tune the submission to our desires.

WARNING Because the options hash is passed to the `beforeSubmit` callback, you might be tempted to modify it. Tread carefully! It's obviously too late to change the `beforeSubmit` callback because it's already executing, but you can add or change other simple settings like `resetForm` or `clearForm`. Be careful with any other changes; they could cause the operation to go awry. Please note that you can't add or change the `semantic` property because its work is already over by the time the `beforeSubmit` callback is invoked.

If you were wondering if a lab page had been set up for this command, wonder no more! Bring up the page `chapter9/form/lab.ajaxSubmit.html` in your browser, and you'll see the display in figure 9.3.

NOTE Note that, because we're going to be submitting requests to the server, you must run this page under an active web server as described for the examples of chapter 8 in section 8.2.

This lab presents a now-familiar form that we can operate upon with the `ajaxSubmit()` command. The topmost pane contains the form itself; the middle pane contains a control panel that allows us to add the `resetForm` or `clearForm` options to the call; and a results pane will display three important bits of information when the command is invoked—the parameter data that was submitted to the request, the options hash that was passed to the command, and the response body.

If you care to inspect the code of the lab, you'll note that the first two items of information are displayed by a `beforeSubmit` callback, and the third by the

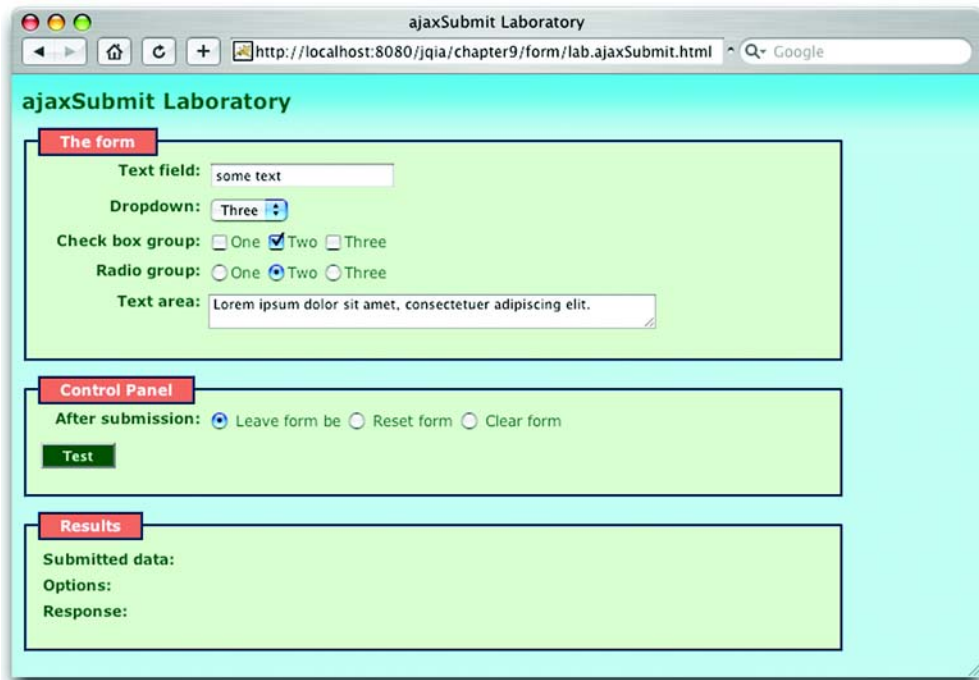


Figure 9.3 The ajaxSubmit Laboratory lets us play around with the workings of the `ajaxSubmit()` method.

target option. (For clarity, the `beforeSubmit` function isn't shown as part of the options display.)

When the Test button is clicked, a request is initiated via an `ajaxSubmit()` command applied to a wrapped set containing the form of the first pane. The URL of the request defaults to the action of that form: `reflectData.jsp`, which formats an HTML response depicting the parameters passed to the request.

Leaving all controls as they are upon initial load, click the Test button. You'll see the results as shown in figure 9.4.

The Submitted data, as expected, reflects the names and values of all successful controls; note the absence of unchecked check boxes and radio buttons. This perfectly mimics the submission of data that would occur if the form were to be submitted normally.

The Options used to make the request are also shown, allowing us to see how the request was made as we change the options in the Control Panel. For example,

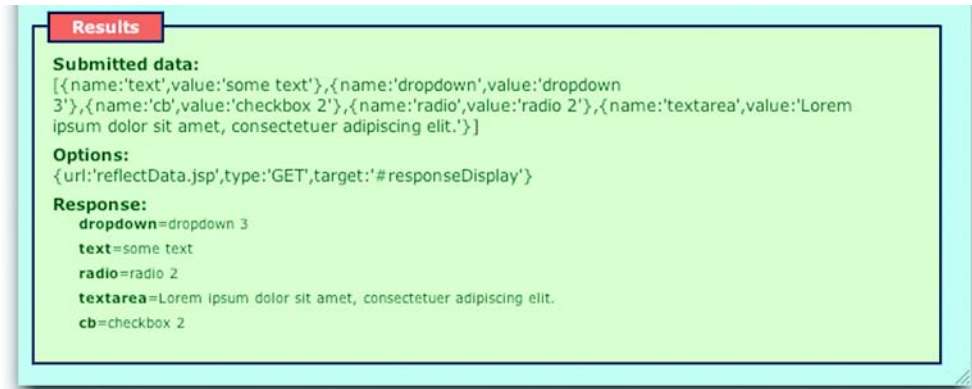


Figure 9.4 The Results pane shows us the data sent to the request, the options used to invoke the command, and the response body reflecting the data passed to the server resource.

if we check the Reset Form check box and click Test, we'll see how the `resetForm` option has been added to the method call.

The parameters detected by the server-side resource (by default, a JSP) are shown last. We can compare the response with the Submitted data to make sure that they always jive.

Run through various scenarios in the lab, changing form data and options to suit your whims, and observe the results. This should allow you to get a good understanding of how the `ajaxSubmit()` method operates.

In this section, we've assumed that we want to initiate a request using a form's data under script control. We'd want to do this when an event other than a normal semantic submission event takes place—perhaps, clicking a button other than a submit button (as in the lab page) or a mouse event such as the one we used to invoke The Termifier requests in the examples of chapter 8. But sometimes, perhaps most often, the request submission will be the result of a normal semantic submission event.

Let's see how the Form Plugin helps us set that up.

Hijacking a form's submission

The `ajaxSubmit()` method is great for those times when we want to initiate a request under script control as a result of an event other than a form submission; but, often, we want to take a conventional form submission and hijack it, sending it to the server as an Ajax request rather than the usual full-page refresh.

We could leverage our knowledge of event handling and the `ajaxSubmit()` command to reroute the submission ourselves. As it turns out, we won't have to; the Form Plugin anticipates this need with the `ajaxForm()` method.

This method instruments the form so that the submission is blocked when the form is submitted through one of the normal semantics events (such as clicking a submit button or pressing the Enter key when the form has focus) and an Ajax request that emulates the request is initiated.

`ajaxForm()` uses `ajaxSubmit()` under the covers, so it's not surprising that their syntaxes are similar.

Command syntax: `ajaxForm`

`ajaxForm(options)`

Instruments the target form so that when submission of the form is triggered, the submission reroutes through an Ajax request initiated via the `ajaxSubmit()` command. The `options` parameter passed to this method is passed on to the `ajaxSubmit()` call.

Parameters

`options` (Object|Function) An optional object hash containing properties as described in table 9.1. If the only desired option is a success callback, it can be passed in place of the options hash.

Returns

The wrapped set.

Typically, we apply `ajaxForm()` to a form in the ready handler; then, we can forget about it and let the command apply instrumentation to reroute the form submission on our behalf.

It's possible, indeed customary, to declare the markup for HTML forms as if they are going to be submitted normally and to let `ajaxForm()` pick up these values from the declaration of the form. For times when users have disabled JavaScript, the form degrades gracefully and submits normally without us having to do anything special whatsoever. How convenient!

If, at some point after a form has been bound with `ajaxForm()`, we need to remove the instrumentation to let the form submit normally, the `ajaxFormUnbind()` command will accomplish that.

For fans of the lab pages, an `ajaxForm` Laboratory can be found in the file `chapter9/form/lab.ajaxForm.html`. Loaded into a browser, this page will appear as shown in figure 9.5.

Command syntax: ajaxFormUnbind**ajaxFormUnbind()**

Removes the instrumentation applied to the form in the wrapped set so that its submission can occur normally

Parameters

none

Returns

The wrapped set

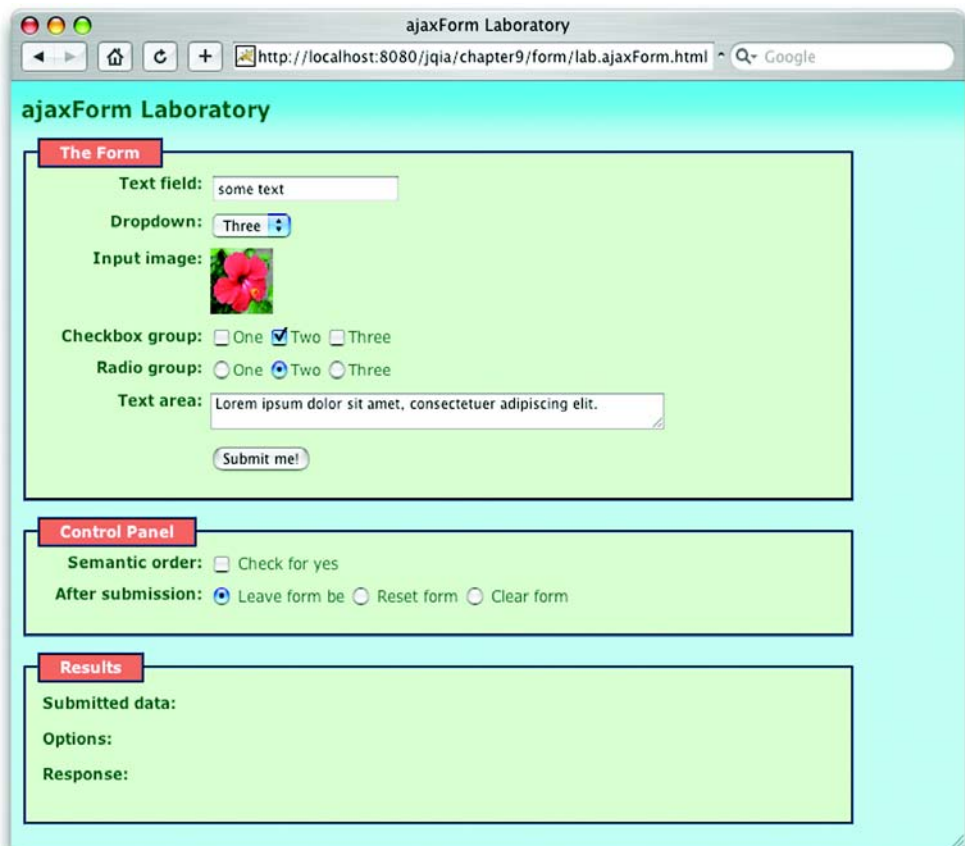


Figure 9.5 The ajaxForm Laboratory page allows us to observe the hijacking of form submission to an Ajax request.

This lab looks and works a lot like the ajaxSubmit Laboratory with a few important changes:

- The Test button has been removed and the Submit me! button has been added to the form.
- The Control Panel allows us to specify whether the `semantic` property is added to the options.
- An input element of type *image* has been added so that we can observe the difference in behavior that occurs when `semantic` is set to `true`.

This form can be submitted in the following three ways:

- Clicking the Submit me! button
- Pressing the Enter key while the focus is on a focusable element
- Clicking the Input image control (hibiscus blossom)

In any of these cases, you'll see that the page isn't refreshed; the form submission is rerouted through an Ajax request whose results are shown in the bottom pane of the page. Once again, play around with the controls on this page to become familiar with how the `ajaxForm()` command operates. When you have it down, we have one more Form Plugin subject to tackle.

9.1.4 **Uploading files**

A somewhat hidden, but useful, feature of the Form Plugin is its ability to automatically detect and deal with forms that need to upload files specified by input elements of type *file*. Because XHR is unable to accommodate such requests, the `ajaxSubmit()` command (and by proxy `ajaxForm()`) reroutes the request to a dynamically created and hidden `<iframe>`, while setting the content type of the request correctly as *multipart/form-data*.

The server code must be written to handle such file upload requests and multipart forms; but, from the viewpoint of the server, the request looks like any other multipart request generated by a conventional form submission. And from the perspective of the page code, this works exactly like a regular `ajaxSubmit()`.

Bravo!

Now let's set our sights on another useful jQuery plugin.

9.2 The Dimensions Plugin

Knowing the exact position and dimensions of an element is sometimes key to creating Rich Internet Applications. For example, when implementing dropdown menus, we want the menu to appear in a precise position in relation to its triggering element.

Core jQuery has the `width()`, `height()`, and `offset()` commands but lacks the ability to precisely locate an element in all circumstances. That's where the Dimensions Plugin comes in.

Let's take a run through its API.

9.2.1 Extended width and height methods

The Dimensions Plugin extends the core `width()` and `height()` commands so that they can be used to obtain the width or height of the window and document objects; something the core commands can't do. The syntaxes for these extended commands are as follow:

Command syntax: width

width()

Returns the width of the first element, window, or document object in the wrapped set. If the first wrapped element isn't the window or the document, the core jQuery command is called.

Parameters

none

Returns

The width of the window, document, or element.

Command syntax: height

height()

Returns the height of the first element, window, or document object in the wrapped set. If the first wrapped element isn't the window or the document, the core jQuery command is called.

Parameters

none

Returns

The height of the window, document, or element.

These extended commands don't interfere with the corresponding core commands when passed a value as a parameter (in order to set the width or height of elements), *except* when the first element is the window or document element. In such cases, the commands act as if no parameter was passed and the width or height of the window or document is returned. Be warned, and code accordingly.

The `width()` and `height()` commands return the dimensions assigned to the *content* of an element, but sometimes we want to account for other aspects of the box model, such as any padding or border applied to the element. For such occasions, the Dimensions Plugin provides two sets of commands that take these other dimensions into account.

The first of these, `innerWidth()` and `innerHeight()`, measure not only the content of the element but any padding applied to it as well. The second set, `outerWidth()` and `outerHeight()`, include not only the padding but also any border and, optionally, margins.

Command syntax: `innerWidth` and `innerHeight`

`innerWidth()`

`innerHeight()`

Returns the *inner* width or height of the first element in the wrapped set. The inner dimension includes the content and any padding applied to the element.

Parameters

none

Returns

The inner width or height of the first element in the wrapped set.

Command syntax: `outerWidth` and `outerHeight`

`outerWidth()`

`outerHeight()`

Returns the *outer* width or height of the first element in the wrapped set. The outer dimension includes the content, any padding, and any border applied to the element.

Parameters

`options` (Object) An object hash that accepts a single option, `margin`, which specifies whether margins should be accounted for in the calculation. The default is `false`.

Returns

The outer width or height of the first element in the wrapped set.

Note that, for all the inner and outer methods, specifying window or document have the same effect.

Now let's learn about the other dimensions this plugin allows us to locate.

9.2.2 Getting scroll dimensions

As has long been true for user interfaces of all types, content doesn't always fit in the space allotted to it. This issue has been addressed via the use of scrollbars, which allow users to scroll through all the content even if it can't all be seen within a viewport at once. The web is no different; content frequently overflows its bounds.

We may need to know the scrolled state of the window or of content elements that allow scrolling when trying to place new content (or move existing content) in relation to the window or scrolled element. Additionally, we may want to affect the scrolled position of the window or scrolled element.

The Dimensions Plugin allows us to obtain or set the scrolled position of these elements with the `scrollTop()` and `scrollLeft()` methods.

Command syntax: `scrollTop` and `scrollLeft`

`scrollTop(value)`

`scrollLeft(value)`

Gets or sets the scroll dimensions for the window, document, or scrollable content element. Scrolled elements are content-containing elements with a CSS `overflow`, `overflow-x`, or `overflow-y` value of `scroll` or `auto`.

Parameters

value (Number) The value, in pixels, to which the scroll top or left dimension is set. Unrecognized values are defaulted to 0. If omitted, the current value of the top or left scroll dimension is obtained and returned.

Returns

If a value parameter is provided, the wrapped set is returned. Otherwise, the requested dimension is returned.

Wrapping either window or document produces the same results.

Do you want to play with this in a lab page? If so, bring up the file `chapter9/dimensions/lab.scroll.html`, and you'll see the display in figure 9.6.

This lab page allows us to apply top and left scroll values to a test subject and to the window itself; it uses the *getter* version of the `scrollTop()` and `scrollLeft()` commands to display the scroll values of the scroll dimensions at all times.

The Lab Control Panel pane of this page contains two text boxes in which to enter numeric values to be passed to the `scrollTop()` and `scrollLeft()`

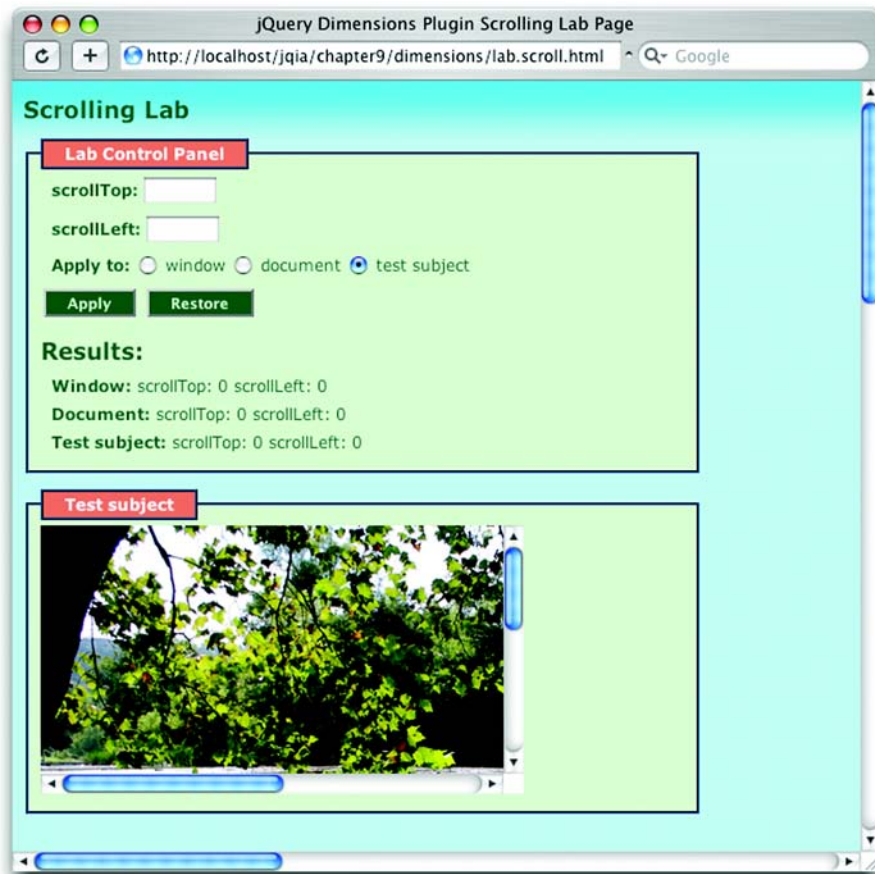


Figure 9.6 The Scrolling Lab lets us observe the effects of the `scrollTop()` and `scrollLeft()` methods.

commands, as well as three radio buttons that allow us to choose which target the commands are applied to. We can choose the Window, the Document, or the Test subject `<div>` element. Note that we set the height and width of the `<body>` element of the page to the ridiculous size of 2000 pixels square to force the window to show scrollbars.

The Apply button applies the specified values to the specified target, and the Restore button sets the scroll values for all targets back to 0. Below the buttons, a section shows the current values for the three targets in real time.

The Test subject pane contains the test subject: a 360 by 200 pixel `<div>` element that contains an image that's much larger than can be displayed in that size.

The CSS `overflow` value for this element is set to `scroll`, causing scrollbars to appear so that we can pan around the image.

Work through the following exercises using this page:

- *Exercise 1*—Using the scrollbars on the Test subject, pan around the image. Watch the Results display, and note how the current scroll values are kept up to date. A `scroll` event handler established on that element calls the `scrollTop()` and `scrollLeft()` methods to obtain these values for display.
- *Exercise 2*—Repeat the steps of exercise 1, except this time pan around the page using the window’s scrollbars. Watch the Results (unless you move them off-screen), and note how the window’s scroll values change as you pan around the page. Also, notice how the values for the Document stay in lockstep with those of the Window, emphasizing that, for the scroll methods, specifying either the window or the document performs the same action. We’ve included both the window and the document as targets to convince you of this point.
- *Exercise 3*—Click the Restore button to set everything to *normal*. Select the Test subject as the target, and enter scroll values into the text boxes, such as 100 and 100. Click the Apply button. Oh! What a pretty waterfall! Test other values to see how they affect the Test subject when the Apply button calls the `scrollTop()` and `scrollLeft()` methods.
- *Exercise 4*—Repeat exercise 3 with the Window as the target.
- *Exercise 5*—Repeat exercise 3 with the Document as the target. Convince yourself that, whether you specify the Window or the Document as the target, the same thing happens in all cases.

9.2.3 Of offsets and positions

We might, at first, think that obtaining the position of an element is a simple task. All we need to do is to figure out where the element is in relation to the window origin, right? Well, no.

When the `top` and `left` CSS values are applied to an element, these values are in relation to the element’s *offset parent*. In simple cases this offset parent is the window (or, more precisely, the `<body>` element loaded into the window); but, if any ancestor of an element has a CSS `position` value of `relative` or `absolute`, the closest such ancestor is the element’s offset parent. This concept is also referred to as the *positioning context* of the element.

When determining the location of an element, it’s important to know which position we’re asking for. Do we want to know the position of the element in

relation to the window origin or to its offset parent? Other factors can also be involved. For example, do we want dimensions such as border accounted for?

The Dimensions Plugin handles all of that for us, starting with obtaining the offset parent.

Command syntax: `offsetParent`

`offsetParent`

Returns the offset parent (positioning context) for the first element in the wrapped set. This is the closest ancestor with a `position` value of `relative` or `absolute`, or the `<body>` element if no such ancestor is found. This method should only be applied to visible elements.

Parameters

none

Returns

The offset parent element.

When we want to obtain the relative position of an element from its offset parent, we can use the `position()` command.

Command syntax: `position`

`position()`

Returns the position values (`top` and `left`) of the first element in the wrapped set relative to its offset parent.

Parameters

none

Returns

An object with two properties, `top` and `left`, containing the position values of the element relative to its positioning context (offset parent).

This command is useful for when we want to reposition an element in relation to its current location, but sometimes we want to know the position of an element in relation to the `<body>` element (regardless of what its offset parent is) and to have a little more control over how the calculation is made. For those times, the Dimensions Plugin provides the `offset()` command.

Command syntax: offset**offset(options, results)**

Returns offset information for the first element in the wrapped set. By default, the information is relative to the `<body>` element, and the manner in which it's calculated is controlled by the settings in the `options` parameter.

Parameters

- `options` (Object) An object hash containing settings that control how the method performs its calculations. The possible values are as follows:
- `relativeTo`—(Element) Specifies an ancestor element of the wrapped element to base the relative offset on. This element should have a `position` value of `relative` or `absolute`. If omitted, the default is the `<body>` element.
 - `lite`—(Boolean) Specifies that certain browser-specific optimizations be skipped in the calculations. This will increase performance at the price of accuracy. Defaults to `false`.
 - `scroll`—(Boolean) Specifies whether scroll offsets should be taken into account. Defaults to `true`.
 - `padding`—(Boolean) Specifies whether padding should be included in the calculation. Defaults to `false`.
 - `border`—(Boolean) Specifies whether borders should be included in the calculation. Defaults to `false`.
 - `margin`—(Boolean) Specifies whether margins should be included in the calculation. Defaults to `true`.
- `results` (Object) An optional object to receive the results of the method. If omitted, a new object is created, populated with the results and returned as the value of the method. If specified, the passed object is augmented with the result properties, and the wrapped set is returned from the method. This is useful when you want the method to participate in a jQuery command chain.

Returns

The wrapped set if a `results` object is specified, the `results` object if not. The `results` object contains properties `top` and `left`, as well as `scrollTop` and `scrollLeft` unless the `scroll` option is explicitly set to `false`.

To ensure that the values returned from this method are accurate, we express dimensions and positions of the elements on the page in pixel values. The default settings for this method usually give an accurate representation; but, if we're more interested in speed than in accuracy (for example, if we're using this method on many elements in a tight loop), we might want to explore how the `lite` option and other settings work in our situation.

Let's move away from the realm of positions and measurements and look at another plugin that's helpful on pages with lots of dynamic elements and event handling.

9.3 The Live Query Plugin

If you've grokked some of the higher-level concepts presented in this book, you'll have noted that jQuery has a profound effect on the structure of the pages that we write, using it to best advantage. Employing the precepts of Unobtrusive JavaScript, our pages usually consist of the HTML markup in the body and a ready handler that sets up the behavior of the page, including establishing the event handlers for the elements defined within the body.

Not only does jQuery make it incredibly easy to set up our pages in this way, it also makes it easy for us to change the page radically during its loaded lifetime. During the life cycle of the page, many DOM elements that didn't exist when the ready handler was executed can later be added to the DOM tree. When adding such elements, we frequently must establish event handlers as we create the elements to define their behavior, as we did for the initially loaded elements. The novice web author might churn out lots of repeated, cut-and-pasted code, but more experienced developers factor out common elements into functions or JavaScript classes.

But wouldn't it be nice if we could declare the behavior of all elements that will ever exist on the page in the ready handler regardless of whether they exist at page load or not?

Seems like a pipe dream, doesn't it? But it's not; the Live Query Plugin allows us to do just that!

Live Query lets us establish the following behaviors for DOM elements based on their match to a jQuery selector that we define:

- Establish events handlers for elements that match the selector
- Trigger a function to be executed when any element matches the selector
- Trigger a function to be executed when any element no longer matches the selector

The Live Query Plugin also allows us to unbind any of these behaviors at any time.

We'll start our overview of Live Query by looking at how it allows us to establish event handlers for DOM elements whether they exist or not.

9.3.1 Establishing proactive event handlers

The Live Query Plugin allows us to establish event handlers in a proactive fashion—establishing event handlers on elements that match a jQuery selector now or at anytime in the future. The established handlers apply, not only to existing elements that match the selector when the handler is established, but also to any elements that might match the selector pattern later on in the life cycle of the

page—including existing elements that might be changed to match the pattern and newly created elements that match.

When such elements are changed so that they no longer match the selector, handlers established by Live Query are automatically removed from those elements.

The changes that affect whether an element matches the selector pattern or not hinge on using jQuery methods. If the elements are mucked about with outside the realm of jQuery, obviously Live Query loses the hooks it needs to keep track of the elements. If we absolutely need to make changes outside jQuery's control, Live Query does have a way to help us deal with that; we'll get to that later.

All of the Live Query behaviors, including event listeners, are established on elements using the `livequery()` method. The format that establishes proactive event handlers is as follows:

Command syntax: `livequery`

`livequery(event, listener)`

Establishes a function as the event handler for the specified event type on all elements in the matched set and any elements that will match the selector of the matched set at a later point.

Parameters

- `event` (String) The type of event for which to establish the listener. This is the same set of events as used with the jQuery `bind()` command.
- `listener` (Function) The function to establish as the event listener. The function context (`this`) for each invocation is the matched element.

Returns

The wrapped set.

This form of `livequery()` is called exactly like the jQuery `bind()` command. Like `bind()`, it establishes the handler for all elements in the matched set. But it also automatically establishes the handler on any elements that match the selector pattern at any time while the page is loaded. It also unbinds the listener from any elements, including those from the original matched set, that no longer match the pattern.

This is immensely powerful. It allows us to set up the behavior of elements that match a selector of our choosing *once* in the ready handler, without having to worry about keeping track of such things later as elements are changed or added to the page. How cool is that?

The establishment of event handlers is a special case—a common one, which is why it gets special attention—of performing actions when elements are changed (or added) so that they match or no longer match the original selector pattern. We might like to do a multitude of other things at such points, and Live Query doesn't disappoint us.

9.3.2 Defining match and mismatch listeners

If we want to perform an action (other than binding or unbinding event handlers) when elements make a transition into or out of the state of matching a particular selector, we can use another form of the `livequery()` command.

Command syntax: `livequery`

`livequery(onmatch, onmismatch)`

Establishes callback functions invoked when elements transition into or out of the state of matching the selector for the matched set.

Parameters

- `onmatch` (Function) Specifies a function that serves as the *match listener*. This function is invoked for any element (established as the function context) that transitions into the matched state. If any existing elements match at the time of this method call, the function is called immediately for each such element.
- `onmismatch` (Function) Specifies an optional function that serves as the *mismatch listener*. This function is invoked for any element (established as the function context) that transitions out of the matched state. If omitted, no *mismatch listener* is established.

Returns

The wrapped set.

If we only want to establish the optional mismatch listener, we can't do so by passing `null` as the first parameter to this method because this causes the second parameter to be established as the match listener just as if it had been passed as the first argument. Instead, we pass in a no-op function as the first parameter:

```
$('#div.whatever').livequery(
  function() {},
  function() { /* mismatch actions go here */ }
);
```

As with the event listeners established by Live Query, the change causing the transition into or out of the matched state automatically triggers these functions when the change is performed by a jQuery method. But what about those times when we can't use jQuery for such changes?

9.3.3 Forcing Live Query evaluation

If we effect changes to elements that cause them to transition into or out of the matched state for Live Query listeners that we've established through means other than jQuery functions, we can force Live Query to trigger its listeners with a utility function.

Function syntax: \$.livequery.run**\$.livequery.run()**

Forces Live Query to perform a global evaluation of elements, triggering any appropriate listeners. This is useful when changes to elements are made outside the control of jQuery methods.

Parameters

none

Returns

Undefined.

We've seen that Live Query automatically removes event listeners when an element leaves the matched state and that we can establish a mismatch listener to do whatever we want on such mismatch transitions. But what do we do when we want to take a sledgehammer to the listeners?

9.3.4 Expiring Live Query listeners

In the same way that jQuery provides an `unbind()` command to undo the action of `bind()`, Live Query provides its own means to undo the establishment of Live Query event handlers and match/mismatch handlers—the `expire()` command, which sports many parameter formats.

Command syntax: expire**expire()****expire(event, listener)****expire(onmatch, onmismatch)**

Removes listeners associated with the selector of the match set. The format without parameters removes all listeners associated with the selector. See the description of the parameters below for how they affect this command.

Parameters

event (String) Specifies that event listeners for the event type be unbound. If no listener is specified, all listeners for the event type are removed.

listener (Function) When specified, only the specific listener is unbound from the selector (for the specified event type).

onmatch (Function) Specifies the match listener that's to be unbound from the selector.

mismatch (Function) If present, specifies the mismatch listener to be unbound from the selector.

Returns

The wrapped set.

We've provided a lab page to help illustrate the use of these techniques. Bring up the file `chapter9/livequery/lab.livequery.html` in your browser, and you'll see the display of figure 9.7.

This lab page displays three panes: the Control Panel with buttons that do interesting things, a Test Subjects container, and the Console, which displays messages to let us know what's going on.

The setup for this page bears some explanation. In the ready handler, the Control Panel buttons are instrumented to perform their respective actions once clicked (refer to the file for details if interested), and two Live Query statements are executed:

```

$( 'div.testSubject' ).livequery( 'click',
  function(){
    $(this).toggleClass( 'matched' );
  }
);
$( 'div.matched' ).livequery(
  function(){ reportMatch( this, true ); },
  function(){ reportMatch( this, false ); }
);

```

❶ Establishes a proactive click event handler

❷ Establishes match and mismatch handlers

The first of these statements establishes a Live Query event handler for all `<div>` elements with the class `testSubject` ❶. These elements include the single test

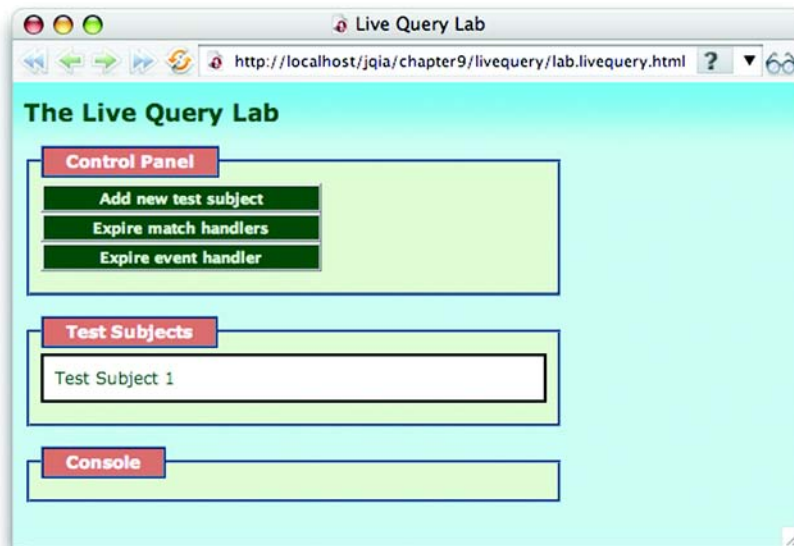


Figure 9.7 The Live Query Lab lets us observe Live Query in action.

subject already resident when the page is loaded (see figure 9.7), as well as all future test subject elements that will be created after clicking the Add New Test Subject button. Not only is the click event handler (whose activity we'll discuss in a minute) immediately established on the existing test subject element, but the click handler will *automatically* be added to any future test subjects dynamically added to the page (which will all be created as `<div>` elements with class `testSubject`).

The click handler causes the class `matched` to be toggled for the `target` of the event. To make it easy to see which test subjects have this class and which do not, we set up CSS rules so that elements without the class have a black border and white background, and elements *with* the class are rendered with a thicker maroon border and khaki background. We're all about trendy Web 2.0 colors!

The second of these statements establishes match and mismatch handlers for all `<div>` elements with the class `matched` ②. Each of these handlers issues a message to the Console, announcing that an element has become matched or mismatched respectively. Because no elements on the page possess the `matched` class at page load, the Console is empty when the page is initially displayed.

Now, let's see what the Control Panel buttons do:

- *Add New Test Subject*—This button adds a new test subject `<div>` element to the page. The element is created with an `id` of `testSubject#`, where `#` is a running count, and a class of `testSubject`. One such element is pre-populated in Test Subjects via HTML markup.
- *Expire Match Handlers*—This button executes the statement `$('.div.matched').expire();`, which causes the match and mismatch handlers we established in the ready handler to expire.
- *Expire Event Handler*—This button executes the statement `$('.div.testSubject').expire();`, which causes the proactive event handler we established in the ready handler to expire.

Now that we understand how the lab is wired, let's run through some exercises:

- *Exercise 1*—Load the page, and click within the bounds of Test Subject 1. Because the Live Query event handler for the click event that we established causes the `matched` class to be toggled (in this case added) to the element, we see that the element changes colors as displayed in figure 9.8. Additionally, because the addition of the `matched` class causes the element to match the selector we used to establish the match and mismatch handlers, we see that the match handler has fired and written a message to the Console.

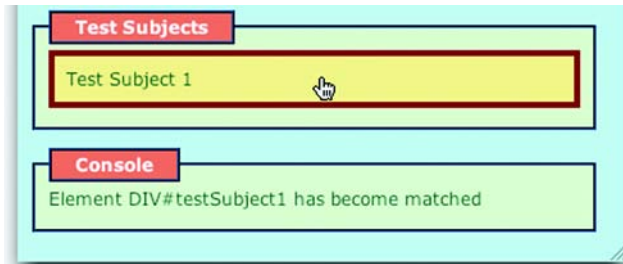


Figure 9.8
The addition of the matched class to the test subject triggers a change in rendition, as well as the established match handler.

- *Exercise 2*—Click Test Subject 1 again. This toggles the `matched` class, removing it from the element. The element returns to its original appearance, and the mismatch handler is triggered because the element no longer matches the selector, resulting in a Console message to that effect.
- *Exercise 3*—Click the Add New Test Subject button. A new test subject `<div>` element is added to the page. Because this element is created with the class of `testSubject`, it now matches the selector used to establish the proactive click handler that toggles the `matched` class. This triggers Live Query to *automatically* bind the click handler to this new element (the code to add this element doesn't bind any handlers to the newly created element). To test this supposition, click the newly created Test Subject 2. We see that it changes rendition and that a match handler has been called on its behalf, proving that the click handler for toggling the `matched` class of the element was automatically added to the newly created element. See the evidence in figure 9.9.

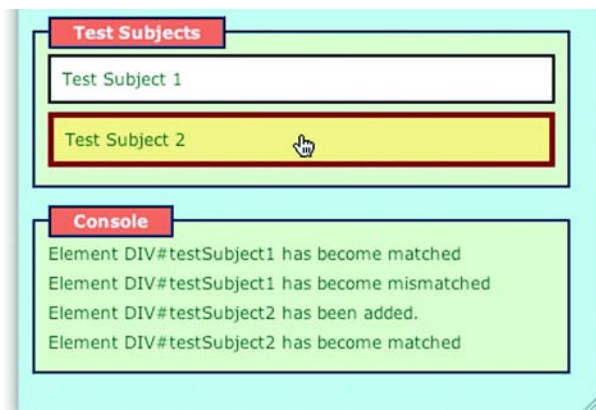


Figure 9.9
The fact that the newly added Test Subject 2 reacts to clicks in the same way as Test Subject 1 proves that Live Query has automatically added the click handler, as well as the match and mismatch handlers, to the newly created element.

- *Exercise 4*—Experiment with the Add New Test Subject button and Test Subjects until you’re convinced that the event, match, and mismatch handlers are always automatically added to the test subject elements whenever appropriate.
- *Exercise 5*—Play around with expiring Live Query handlers. Reload the page so that you start with fresh settings, and add a test subject or two to the page with the Add New Test Subject button. Now click the Expire Match Handlers button, which expires the match and mismatch handlers that were established on the Test Subjects. Notice that, when you click the Test Subjects, no match/mismatch messages appear in the Console, proving that the handlers have been removed. The click event handler still toggles the `matched` class because the element still changes appearance when clicked.
- *Exercise 6*—Click the Expire Event Handler button. This expires the Live Query click handler for the Test Subjects. Note that the Test Subjects are now unresponsive to mouse clicks and that they retain the state they had when you clicked the button.

It doesn’t take much imagination to see the power that this plugin brings to pages in Rich Internet Applications where elements are changing all the time, including popping into and out of existence. By allowing us to establish the behaviors of these events up front, the Live Query Plugin helps us minimize the amount of code we need to write when changing and adding elements to the page.

Now, let’s move on to one more important and useful plugin.

9.4 Introduction to the UI Plugin

When it comes to Rich Internet Applications, the UI is king. It’s not surprising that many jQuery plugins focus on enabling rich user interfaces. In this section, we’ll introduce the official UI Plugin, an important and recent addition to the jQuery family. Because it’s an important component, we’d love to cover this plugin to the same depth that we examined core jQuery; but reality intervenes, and practical space considerations prevent more extensive coverage.

We’ll extensively cover two of the essential methods defined by this plugin—the ones that provide support for drag-and-drop on our pages—giving you a good feel for how the remainder of the plugin operates. Then, we’ll provide an overview of the remainder of the plugin to demonstrate what it can do to bring

Rich Internet Application functionality to our pages. For more details regarding these areas, please visit <http://docs.jquery.com/ui>.

The UI Plugin provides three major areas of support: mouse interaction, widgets, and visual effects. The drag-and-drop operations fall under the category of mouse interaction; that's where we'll start.

9.4.1 *Mouse interactions*

Interacting with the mouse pointer is an integral and core part of any GUI. Although many simple mouse pointer interactions are built into web interfaces (clicking, for example), the web doesn't natively support some advanced interaction styles available to desktop applications. A prime example of this deficiency is the lack of support for *drag-and-drop*.

Drag-and-drop is an ubiquitous interaction technique for desktop user interfaces. For example, in the GUI file manager for any desktop system, we easily copy files or move them around the filesystem by dragging and dropping them from folder to folder or even delete them by dragging and dropping them onto a Trash or Wastebasket icon. But as prevalent as this interaction style is within desktop applications, it's as sparse in web applications, mainly because modern browsers don't natively support drag-and-drop. Correctly implementing it is a daunting task.

"Daunting?" you scoff. "A few captured mouse events and some CSS fiddling. What's the big deal?"

Although the high-level concepts aren't that difficult to grasp, it turns out that implementing the nuances of drag-and-drop support, particularly in a robust and browser-independent manner, can become painful quickly. But in the same way that jQuery and its plugins have eased our pain before, they do so again with support for web-enabled drag-and-drop.

But before we can drag *and* drop, we first need to learn how to drag.

Dragging things around

Although we'd be hard-pressed to find the term *draggable* in most dictionaries, it's the term that's commonly applied to items that can be dragged about in a drag-and-drop operation. Likewise, it's the term that the UI Plugin uses to describe such elements and as the name of the method that applies this ability to elements in a matched set.

Command syntax: `draggable`

`draggable(options)`

Makes the elements in the wrapped set draggable according to the specified options.

Parameters

`options` (Object) An object hash of the options to be applied to the draggable elements, as described in table 9.2. With no options specified, the elements are freely draggable anywhere within the window.

Returns

The wrapped set.

We must include at least the two following files to use the `draggable()` command (in addition to the core jQuery script file):

```
ui.mouse.js
ui.draggable.js
```

To obtain extended options, we also include the following:

```
ui.draggable.ext.js
```

The options, both basic and extended, supported by this method are shown in table 9.2.

Table 9.2 Basic and extended options for the `draggable()` command

Name	Description
Basic options	
<code>helper</code>	(String Function) Specifies exactly what is to be dragged. If specified as <i>original</i> (the default), the original item is dragged. If <i>clone</i> is specified, a copy of the element is created for dragging. You can also specify a function that accepts the original DOM element as its parameter and returns an element to be dragged. This is most often a clone of the original element with a transformation of some type applied—for example, <code>function(e){return \$(e).clone().css('color','green')}</code> .
<code>ghosting</code>	(Boolean) If <code>true</code> , a synonym for <code>helper: 'clone'</code> .
<code>handle</code>	(jQuery Element) An alternate element, or jQuery wrapped set containing an element, that serves as the drag handle—the item clicked on to initiate the drag operation. This element is frequently a child element of the draggable but can be any element on the page.

continued on next page

Table 9.2 Basic and extended options for the `draggable()` command (continued)

Name	Description
preventionDistance	(Number) The number of pixels that the mouse pointer needs to be moved after a click for a drag operation to commence. This can be used to help prevent accidental dragging. If omitted, the default is 0.
dragPrevention	(Array) An array of selectors of child elements that shouldn't initiate a drag operation when clicked. The default is ['input', 'textarea', 'button', 'select', 'option']. This is useful for preventing the start of drag operation when an embedded control is clicked, for example.
cursorAt	(Object) Specifies the spatial relationship between the mouse pointer and the draggable object while a drag operation is under way. The specified object can define properties of <code>top</code> , <code>left</code> , <code>bottom</code> or <code>right</code> . For example, an object of { <code>top:5</code> , <code>left:5</code> } causes the pointer to be positioned 5 pixels from the upper-left corner of the dragged element. If omitted, the pointer maintains its original relative position at the point of the mouse click that initiates the drag.
appendTo	(String Element) Specifies an element that the dragged helper is appended to at the end of the drag operation. Specifying the string <i>parent</i> (the default) leaves the helper in its original hierarchy.
start	(Function) A callback function called when a drag operation starts. Its function context (<code>this</code>) is set to the draggable element, and it's passed two parameters: the <code>event</code> instance in which the <code>target</code> property is set to the draggable element and an object containing the following properties: <ul style="list-style-type: none"> ■ <code>helper</code>—Current helper element ■ <code>position</code>—Object containing properties <code>top</code> and <code>left</code>, specifying the position of the mouse at the start of the drag operation ■ <code>offset</code>—Object specified for <code>cursorAt</code> ■ <code>draggable</code>—Internal JavaScript draggable object (not very useful) ■ <code>options</code>—Options hash used to create the draggable
stop	(Function) A callback function called when the drag operation completes. It's passed the same two parameters as the <code>start</code> callback. The <code>position</code> property of the second parameter reports the location of the upper-left corner of the draggable.
drag	(Function) A callback function continuously invoked while a drag operation is underway. It's passed the same two parameters as the <code>start</code> callback. The <code>position</code> property of the second parameter reports the location of the upper-left corner of the draggable.
Extended options	
axis	(String) Constrains the axis along which the draggable can be moved: <code>x</code> for the horizontal axis and <code>y</code> for the vertical axis. If omitted, no axis constraints are imposed.

continued on next page

Table 9.2 Basic and extended options for the `draggable()` command (continued)

Name	Description
<code>containment</code>	(String Object Element) Specifies the bounds within which the draggable can be moved. If omitted, no containment is imposed. Can be specified as the following: <ul style="list-style-type: none"> ▪ <i>parent</i>—Contains the draggable within its parent so that no scrollbars need be added to the parent ▪ <i>document</i>—Contains the draggable within the current document so that no scrollbars need be added to the window ▪ <i>a selector</i>—Identifies the containing element ▪ <i>an object</i>—Specifies a bounding rectangle relative to the parent with the properties <code>left</code>, <code>right</code>, <code>top</code>, and <code>bottom</code>
<code>effect</code>	(Array) An array of two strings that applies a fade effect to cloned helpers. Can be specified as <code>['fade', 'fade']</code> , <code>['fade', '']</code> , or <code>['', 'fade']</code> . This may seem like an odd way to specify the effects, but it allows for future support for additional effects types; currently, only <code>fade</code> is supported.
<code>grid</code>	(Array) An array of two numbers that specify a rectangular grid defining discrete locations that the draggable can be moved to—for example, <code>[100,100]</code> . The origin of the grid is relative to the original location of the draggable. If omitted, no grid constraint is placed on movement.
<code>opacity</code>	(Number) Specifies the opacity of the dragged helper during the drag operation as a value between 0.0 and 1.0 (inclusive). If omitted, the opacity of the dragged helper is unchanged.
<code>revert</code>	(Boolean) If <code>true</code> , the draggable is moved back to its original position when the drag operation concludes. If omitted or specified as <code>false</code> , the location isn't reverted.

If you thought we were going to pass up an opportunity for a fun lab page to demonstrate these options, guess again! But before we get to that, let's take a look at the other three methods related to draggables.

If we want to make a draggable element no longer draggable, the `draggableDestroy()` command removes its draggability.

Command syntax: `draggableDestroy`

`draggableDestroy()`

Removes draggability from the elements in the wrapped set

Parameters

`none`

Returns

The wrapped set

If all we want to do is to temporarily suspend an element's draggability and restore it at a later time, we disable draggability with `draggableDisable()` and restore it with `draggableEnable()`.

Command syntax: `draggableDisable`

`draggableDisable()`

Suspends draggability of the wrapped draggable elements without removing the draggability information or options

Parameters

none

Returns

The wrapped set

Command syntax: `draggableEnable`

`draggableEnable()`

Restores draggability to any draggables in the matched set that have been disabled via `draggableDisable()`

Parameters

none

Returns

The wrapped set

Through the use of the UI Draggables Lab, let's examine the draggable options. Bring up the page `chapter9/ui/lab.draggables.html` in your browser, and you'll see the display of figure 9.10.

This lab page sports a now-familiar layout: a Control Panel pane contains controls that allow us to specify the options for `draggable()`, a Test Subject pane contains an image element to serve as the draggable test subject, and a Console pane reports information about an ongoing drag operation.

When the Apply button (found in the Control Panel) is clicked, the specified options are collected and a `draggable()` command is issued. The format of the command is displayed below the Apply button. (For clarity, only the options specified via the Control Panel are displayed. The callbacks added to the options to effect the display in the Console pane aren't shown but are included in the command that's issued.) You'll observe the actions of the Disable and Enable buttons in exercise 3. The Reset button restores the option controls to initial conditions and destroys any draggable capability set on the test subject.

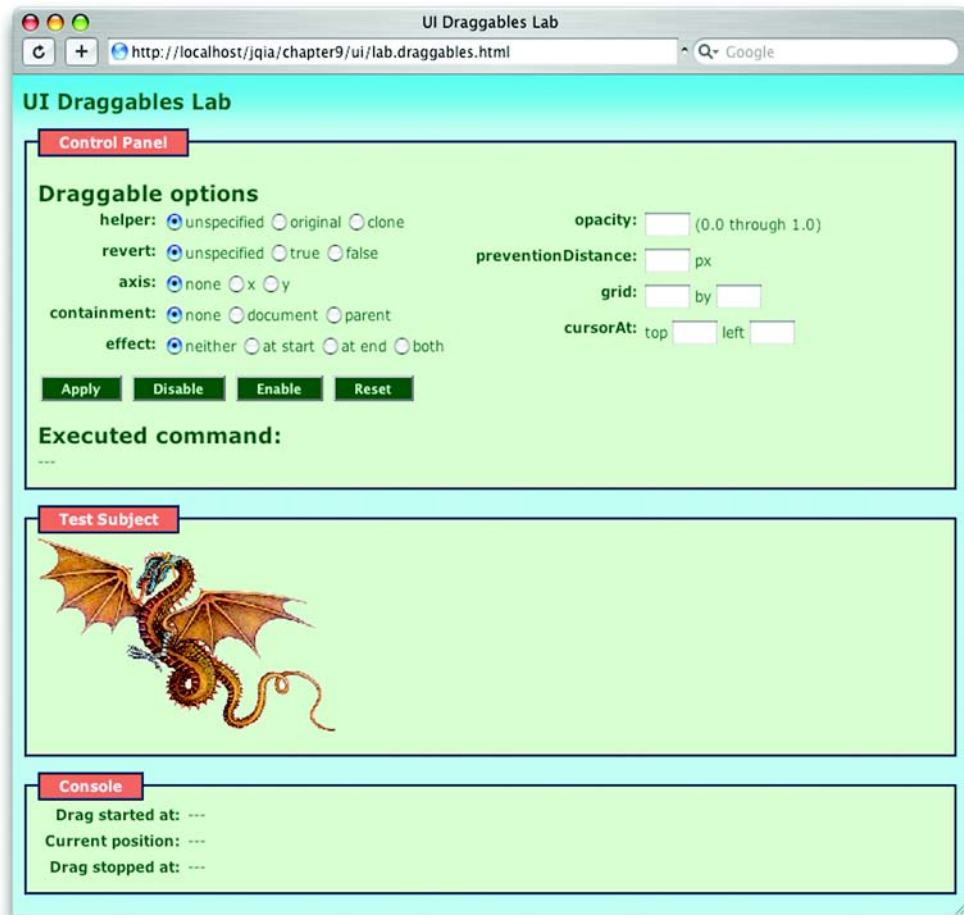


Figure 9.10 The UI Draggables Lab page allows us to play around with most of the options available for draggable items.

Let's dig into this lab and start some exercises!

Exercise 1—For the first exercise, we'll create a simple draggable with all options defaulted. Bring up the UI Draggables Lab page in your browser, and leave all option controls as they are upon load; they are set so that their initial conditions specify no options.

Try to click and drag the Test Subject dragon image. Nothing much happens unless you're using OS X, in which case you *will* see something happen. OS X allows you to drag images off of web pages to copy them to your local system via

drag-and-drop. Don't confuse this system-enabled drag-and-drop with the operations we're going to enable on our pages.

Now, click the Apply button, and observe that the executed command is displayed as follows:

```
$('#testSubject').draggable({});
```

Try dragging the dragon image again. Note how you can make the dragon fly around the browser window. (No flapping wings though—that'd be a more advanced animation!) Also, note how you can move the image to the edges of the window and, when moved to the extreme right, it causes a horizontal scrollbar to appear where none was before (in most browsers).

Drop the dragon, and pick him up again. Move him around as many times as you like. Observe how the values in the console are kept up to date during a drag operation. This is effected using the `start`, `stop`, and `drag` callbacks.

Exercise 2—Reload the page to restore initial conditions. If you're using Firefox or Camino, these browsers have an annoying feature: Form controls aren't restored to initial conditions when you reload a page using the Reload toolbar button. To cause the page to reload to initial conditions, move the text focus to the URL address field and hit the Enter key.

Now, set the `helper` option to `Original`, click the Apply button, and drag the Test Subject image around the window. You should notice no difference from the behavior of exercise 1 because `Original` is the default setting when `helper` is `Unspecified`. Change the value to `Clone`, and click Apply.

Now when you drag the image, you see a *copy* of the image being dragged about rather than the original element. Once you conclude the drag, the clone disappears.

Exercise 3—Leaving everything as you left it at the end of exercise 2, click the Disable button, and note the command that's issued:

```
$('#testSubject').draggableDisable();
```

Try to drag the dragon. Nothing happens. Now click the Enable button, which issues the following command:

```
$('#testSubject').draggableEnable();
```

Note that you can drag again *and* that the original options (in this case the clone helper) are still in effect, demonstrating the difference between `draggableDestroy()`, which removes drag capability completely, and `draggableDisable()`, which only suspends it until `draggableEnable()` is called.

Exercise 4—Reset the Lab, and choose `True` for the `revert` option, and click `Apply`. Drag the test subject, and note how it moves back to its original location when the operation concludes. Now, select `Clone` for the `helper`, click `Apply`, and repeat the exercise. Note that `revert` is applied to the clone.

Exercise 5—Reset the Lab, and experiment with the setting of the `axis` option. You can use this option to constrain the movement during a drag to the horizontal or vertical planes.

Exercise 6—In this exercise, we'll turn our attention to the `containment` option. You'll want to increase the height of your browser window to as much as your screen will allow; hopefully, your resolution will cause some extra space to be shown below the Console pane.

Up to this point, we've left `containment` unspecified. Recall how you were able to move the dragon anywhere within the browser window. Now choose `Document`, and click `Apply`. When you drag the image note two important things:

- You can no longer move the image beyond the edges of the window, so no scrollbars appear where there were previously none.
- You can't move the image below the bottom of the Console pane where the *document* stops but the window doesn't.

Now change the setting to `Parent`, and click `Apply`. When a drag operation is started, note how you can only drag the image around the inside of the Test Subject pane (a `<fieldset>` element), which is the parent of the test subject element. Note, also, that this is true even if the drag operation started outside the parent (as a result of a previous drag operation that was not constrained to the parent).

Exercise 7—Choose a `helper` option of `Clone`, and observe the effect of the `Effect` option on the helper in its various settings.

Exercise 8—Reset the Lab, and specify an `Opacity` of `0.5`. Observe how the opacity of the dragged element is affected whether the original element or a helper clone is being dragged.

Exercise 9—Reset the Lab, and set the `preventionDistance` option to a large value such as `200`. After clicking `Apply`, start a drag operation by clicking the edge of the dragon's left wing and moving the mouse pointer to the right. You traverse almost the entire width of the dragon (whose image width is `250` pixels) before the drag operation starts. It would be rare to set this option to such a large value, but we did so here for illustration of the behavior of the option. More often, this would be set to a much smaller value to prevent accidental movements of a few pixels from initiating an unintended drag operation.

Exercise 10—Reset the Lab, and specify `grid` values of 100 and 100. After clicking Apply, note how the image can now only be dragged in discrete movements of 100 pixels in either direction. Play around with other values to observe their behavior on the drag movement.

Exercise 11—Reset the Lab, and enter values of 10 and 10 for the `cursorAt` option. Click the Apply button. When a drag operation is started, the relative position of the image and mouse pointer is set so that the pointer is positioned 10 pixels in either direction, relative to the upper-left corner of the image (near the tip of the wing) no matter where the pointer was positioned within the image at the start of the drag.

Exercise 12—Go nuts! Play around with the settings of the lab, individually and in concert, until you feel confident that you understand how they each affect the drag operation of draggable elements.

Dragging things around the screen is all well and good, but is it really useful? It's fun for a time, but like playing with a yo-yo (unless we're true aficionados), it loses its charm quickly. In practical applications, we could use it to allow users to move modular elements around the screen (and if we're nice, we'd remember their chosen positions in cookies or other persistence mechanisms), or in games or puzzles.

Drag operations truly shine when there's something interesting to drop dragged elements on. Let's see how we can make *droppables* to go with our draggables.

Dropping dragged things

The flip side of the coin from draggables is *droppables*—elements that can accept dragged items and do something interesting when such an event occurs. Creating droppable items from page element is similar to creating draggables; in fact, it's even easier because there are fewer options to worry about.

And like draggables, droppables are split into two script files: the basic file that defines the `droppable()` command and its basic options and an option file that contains the extended options. These files are

```
ui.droppable.js  
ui.droppable.ext.js
```

The syntax for the `droppable()` command is as follows:

Command syntax: droppable

droppable(options)

Establishes the elements in the wrapped set as droppables, or elements on which draggables can be dropped.

Parameters

`options` (Object) The options applied to the droppable elements. See table 9.3 for details.

Returns

The wrapped set.

Once an element is instrumented as a droppable, it exists in one of three states: inactive, active, and armed.

Inactive state is the droppable's normal state where it stays most of the time, waiting to detect when a drag operation starts. When a drag starts, the droppable determines if the draggable is a suitable element for dropping (we'll discuss the concept of *suitability* in a moment) and, if so (and *only* if so), enters *active state*. In active state, the droppable monitors the drag operation, waiting until either the drag operation terminates, in which case the droppable returns to inactive state, or the draggable hovers over the droppable. When a suitable draggable element hovers over the droppable element, it enters *armed state*.

If the drag operation terminates while the droppable is in armed state, the draggable is considered to have been *dropped* onto the droppable. If the draggable continues to move so that it no longer hovers over the droppable, the droppable returns to active state.

Wow, that's a load of state changes to keep track of! The diagram of figure 9.11 should help you keep it all straightened out.

As with draggables, a basic set of options is available in the primary script file, and a set of extended options is available if the additional script file is included. Both sets of options are described in table 9.3.

When we looked at creating draggables, we saw that we could create a perfectly serviceable draggable without specifying any options to the `draggable()` method—not so with `droppable()`. Although nothing *bad* will happen if we don't specify any options when we make a call to `droppable()`, nothing good will happen either. A droppable created without an `accept` option does a pretty good impersonation of a brick.

By default, a droppable won't consider *any* draggable suitable. And a droppable that we can't drop anything on isn't useful, is it? To create a droppable that we

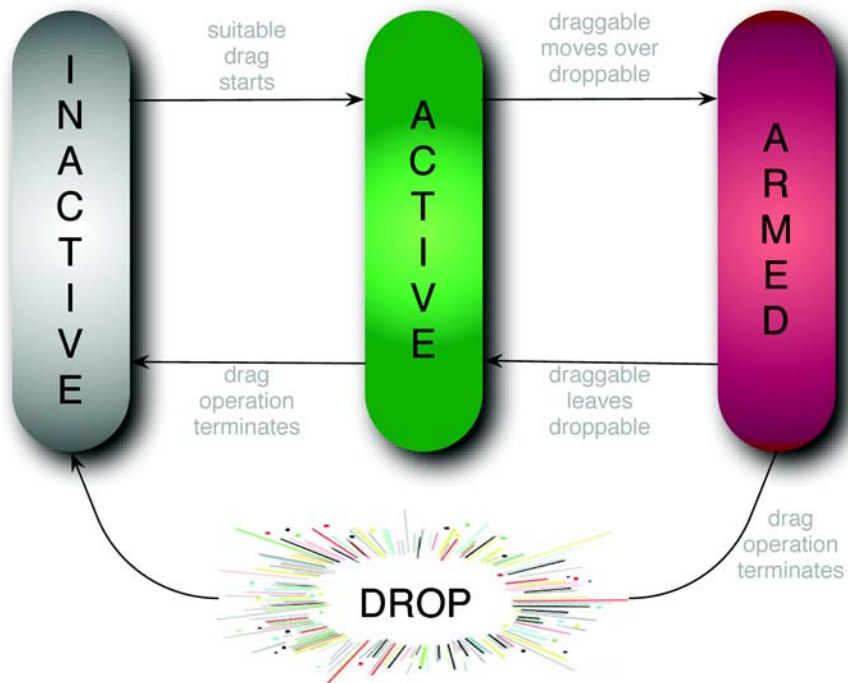


Figure 9.11 As a suitable draggable moves about the page, the droppable moves between its various states.

Table 9.3 Basic and extended options for the `droppable()` command

Name	Description
Basic options	
<code>accept</code>	(String Function) Specifies which draggables are suitable for dropping on the droppables. This can be a string describing a jQuery selector or a function that returns <code>true</code> to specify that a draggable is acceptable. When a function is specified, it's invoked with the candidate draggable passed as its only parameter.
<code>tolerance</code>	(String) A string value that defines how a draggable must be positioned in relation to the droppable in order to arm the droppable. The possible values are as follow: <ul style="list-style-type: none"> ■ <code>touch</code>—Arms the droppable if the draggable touches the droppable, or if any part of the draggable overlaps the droppable ■ <code>pointer</code>—Arms the droppable if the mouse pointer enters the droppable during a drag operation ■ <code>intersect</code>—Arms the droppable if 50% of the draggable intersects the droppable ■ <code>fit</code>—Arms the droppable if the draggable is completely contained within the droppable

continued on next page

Table 9.3 Basic and extended options for the `droppable()` command (continued)

Name	Description
<code>activate</code>	(Function) A callback function invoked when a drag operation for an acceptable draggable commences—when the droppable makes a transition from inactive to active state. The function context (<code>this</code>) is set to the droppable element. This function is passed the event instance and an object that contains information about the operation with the following properties: <ul style="list-style-type: none"> ■ <code>draggable</code>—The draggable instance ■ <code>droppable</code>—The droppable instance ■ <code>element</code>—The draggable element ■ <code>helper</code>—The draggable helper ■ <code>options</code>—The options passed to <code>droppable()</code>
<code>deactivate</code>	(Function) A callback function invoked when the droppable reverts to inactive state. This can be a transition from either active or armed state. The function context (<code>this</code>) is set to the droppable element, and this function is passed the same parameters as described for the <code>activate</code> callback.
<code>over</code>	(Function) A callback function invoked when the droppable makes a transition from active to armed state as a result of the draggable meeting the criteria defined by the <code>tolerance</code> option. The function context (<code>this</code>) is set to the droppable element, and this function is passed the same parameters as described for the <code>activate</code> callback.
<code>out</code>	(Function) A callback function invoked when the droppable makes a transition from armed to active state because of the draggable leaving the droppable as defined by the criteria specified by the <code>tolerance</code> option. The function context (<code>this</code>) is set to the droppable element, and this function is passed the same parameters as described for the <code>activate</code> callback.
<code>drop</code>	(Function) A callback function invoked when the draggable is dropped on the armed droppable. The function context (<code>this</code>) is set to the droppable element, and this function is passed the same parameters as described for the <code>activate</code> callback.
Extended options	
<code>activeClass</code>	(String) A CSS class name applied to the droppable when it's in active state.
<code>hoverClass</code>	(String) A CSS class name applied to the droppable when a suitable draggable is hovering over it—when the droppable is in armed state.

can drop something on, we need to specify an `accept` option that defines the draggables that the droppable should consider *suitable*.

Because nothing drives home concepts like playing with them yourself, we created a Draggables Lab. Bring up the file of `chapter9/ui/lab.droppables.html`, and you'll see the display of figure 9.12.

Similar to the other labs, there's a Control Panel that lets us specify the options to be applied to the droppable after clicking the Apply button. The Disable,

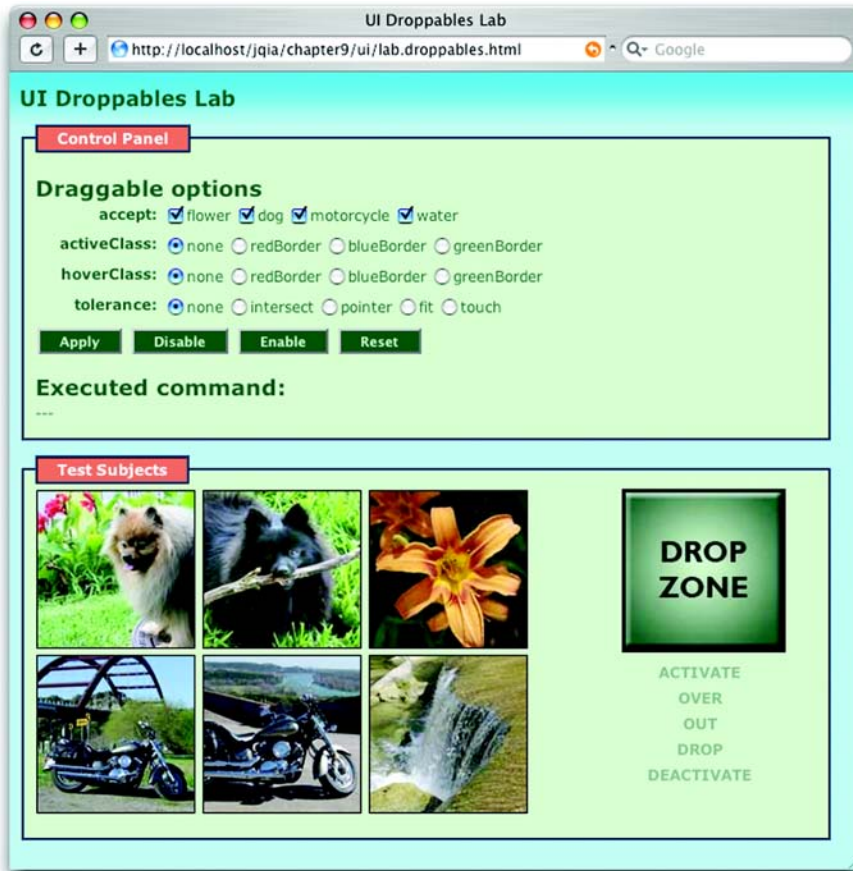


Figure 9.12 The UI Droppables Lab lets us examine the effect of the various options on drag-and-drop operations.

Enable, and Reset buttons serve functions similar to their counterparts in the Draggables Lab.

In the Test Subjects Pane are six draggable elements and an element (which we'll call the *Drop Zone*) that will become droppable after clicking the Apply button. Below the Drop Zone are grayed-out text elements that read Activate, Over, Out, Drop, and Deactivate. When a corresponding droppable callback (added to the droppable behind the scenes by the lab) is invoked, the appropriate text element, which we'll refer to as a *callback indicator*, is momentarily highlighted to indicate that the callback has fired.

Let's dig in and get the skinny on droppables using this lab.

Exercise 1—In this exercise, we're going to begin familiarizing ourselves with the `Accept` option, which is what tells the droppable what constitutes a *suitable* (or acceptable) draggable. Although this option can be set to any jQuery selector (or even a function that can programmatically make suitability determinations), for the purpose of the lab, we'll concentrate on elements that possess particular class names. In particular, we can specify a selector that includes any of the class names `flower`, `dog`, `motorcycle`, and `water` by checking the appropriate check boxes of the `Accept` option controls.

The six draggable image elements on the left side of the Test Subject pane are each assigned one or two of these class names based on what appears in the image. For example, the upper-left draggable possesses the class names `dog` and `flower` (because both a dog and some flowers appear in the photo), whereas the lower-middle image is defined with the class names `motorcycle` and `water` (a Yamaha V-Star and the Colorado River, to be precise).

Before clicking `Apply`, try to drag and drop any of these elements on the Drop Zone. Aside from the dragging, not much happens. Carefully observe the callback indicators, and note how they don't change. This should be no surprise because, at outset, no droppable even exists on the page.

Now, leaving all controls in their initial conditions (including all `accept` check boxes checked), click the `Apply` button. The executed command includes an `accept` option that specifies a selector that matches all four class names.

Once again, try to drag any of the images to the Drop Zone while observing the callback indicators. This time, you'll see the `Activate` indicator briefly highlight, or *throb*, when you begin moving any of the images, indicating that the droppable has noticed that a drag operation has commenced and that the draggable is acceptable for dropping.

Drag the image over and out of the Drop Zone a number of times. The *over* and *out* callbacks are invoked (as shown by the corresponding indicators) at the appropriate times. Now, drop the image outside the confines of the Drop Zone, and watch the `Deactivate` indicator throb.

Finally, repeat the drag operation except, this time, drop the image on top of the Drop Zone. The Drop indicator throbs (indicating that the drop callback was invoked). Note, also, that the Drop Zone is wired to display the most recent image that was dropped upon it.

Exercise 2—Uncheck all of the `accept` check boxes, and click `Apply`. No matter which image you choose, no callback indicators throb, and nothing happens

when you drop an image on the Drop Zone. Without a meaningful `accept` option, our Drop Zone has become a brick.

Exercise 3—Try checking at least one `accept` check box, say `Flower`, and note how only images with flowers in them (known to the page as such because the class name `flower` was defined for them) are construed to be acceptable items.

Try again with whatever combinations of acceptable class names you like until you're comfortable with the concept of the `accept` option.

Exercise 4—Reset the controls, check the `activeClass` radio button `green-border`, and click `Apply`. This supplies an `activeClass` option to the droppable that specifies a class name that defines (you guessed it) a green border.

Now, when you begin to drag an image that's acceptable to the droppable (as defined by the `accept` option), the black border around the Drop Zone is replaced by a green border. If you have trouble getting this to work for you on your own pages, note that you need to be mindful of CSS precedence rules. When an `activeClass` class name is applied, it must be able to override the rule that assigns the default visual rendition that you wish to supplant. This is also true of `hoverClass`.

Exercise 5—Check the `hoverClass` radio button labeled `RedBorder`, and click `Apply`. When an acceptable image is dragged over the Drop Zone, the border changes from green (as a result of the `activeClass` setting) to red.

Experiment with these two option settings until you're comfortable with the timing in the drag operation where each option triggers a class change.

Exercise 6—For this exercise, choose each of the various `tolerance` radio buttons, and note how the setting affects when the droppable makes the transition from active to armed state (as described by the definition of this option in table 9.3). This transition can easily be observed by setting the `hoverClass` option or when the `Over` callback indicator throbs.

Like their counterpart draggables, droppables can be destroyed, temporarily disabled, and then re-enabled. The methods to do so are as follow:

Command syntax: `droppableDestroy`

`droppableDestroy()`

Removes the ability to serve as a droppable from all elements in the wrapped set

Parameters

none

Returns

The wrapped set

Command syntax: droppableDisable**droppableDisable()**

Suspends droppability of the wrapped droppable elements without removing the droppability information or options

Parameters

none

Returns

The wrapped set

Command syntax: droppableEnable**droppableEnable()**

Restores droppability to any droppables in the matched set that had been disabled via `droppableDisable()`

Parameters

none

Returns

The wrapped set

Drag-and-drop is a useful interaction technique for many purposes. It's frequently used to indicate association but may also be used to rearrange the order of elements. This latter use is common enough to warrant direct support by the UI Plugin. Let's look into that.

Other mouse interaction methods

The remaining category of commands in the mouse interaction category of the UI Plugin are *sortable*s, *selectable*s and *resizable*s. The methods in these categories leverage the drag-and-drop capability to allow elements to be reordered within a container and resized, respectively.

As with the UI Plugin commands we've examined so far, each of these operates by applying methods to a matched set and passing in an object hash that specifies the options. For complete details, please refer to <http://docs.jquery.com/UI>.

In addition to dynamic drag-and-drop and other mouse interaction operations, the UI Plugin also provides a number of widgets that we can use to extend the basic set of user interface elements that are natively provided by HTML.

9.4.2 UI widgets and visual effects

Although we'd love to go into great detail regarding the extensive set of user interface widgets that the UI Plugin provides, reality intervenes. In lieu of any deep discussion, we'll at least give you a taste of what's available, so you'll know what to go look up in the online documentation.

The following lists the available widgets with short descriptions. Visit <http://docs.jquery.com/ui> for more details.

Accordion—A simple widget that creates expanding and contracting levels out of simple markup, such as lists or nested `<div>` elements.

Tabs—A widget for creating tabbed sets with a fairly interesting and potentially complicated set of options. Anchor elements are used to specify the *tabs*, and their `href` attributes are used to identify the tabbed sections (using page-internal hash references). This widget accounts for usage of the back button and can be configured to open a specific tab on page load.

Calendar—A widget that provides a dynamic date picker, which is associated with an input element. This widget is highly configurable and can appear on-page or within a dialog box.

Dialog—A modal dialog box widget with move and resize capability.

Slider—A control that creates a *slider* (similar to that available for desktop applications) that can be integrated into a form via hidden elements. The control can be oriented in various ways, and its minimum and maximum values are completely configurable.

Table—A sortable table widget that's considered robust and fast.

In addition to these widgets, the following visual effects are provided:

Shadow—Generates a drop shadow for specified elements.

Magnifier—Causes the contents of elements to enlarge (magnify) upon mouse proximity.

With these UI elements at our disposal, we gain many options for creating great Rich Internet Applications. As they say, "But wait! There's more!" The jQuery community is willing to share their enhancements to jQuery. Visit <http://jquery.com/plugins> for some insight into the many—and we do mean *many*—plugins that have been published by other jQuery users.

9.5 Summary

From the outset, jQuery's creators designed it to provide an easy, but robust, plugin architecture. The idea was that the core jQuery download would stay small

and limber, providing only those core features required by most web application authors and leaving the rest to plugins that could be included as needed. That strategy has served jQuery well as the community of its users has created and published a vast array of plugins that any user of jQuery can download and employ.

Taking a survey of some of the more often-used plugins, we saw a wide range of functionality that enhances the core jQuery feature set.

The Form Plugin provides wrapped methods that allow us to deal with form elements in a controlled fashion, even allowing us to instrument forms for easy submission through Ajax requests rather than the traditional full-page refresh.

Getting accurate (or even quick-and-dirty estimates) of the position and dimensions of DOM elements is provided by the Dimensions Plugin, which is essential when trying to accurately place elements on the page in relation to one another or to the page origin.

Another plugin that's indispensable is the Live Query Plugin, which allows us to register event handlers for elements that don't even exist yet. This seemingly non-causal ability is an enormous advantage for pages in which we expect DOM elements to be frequently created and destroyed over the lifetime of the page.

The UI Plugin, which we unfortunately weren't able to fully explore, provides such essential user interface capabilities as drag-and-drop, sorting, and a handful of useful user interface widgets.

And that's just the beginning. Drop in on <http://jquery.com/plugins> for a list of other available plugins. More and more get added all the time!

9.6 The end?

Hardly!

Even though we've presented the entire jQuery API within the confines of this book, it would have been impossible to show you all the many ways that this broad API can be used on your pages. The examples that we presented were chosen specifically to lead you down the path of discovering how you can use jQuery to solve the problems that you encounter on a day-to-day basis on your web application pages.

jQuery is a living project. Heck, it was quite a chore for your authors to keep up with the rapid developments in the library over the course of writing this book. The core library is constantly evolving into a more useful resource, and more and more plugins are appearing on practically a daily basis.

We urge you to keep track of the development in the jQuery community and sincerely hope that this book has been a great help in starting you on writing

better Rich Internet Applications in less time and with less code than you might have ever believed possible.

We wish you health and happiness, and may all your bugs be easily solvable!

appendix: JavaScript that you need to know but might not!

This appendix covers

- Which JavaScript concepts are important for effectively using jQuery
- JavaScript Object basics
- How functions are *first-class objects*
- Determining (and controlling) what *this* means
- What's a closure?

One of the great benefits that jQuery brings to our web applications is the ability to implement a great deal of scripting-enabled behavior without having to write a whole lot of script ourselves. jQuery handles the nuts-and-bolts details so that we can concentrate on the job of making our applications do what they need to do!

For the first few chapters in this book, we only needed rudimentary JavaScript skills to code and understand those examples. With the chapters on advanced topics such as event handling, animations, and Ajax, we must understand a handful of fundamental JavaScript concepts to make effective use of the jQuery library. You may find that a lot of things that you, perhaps, took for granted in JavaScript (or took on blind faith) will start to make more sense.

We're not going to go into an exhaustive study of all JavaScript concepts—that's not the purpose of this book. The purpose of this book is to get us up and running with effective jQuery in the shortest time possible. To that end, we'll concentrate on the fundamental concepts that we need to make the most effective use of jQuery in our web applications.

The most important of these concepts centers around the manner in which JavaScript defines and deals with functions, specifically the way in which functions are *first-class objects* in JavaScript. What do we mean by that? Well, in order to understand what it means for a function to be an object, let alone a *first-class* one, we must first make sure that we understand what a JavaScript object itself is all about. So let's dive right in.

A.1 JavaScript Object fundamentals

The majority of object-oriented (OO) languages define a fundamental `Object` type of some kind from which all other objects are derived. Likewise, in JavaScript, the fundamental `Object` serves as the basis for all other objects, but that's where the comparison stops. At its basic level, the JavaScript `Object` has little in common with the fundamental object defined by its OO brethren languages.

At first glance, a JavaScript `Object` may seem like a boring and mundane item. Once created, it holds no data and exposes little in the way of semantics. But those limited semantics *do* give it a great deal of potential.

Let's see how.

A.1.1 How objects come to be

A new object comes into existence via the `new` operator paired with the `Object` constructor. Creating an object is as easy as

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

It could be even easier (as we'll see shortly), but this will do for now.

But what can we *do* with this new object? It seemingly contains nothing: no information, no complex semantics, nothing. Our brand-new, shiny object doesn't get interesting until we start adding things to it—things known as *properties*.

A.1.2 Properties of objects

Like their server-side counterparts, JavaScript objects can contain data and possess methods (well...sort of, but that's getting ahead of ourselves). Unlike those server-side brethren, these elements aren't pre-declared for an object; we create them dynamically as needed.

Take a look at the following code fragment:

```
var ride = new Object();
ride.make = 'Yamaha';
ride.model = 'V-Star Silverado 1100';
ride.year = 2005;
ride.purchased = new Date(2005,3,12);
```

We create a new `Object` instance and assign it to a variable named `ride`. We then populate this variable with a number of *properties* of different types: two strings, a number, and an instance of the `Date` type.

We don't need to declare these properties prior to assigning them; they come into being merely by the act of our assigning a value to them. That's mighty powerful juju that gives us a great deal of flexibility. But before we get too giddy, let's remember that flexibility always comes with a price!

For example, let's say that in a subsequent part of the page, we want to change the value of the purchase date:

```
ride.purchased = new Date(2005,2,1);
```

No problem...unless we make an inadvertent typo such as

```
ride.purcahsed = new Date(2005,2,1);
```

There's no compiler to warn us that we've made a mistake; a new property named `purcahsed` is cheerfully created on our behalf, leaving us to wonder later on why the new date didn't *take* when we reference the correctly spelled property.

With great power comes great responsibility (where have we heard that before?), so type carefully!

NOTE JavaScript debuggers such as Firebug for Firefox can be lifesavers when dealing with such issues. Because typos such as these frequently result in no JavaScript errors, relying on JavaScript consoles or error dialog boxes is usually less than effective.

From this example, we've learned that an instance of the JavaScript `Object`, which we'll simply refer to as an *object* from here forward, is a collection of *properties*, each of which consists of a *name* and a *value*. The name of a property is a string, and the value can be any JavaScript object, be it a `Number`, `String`, `Date`, `Array`, basic `Object`, or any other JavaScript object type (including, as we shall see, functions).

This makes the primary purpose of an `Object` instance to serve as a container for a named collection of other objects. This may remind you of concepts in other languages: a Java map for example, or dictionaries or hashes in other languages.

When referencing properties, we can chain references to properties of objects serving as the properties of a parent object. Let's say that we add a new property to our `ride` instance that captures the owner of the vehicle. This property is another JavaScript object that contains properties such as the name and occupation of the owner:

```
var owner = new Object();
owner.name = 'Spike Spiegel';
owner.occupation = 'bounty hunter';
ride.owner = owner;
```

To access the nested property, we write

```
var ownerName = ride.owner.name;
```

There are no limits to the nesting levels we can employ (except the limits of good sense). When finished—up to this point—our object hierarchy looks as shown in figure A.1.

Note how each value is a distinct instance of a JavaScript type.

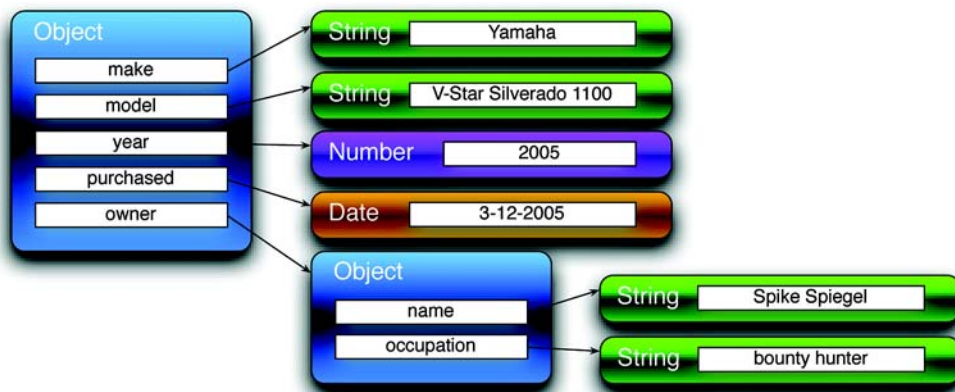


Figure A.1 Our object hierarchy shows that `Objects` are containers for named references to other `Objects` or JavaScript built-in objects.

By the way, there's no need for all the intermediary variables (such as `owner`) we've created for illustrative purposes in these code fragments. In a short while, we'll see more efficient and compact ways to declare objects and their properties.

Up to this point, we've referenced properties of an object by using the dot (period character) operator; but, as it turns out, that's a synonym for a more general operator for performing property referencing.

What if, for example, we have a property named `color.scheme`? Do you notice the period in the middle of the name? It throws a monkey wrench into the works because the JavaScript interpreter will try to look up `scheme` as a nested property of `color`.

"Well, just don't do that!" you say. But what about space characters? What about other characters that could be mistaken for delimiters rather than as part of a name? And most importantly, what if we don't even know what the property name is but have it as a value in another variable or as the result of an expression evaluation?

For all these cases the dot operator is inadequate, and we must use the more general notation for accessing properties. The format of the general property reference operator is

```
object[propertyNameExpression]
```

where *propertyNameExpression* is a JavaScript expression whose evaluation as a string forms the name of the property to be referenced. For example, all three of the following references are equivalent:

```
ride.make  
ride['make']  
ride['m'+ 'a'+ 'k'+ 'e']
```

as well as

```
var p = 'make';  
ride[p];
```

Using the general reference operator is the only way to reference properties whose names don't form valid JavaScript identifiers, such as

```
ride["a property name that's rather odd!"]
```

which contains characters not legal for JavaScript identifiers, or whose names are the values of other variables.

Building up objects by creating new instances with the `new` operator and assigning each property using separate assignment statements can be a tedious affair. In the next section, we'll look at a more compact and easy-to-read notation for declaring our objects and their properties.

A.1.3 Object literals

In the previous section, we created an object that modeled some of the properties of a motorcycle, assigning it to a variable named `ride`. To do so, we used two new operations, an intermediary variable named `owner`, and a bunch of assignment statements. This is tedious—as well as wordy and error-prone—and makes it difficult for us to visually grasp the structure of the object during a quick inspection of the code.

Luckily, we can use a notation that's more compact and easier to visually scan. Consider the following statement:

```
var ride = {
  make: 'Yamaha',
  model: 'V-Star Silverado 1100',
  year: 2005,
  purchased: new Date(2005,3,12),
  owner: {
    name: 'Spike Spiegel',
    occupation: 'bounty hunter'
  }
};
```

Using an *object literal*, this fragment creates the same `ride` object that we built up with assignment statements in the previous section.

This notation, which has come to be termed *JSON* (JavaScript Object Notation¹), is much preferred by most page authors over the multiple-assignment means of object building. Its structure is simple; an object is denoted by a matching pair of braces, within which properties are listed delimited by commas. Each property is denoted by listing its name and value separated by a colon character.

NOTE Technically, JSON has no means to express date values, primarily because JavaScript itself lacks any kind of date literal. When used in script, the `Date` constructor is usually employed as shown in the previous example. When used as an interchange format, dates are frequently expressed either as a string containing the ISO 8601 format or a number expressing the date as the millisecond value returned by `Date.getTime()`.

As we can see by the declaration of the `owner` property, object declarations can be nested.

¹ For more information, you can visit <http://www.json.org/>.

By the way, we can also express arrays in JSON by placing the comma-delimited list of elements within square brackets as in the following:

```
var someValues = [2,3,5,7,11,13,17,19,23,29,31,37];
```

As we've seen in the examples presented in this section, object reference are frequently stored in variables or in properties of other objects. Let's take a look at a special case of the latter scenario.

A.1.4 Objects as window properties

Up to this point, we've seen two ways to store a reference to a JavaScript object: variables and properties. These two means of storing references use differing notation, as shown in the following snippet:

```
var aVariable =  
  'Before I teamed up with you, I led quite a normal life.';  
  
someObject.aProperty =  
  'You move that line as you see fit for yourself.';
```

These two statements each assign a `String` instance (created via literals) to a variable and an object property respectively using assignment operations. (Kudos to you if you can identify the source of the obscure quotes; no cheating with Google! There was a clue earlier in the chapter.)

But are these statements *really* performing different operations? As it turns out, they're not!

When the `var` keyword is used at the top level, outside the body of any containing function, it's only a programmer-friendly notation for referencing a property of the pre-defined JavaScript window object. Any reference made in top-level scope is implicitly made on the window instance.

This means that all of the following statements are equivalent:

```
var foo = bar;
```

and

```
window.foo = bar;
```

and

```
foo = bar;
```

Regardless of which notation is used, a window property named `foo` is created (if it's not already in existence) and assigned the value of `bar`. Also, note that because `bar` is unqualified, it's assumed to be a property on `window`.

It probably won't get us into conceptual trouble to think of top-level scope as *window* scope because any unqualified references *at the top level* are assumed to be window properties. The scoping rules get more complex when we delve deeper into the bodies of functions—much more complex, in fact—but we'll be addressing that soon enough.

That pretty much covers things for our overview of the JavaScript Object. The important concepts to take away from this discussion are

- A JavaScript object is an unordered collection of properties.
- Properties consist of a name and a value.
- Objects can be declared using object literals.
- Top-level *variables* are properties of `window`.

Now, let's discuss what we meant when we referred to JavaScript functions as *first-class objects*.

A.2 Functions as first-class citizens

In many traditional OO languages, objects can contain data, and they can possess methods. In these languages the data and the methods are usually distinct concepts; JavaScript walks a different path.

Functions in JavaScript are considered objects like any of the other object types that are defined in JavaScript, such as `Strings`, `Numbers`, or `Dates`. Like other objects, functions are defined by a JavaScript constructor—in this case `Function`—and can be

- Assigned to variables
- Assigned as a property of an object
- Passed as a parameter
- Returned as a function result
- Created using literals

Because functions are treated in the same way as other objects in the language, we say that functions are *first-class objects*.

But you might be thinking to yourself that functions are fundamentally different from other object types like `String` or `Number` because they possess not only a value (in the case of a `Function` instance, its body) but also a *name*.

Well, not so fast!

A.2.1 What's in a name?

A large percentage of JavaScript programmers operate under a false assumption that functions are named entities. Not so. If you're one of these programmers, you've been fooled by a Jedi mind trick. As with other instances of objects—be they Strings, Dates, or Numbers—functions are referenced *only* when they are assigned to variables, properties, or parameters.

Let's consider objects of type `Number`. We frequently express instances of `Number` by their literal notation such as `213`. The statement

```
213;
```

is perfectly valid, but it is also perfectly useless. The `Number` instance isn't all that useful unless it has been assigned to a property or a variable, or bound to a parameter name. Otherwise, we have no way to reference the disembodied instance.

The same applies to instances of `Function` objects.

“But, but, but...” you might be saying, “what about the following code?”

```
function doSomethingWonderful() {  
  alert('does something wonderful');  
}
```

“Doesn't that create a function *named* `doSomethingWonderful`?”

No, it doesn't. Although that notation may seem familiar and is ubiquitously used to create *top-level* functions, it's the same syntactic sugar used by `var` to create window properties. The `function` keyword automatically creates a `Function` instance and assigns it to a window property created using the function “name” (what we referred to earlier as a *Jedi mind trick*) as in the following:

```
doSomethingWonderful = function() {  
  alert('does something wonderful');  
}
```

If that looks weird to you, consider another statement using the exact same format, except this time using a `Number` literal:

```
aWonderfulNumber = 213;
```

There's nothing strange about that, and the statement assigning a function to a top-level variable (a.k.a. window property) is no different; a function literal is used to create an instance of `Function` and then is assigned to the variable `doSomethingWonderful` in the same way that our `Number` literal `213` was used to assign a `Number` instance to the variable `aWonderfulNumber`.

If you've never seen the syntax for a *function literal*, it might seem odd. It's composed of the keyword `function`, followed by its parameter list enclosed in parentheses, then followed by the function body.

When we declare a top-level named function, a `Function` instance is created and *assigned* to a property of `window` that's automatically created using the so-called function name. The `Function` instance itself no more has a name than a `Number` literal or a `String` literal. Figure A.2 illustrates this concept.

Gecko browsers and function names

Browsers based on the Gecko layout engine, such as Firefox and Camino, store the name of functions defined using the top-level syntax in a nonstandard property of the function instance named `name`. Although this may not be of much use to the general development public, particularly considering its confinement to Gecko-based browsers, it's of great value to writers of browser plugins and debuggers.

Remember that, when a top-level variable is created in an HTML page, the variable is created as a property of the `window` instance. Therefore, the following statements are all equivalent:

```
function hello(){ alert('Hi there!'); }  
hello = function(){ alert('Hi there!'); }  
window.hello = function(){ alert('Hi there!'); }
```

Although this may seem like syntactic juggling, it's important to understanding that `Function` instances are *values* that can be assigned to variables, properties, or parameters just like instances of other object types. And like those other object types, nameless disembodied instances aren't of any use unless they're assigned to a variable, property, or parameter through which they can be referenced.

We've seen examples of assigning functions to variables and properties, but what about passing functions as parameters? Let's take a look at why and how we do that.

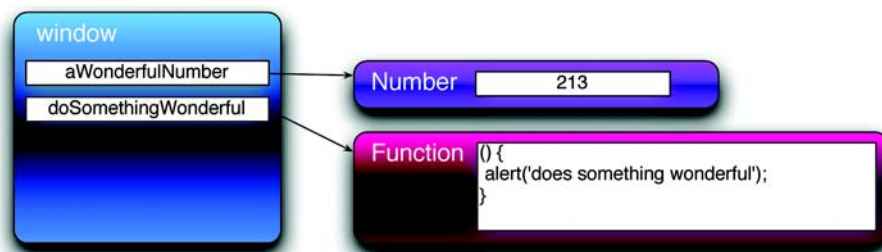


Figure A.2 A `Function` instance is a nameless object like the `Number 213` or any other JavaScript value. It's named only by references that are made to it.

A.2.2 Functions as callbacks

Top-level functions are all well and good when our code follows a nice and orderly synchronous flow, but the nature of HTML pages—once loaded—is far from synchronous. Whether we are handling events, instituting timers, or making Ajax requests, the nature of the code in a web page is asynchronous. And one of the most prevalent concepts in asynchronous programming is the notion of a *callback* function.

Let's take the example of a timer. We can cause a timer to fire—let's say in five seconds—by passing the appropriate duration value to the `window.setTimeout()` method. But how does that method let us know when the timer has expired so that we can do whatever it is that we're waiting around for? It does so by invoking a function that *we* supply.

Let's consider the following code:

```
function hello() { alert('Hi there!'); }

setTimeout(hello,5000);
```

We declare a function named `hello` and set a timer to fire in 5 seconds, expressed as 5000 milliseconds by the second parameter. In the first parameter to the `setTimeout()` method, we pass a function reference. Passing a function as a parameter is no different than passing any other value—just as we passed a `Number` in the second parameter.

When the timer expires, the `hello` function is called. Because the `setTimeout()` method makes a call *back* to a function in our own code, that function is termed a *callback* function.

This code example would be considered naïve by most advanced JavaScript coders because the creation of the `hello` name is unnecessary. Unless the function is to be called elsewhere in the page, there's no need to create the `window` property `hello` to momentarily store the `Function` instance to pass it as the callback parameter.

The more elegant way to code this fragment is

```
setTimeout(function() { alert('Hi there!'); },5000);
```

in which we express the function literal directly in the parameter list, and no needless name is generated. This is an idiom that we'll often see used in jQuery code when there is no need for a function instance to be assigned to a top-level property.

The functions we've created in the examples so far are either top-level functions (which we know are top-level `window` properties) or assigned to parameters

in a function call. We can also assign `Function` instances to properties of objects, and that's where things get really interesting. Read on...

A.2.3 What's this all about?

OO languages automatically provide a means to reference the current instance of an object from within a method. In languages like Java and C++, a variable named `this` points to that current instance. In JavaScript, a similar concept exists and even uses the same `this` keyword, which also provides access to an object associated with a function. But OO programmers beware! The JavaScript implementation of `this` differs from its OO counterparts in subtle but significant ways.

In class-based OO languages, the `this` pointer generally references the instance of the class within which the method has been declared. In JavaScript, where functions are first-class objects that aren't declared as part of anything, the object referenced by `this`—termed the *function context*—is determined not by how the function is declared but by how it's *invoked*.

This means that the *same* function can have *different* contexts depending on how it's called. That may seem freaky at first, but it can be quite useful.

In the default case, the context (`this`) of an invocation of the function is the object whose property contains the reference used to invoke the function. Let's look back to our motorcycle example for a demonstration, amending the object creation as follows (additions highlighted in bold):

```
var ride = {
  make: 'Yamaha',
  model: 'V-Star Silverado 1100',
  year: 2005,
  purchased: new Date(2005,3,12),
  owner: {name: 'Spike Spiegel', occupation: 'bounty hunter'},
  whatAmI: function() {
    return this.year+' '+this.make+' '+this.model;
  }
};
```

To our original example code, we add a property named `whatAmI` that references a `Function` instance. Our new object hierarchy, with the `Function` instance assigned to the property named `whatAmI`, is shown in figure A.3.

When the function is invoked through the property reference as in

```
var bike = ride.whatAmI();
```

the function context (the `this` reference) is set to the object instance pointed to by `ride`. As a result, the variable `bike` gets set to the string *2005 Yamaha V-Star*

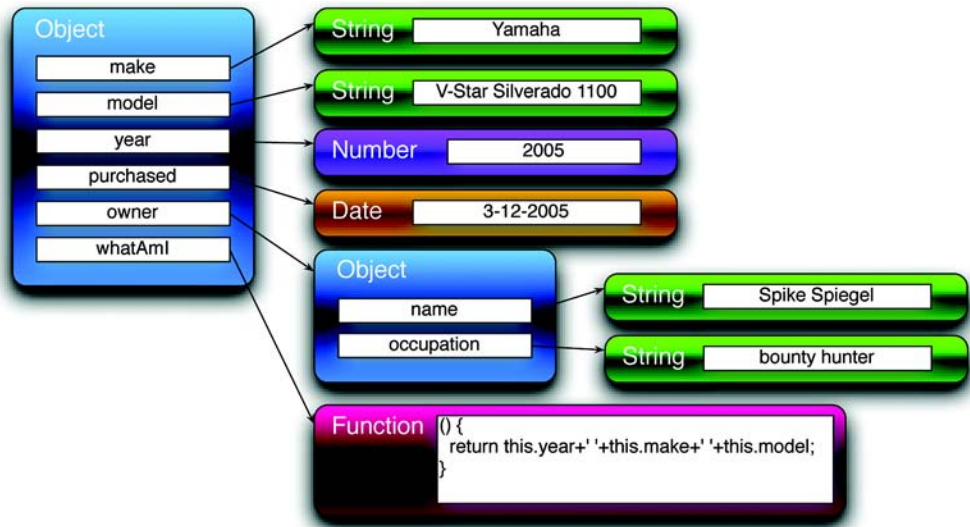


Figure A.3 This model clearly shows that the function isn't part of the Object but is only referenced from the Object property named `whatAmI`.

Silverado 1100 because the function picks up the properties of the object through which it was invoked via `this`.

The same is true of top-level functions. Remember that top-level functions are properties of `window`, so their function contexts, when called *as* top-level functions, are the `window` objects.

Although that may be the usual and implicit behavior, JavaScript gives us the means to explicitly control what's used as the function context. We can set the function context to whatever we want by invoking a function via the `Function` methods `call()` or `apply()`.

Yes, as first-class objects, even functions have methods as defined by the `Function` constructor.

The `call()` method invokes the function specifying, as its first parameter, the object to serve as the function context, while the remainder of the parameters become the parameters of the called function—the second parameter to `call()` becomes the first argument of the called function and so on. The `apply()` method works in a similar fashion except that its second parameter is expected to be an array of objects that become the arguments to the called function.

Confused? It's time for a more comprehensive example. Consider the code of listing A.1 (found in the downloadable code as `appendixA/function.context.html`).

Listing A.1 Demonstrating that the value of the function context is dependent on how the function is invoked

```

<html>
  <head>
    <title>Function Context Example</title>
    <script>
      var o1 = {handle:'o1'};
      var o2 = {handle:'o2'};
      var o3 = {handle:'o3'};
      window.handle = 'window';

      function whoAmI() {
        return this.handle;
      }

      o1.identifyMe = whoAmI;

      alert(whoAmI());
      alert(o1.identifyMe());
      alert(whoAmI.call(o2));
      alert(whoAmI.apply(o3));

    </script>
  </head>

  <body>
  </body>
</html>

```

In this example, we define three simple objects, each with a `handle` property that makes it easy to identify the object given a reference ❶. We also add a `handle` property to the `window` instance so that it's also readily identifiable.

We then define a top-level function that returns the value of the `handle` property for whatever object that serves as its function context ❷ and assign the *same* function instance to a property of object `o1` named `identifyMe` ❸. We can say that this creates a method on `o1` named `identifyMe`, although it's important to note that the function is declared independently of the object.

Finally, we issue four alerts, each of which uses a different mechanism to invoke the same function instance. When loaded into a browser, the sequence of four alerts is as shown in figure A.4.

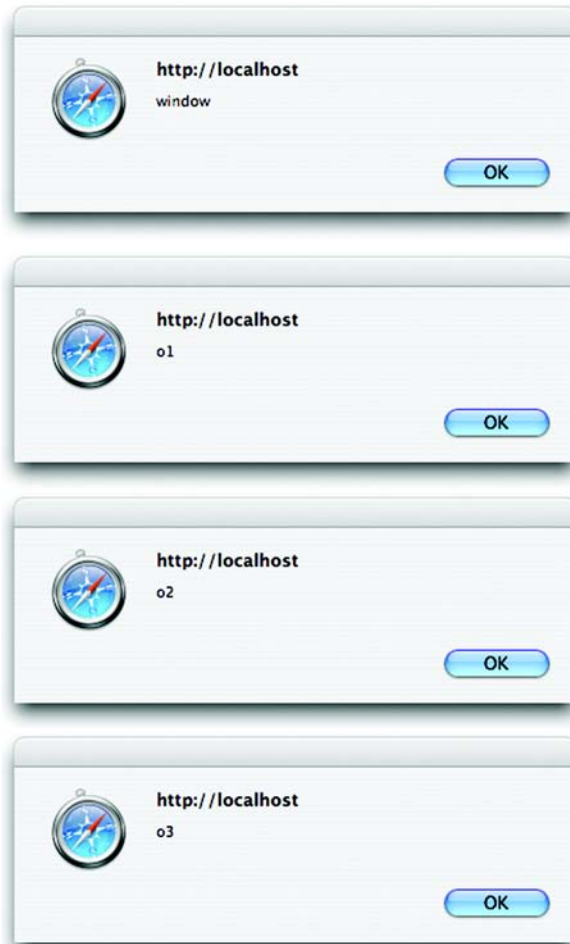


Figure A.4
The object serving as the function context changes with the manner in which the function is called.

This sequence of alerts illustrates the following:

- When the function is called directly as a top-level function, the function context is the `window` instance **4**.
- When called as a property of an object (`o1` in this case), the object becomes the function context of the function invocation **5**. We could say that the function acts as a *method* for that object—as in OO languages. But take care not to get too blasé about this analogy. You can be led astray if you're not careful, as the remainder of this example's results will show.

- Employing the `call()` method of `Function` causes the function context to be set to whatever object is passed as the first parameter to `call()`—in this case, `o2` ❹. In this example, the function acts like a method to `o2`, even though it has no association whatsoever—even as a property—with `o2`.
- As with `call()`, using the `apply()` method of `Function` sets the function context to whatever object is passed as the first parameter ❺. The difference between these two methods only becomes significant when parameters are passed to the function (which we didn't do in this example for simplicity).

This example page clearly demonstrates that the function context is determined on a per invocation basis and that a single function can be called with any object acting as its context. It's, therefore, probably never correct to say that a function is a method of an object. It's much more correct to state the following:

A function f acts as a method of object o when o serves as the function context of the invocation of f .

As a further illustration of this concept, consider the effect of adding the following statement to our example:

```
alert(o1.identifyMe.call(o3));
```

Even though we reference the function as a property of `o1`, the function context for this invocation is `o3`, further emphasizing that it's not how a function is declared but how it's invoked that determines its function context.

When using jQuery commands and functions that employ callbacks, this proves to be an important concept. We saw this concept in action early on (even if you didn't realize it at the time) in section 2.3.3 where we supplied a callback function to the `filter()` method of `$` and that function was sequentially invoked with each element of the wrapped set serving as its function context in turn.

Now that we understand how functions can act as methods of objects, let's turn our attention to another advanced function topic that will play an important role in effective usage of jQuery—closures.

A.2.4 Closures

To page authors coming from a traditional OO or procedural programming background, *closures* are often an odd concept to grasp; whereas, to those with a functional programming background, they're a familiar and cozy concept. For the uninitiated, let's answer the question: What are closures?

Stated as simply as possible, a *closure* is a `Function` instance coupled with the local variables from its environment that are necessary for its execution.

When a function is declared, it has the ability to reference any variables that are in its scope at the point of declaration. These variables are carried along with the function *even after* the point of declaration has gone out of scope, *closing* the declaration.

The ability for callback functions to reference the local variables in effect when they were declared is an essential tool for writing effective JavaScript. Using a timer once again, let's look at the illustrative example in listing A.2 (the file `appendixA/closure.html`).

Listing A.2 Gaining access to the environment of a function declaration through closures

```
<html>
  <head>
    <title>Closure Example</title>
    <script type="text/javascript"
      src="../../scripts/jquery-1.2.js"></script>
    <script>
      $(function(){
        var local = 1;      ← ❶
        window.setInterval(function(){ ← ❷
          $('#display')
            .append('<div>At ' + new Date() + ' local=' + local + '</div>');
          local++; ← ❸
        }, 3000);
      });
    </script>
  </head>

  <body>
    <div id="display"></div> ← ❹
  </body>
</html>
```

In this example, we define a ready handler that fires after the DOM loads. In this handler, we declare a local variable named `local` ❶ and assign it a numeric value of 1. We then use the `window.setInterval()` method to establish a timer that will fire every 3 seconds ❷. As the callback for the timer, we specify an inline function that references the `local` variable and shows the current time and the value of `local`, by writing a `<div>` element into an element named `display` that's defined in the page body ❹. As part of the callback, the `local` variable's value is also incremented ❸.

Prior to running this example, if we were unfamiliar with closures, we might look at this code and see some problems. We might surmise that, because the callback will fire off three seconds after the page is loaded (long after the ready handler has finished executing), the value of `local` is undefined during the execution of the callback function. After all, the block in which `local` is declared goes out of scope when the ready handler finishes, right?

But on loading the page and letting it run for a short time, we see the display as shown in figure A.5.

It works! But how?

Although it *is* true that the block in which `local` is declared goes out of scope when the ready handler exits, the closure created by the declaration of the function, which includes `local`, stays in scope for the lifetime of the function.

NOTE You might have noted that the closure, as with all closures in JavaScript, was created implicitly without the need for explicit syntax as is required in some other languages that support closures. This is a double-edged sword that makes it easy to create closures (whether you intend to or not!) but can make them difficult to spot in the code.

Unintended closures can have unintended consequences. For example, circular references can lead to memory leaks. A classic example of this is the creation of DOM elements that refer back to closure variables, preventing those variables from being reclaimed.



Figure A.5 Closures allow callbacks to access their environment even if that environment has gone out of scope.

Another important feature of closures is that a function context is never included as part of the closure. For example, the following code won't execute as we might expect:

```
...
this.id = 'someID';
$('*').each(function(){
    alert(this.id);
});
```

Remember that each function invocation has its own function context so that, in the code above, the function context within the callback function passed to `each()` is an element from the jQuery wrapped set, not the property of the outer function set to `'someID'`. Each invocation of the callback function displays an alert box showing the `id` of each element in the wrapped set in turn.

When access to the object serving as the function context in the outer function is needed, we can employ a common idiom to create a copy of the `this` reference in a local variable that *will* be included in the closure. Consider the following change to our example:

```
this.id = 'someID';
var outer = this;
$('*').each(function(){
    alert(outer.id);
});
```

The local variable `outer`, which is assigned a reference to the outer function's function context, becomes part of the closure and can be accessed in the callback function. The changed code now displays an alert showing the string `'someID'` as many times as there are elements in the wrapped set.

We'll find closures indispensable when creating elegant code using jQuery commands that utilize asynchronous callbacks, which is particularly true in the areas of Ajax requests and event handling.

A.3 Summary

JavaScript is a language that's widely used across the web, but it's often not *deeply* used by many of the page authors writing it. In this chapter, we introduced some of the deeper aspects of the language that we must understand to use jQuery effectively on our pages.

We saw that a JavaScript Object primarily exists to be a *container* for other objects. If you have an OO background, thinking of an Object instance as an unordered collection of name/value pairs may be a far cry from what you think of

as an *object*, but it's an important concept to grasp when writing JavaScript of even moderate complexity.

Functions in JavaScript are *first-class citizens* that can be declared and referenced in a manner similar to the other object types. We can declare them using literal notation, store them in variables and object properties, and even pass them to other functions as parameters to serve as callback functions.

The term *function context* describes the object that's referenced by the `this` pointer during the invocation of a function. Although a function can be made to act like a method of an object by setting the object as the function context, functions aren't declared as methods of any single object. The manner of invocation (possibly explicitly controlled by the caller) determines the function context of the invocation.

Finally, we saw how a function declaration and its environment form a *closure* allowing the function, when later invoked, to access those local variables that become part of the closure.

With these concepts firmly under our belts, we're ready to face the challenges that confront us when writing effective JavaScript using jQuery on our pages.

Symbols

\$

- as function namespace 154
- as namespace prefix 9
- avoiding name collisions 189
- defining locally 164
- in plugins 189
- in ready handler 165
- naming conflicts 14
- sharing with other libraries 164

\$() 6

- adding wrapper methods 216
- for element creation 11

\$.ajax() 249, 261, 278

\$.ajaxSetup() 252

\$.browser 157, 247

\$.each() 169

\$.extend() 177, 191, 210, 261

\$.fn 199, 246, 259

\$.get() 236, 252

\$.getJSON() 237, 241

\$.getScript() 180

\$.grep() 170

\$.inArray() 175

\$.livequery.run() 294

\$.makeArray() 175

\$.map() 172

\$.noConflict() 164, 189, 192

\$.post() 248, 252

\$.styleFloat 163

\$.trim() 9, 168

\$.unique() 176

A

abbr element 262

ActiveX control 218

Adaptive Path 218

add() 36, 247

addClass() 59

adding a wrapper method 199–201

adding select options 159

after() 73

Ajax 218–226

- comprehensive 258–266
- comprehensive jQuery API 249–252
- diagrammed 223
- Form Plugin, using 276–284
- form submission via 276–284
- GET 235–236
- Global Callback Info object 253
- global functions 254–258
- growth catalyst 2
- HTTP requests 233–234
- iframes, using 218
- initializing 219
- loading content 226
- loading scripts 181
- POST requests 248–249
- ready state handler 221
- request life cycle 223
- request parameters 221
- responses 223–224
- responseText 223
- responseXML 223
- setting defaults 252–253
- uploading files 284
- XML, not using 223

ajaxForm Lab 282, 284

ajaxForm() 282

ajaxFormUnbind() 282

ajaxSubmit Lab 279–280

ajaxSubmit() 277, 279

alpha filters 62

andSelf() 47

animate () 145, 148

animations

- CSS properties 146
- custom drop 148–150
- custom fading 147
- custom puff 150–151
- custom scale 148
- easing 147
- stopping 145
- using Flash 127

anonymous event handler 86

anonymous functions 165

anonymous listeners 86

API. *See* jQuery API

append() 70

appending content 70, 72

appendTo() 71

appetizer menu 113–124

Application Programming Interface. *See* jQuery API
 arrays, filtering 170–172
 assigning properties 51
 Asynchronous JavaScript and XML. *See* Ajax
 attr() 52, 54–55
 attribute selectors 22
 attributes 49
 applying 56–58
 diagrammed 50
 Internet Explorer limitations 56
 normalized values 53
 removing 56
 setting 54–56
 values, fetching 52–54
 augmenting wrapped sets 38

B

Basic Event Model 84
 before() 73
 behavior, separating from
 structure 4
 bind() 98, 293
 black box 225
 blur() 102, 107
 Boot Closet page 264–266
 box model 161, 286
 diagrammed 162
 browser detection 155–161
 alternatives 156–157
 jQuery flags 157–163
 bubble phase 94
 bubbling 88–90
 stopping 91

C

callback functions 329–330
 Camino 155–160, 328
 reloading 306
 capability detection 156
 capture phase 94
 cascading dropdowns 237–245
 Cascading Style Sheets 2
 class names 58, 67
 comma operator 38

chain 7
 managing 45–47
 chaining 45–47
 change() 102, 231
 child selector 21
 children() 43
 class names
 adding, removing 58–60
 fetching 67
 Clear and Reset Lab 274
 clearForm() 275
 click() 102, 107
 clone() 45, 78
 closures 84, 90, 211, 334–337
 code consistency, reusability 186
 collapsible list, implementing 128–134
 collecting properties 52
 collisions, naming 187–189
 command chain 7
 managing 45–47
 commands 17
 commerce 229
 container selector 23
 contains() 44
 content scrolling 287
 contents() 43
 copying
 address information 202–206
 elements 78
 creating DOM elements 11
 creating utility functions 194, 196–199
 cross-browser Ajax 219
 CSS
 absolute positioning 150
 hiding elements 128
 inline vs block 128
 offset parent 289
 opacity 142, 147
 overflow 289
 relative positioning 149
 styling lists 132
 See also Cascading Style Sheet
 css() 61–62, 132, 200
 CSS3 6

custom animations 145–151
 custom attributes 52–53
 custom properties 211
 custom selectors 27–30

D

data conversion 173
 data translation 172–176
 Date 195
 date formatting 195–199
 dblclick() 102
 defining functions 193
 dependent dropdowns 237
 DHTML 49
 dimensions 286–289
 Dimensions Plugin 285–291
 height() 285
 innerHeight() 286
 innerWidth() 286
 offset() 290
 offsetParent() 290
 outerHeight() 286
 outerWidth() 286
 position() 290
 scrollLeft() 287
 scrollTop() 287
 width() 285
 Dimensions Plugin Scrolling Lab 287
 exercises 289
 disabling form elements 13
 DOCTYPE 161
 Document Object Model (DOM) 10, 17, 49–58
 cloning elements 78
 copying elements 70, 78–80
 creating new elements 11
 event bubbling 88–90
 form elements 79–80
 generating elements 31–32
 manipulation 68
 moving elements 70–80
 NodeList 50
 removing elements 76–77
 replacing elements 77
 setting content 68
 wrapping elements 75
 document ready handler 9

- document scrolling 288
 - DOM elements, positioning
 - 285–291
 - DOM Level 0 85–87
 - DOM Level 1 84
 - DOM Level 2 91–96
 - DOM manipulation 258
 - DOM. *See* Document Object Model (DOM)
 - Double Submit Problem 57
 - drag-and-drop 300
 - draggable() 300, 304
 - draggableDestroy() 303
 - draggableDisable() 304
 - draggableEnable() 304
 - draggables 300–308
 - axis of movement 307
 - constraining 307
 - containment 307
 - grid 308
 - required files 301
 - suitability concept 309
 - dropdowns 237
 - droppable() 308
 - droppableDestroy() 314
 - droppableDisable() 315
 - droppableEnable() 315
 - droppables 308–315
 - drop event 309
 - options 309
 - required files 308
 - state transition diagram 309
 - states 309
 - dynamically loading scripts
 - 180–183
- E**
-
- each() 13, 51, 90, 148, 200, 211, 246, 337
 - easing 147
 - Easing Plugin 147
 - effects 127
 - custom drop 148
 - custom fading 147
 - custom puff 150
 - custom scale 148
 - fading 140
 - hide 127
 - show 127
 - sliding 143–144
 - elements
 - abbr 262
 - animating 135–145
 - attributes 49–58
 - cloning 78
 - content 68–70
 - copying 70, 78–80
 - event handlers 85
 - fading 140
 - form elements 79–80
 - listeners 85
 - moving 70–80
 - properties 49–58
 - removing 76–77
 - replacing 77
 - selecting 17
 - setting content 68
 - showing and hiding 127–130
 - sliding 143–144
 - span 262
 - styling 58–68
 - title attribute 258
 - tooggling 134
 - wrapping 75
 - empty() 77
 - emptying select elements 246
 - encodeURIComponent() 221
 - end() 46
 - error callback 253
 - error() 102
 - event handlers 83
 - anonymous 86
 - as attributes 86
 - removing 103
 - tooggling 108–110
 - event handling
 - hovering 110–112
 - proactive 292–293
 - Event instance 87, 104
 - cancelBubble 91
 - normalizing 104
 - preventDefault() 106
 - stopPropagation() 91, 106
 - target 107
 - event models
 - Basic 85–87
 - DOM Level 0 85–87
 - DOM Level 2 91–96
 - Internet Explorer 97
 - jQuery 98–124
 - Netscape 85–87
 - event propagation 106
 - diagrammed 95
 - event, toggling 108
 - event-driven interfaces 83
 - events 112–124
 - addEventListener() 92
 - attachEvent() 97
 - binding 98
 - bubble phase 94
 - bubbling 88–90
 - cancelling propagation 91
 - capture phase 94
 - establishing multiple 92–94
 - Event instance 88
 - key codes 105
 - modifier keys 105
 - propagation 94–96
 - semantic actions 106
 - srcElement 88
 - stopping propagation 106
 - target element 88
 - triggering 106–107
 - expando 211
 - expire() 295
 - extending jQuery 13, 186–216
 - comprehensive examples 245
 - defining wrapper methods 199–216
 - emptying select elements 246
 - implementation functions 212–214
 - in \$ namespace 192–193
 - loading select elements 246
 - motivations 186–187
 - naming files 188
 - The Termifier 260–264
 - utility functions 192–193
 - extending objects 176–180

F

fadeIn() 141
 fadeOut() 141
 fadeTo() 142
 fieldSerialize() 272
 fieldValue() 270–272
 filesystem browsing 128
 filter selectors 29
 filter() 41, 202
 filtering data 170–172
 filtering wrapped sets 40
 find selectors 29
 find() 44
 Firebug 321
 Firefox 155, 160, 321, 328
 reload vs. refresh 140
 reloading 306
 fixed-width output 193–194
 flags 154
 \$.boxModel 161
 \$.browser 157
 \$.styleFloat 163
 Flash 127
 float styles, browser differences 163
 flyout 258
 focus() 102, 107
 form controls
 clearing 274–276
 resetting 274–276
 serializing 272–274
 submission order 273
 successful concept 270
 form elements 79–80
 Form Plugin 80, 269–284
 ajaxForm Lab 282
 ajaxForm() 282
 ajaxFormUnbind() 282
 ajaxSubmit Lab 279
 ajaxSubmit() 277
 Clear and Reset Lab 274
 clearForm() 275
 download location 269
 fieldSerialize() 272
 fieldValue() 271
 formSerialize() 272
 Get Form Value Lab 271

 resetForm() 275
 uploading files 284
 form submission
 hijacking 281–284
 semantic events 282
 via Ajax 281–284
 via focusable element 284
 formatting fixed-width output 194
 forms, serializing 227
 formSerialize() 272
 Function
 apply() 331
 as first-class object 326
 call() 331
 function keyword operation 327
 naming 327–328
 function context 84, 200, 330–331
 functional programming 334
 functions
 as callbacks 329–330
 as methods 328, 334
 assigning context 331
 context 330
 function literal 327
 top-level 327

G

Gecko 158, 328
 GET 221, 227, 234, 261
 Get Form Value(s) Lab 271, 273
 get() 35
 GIF 266
 global Ajax functions 253–258
 Global Callback Info object 253
 global flags 154
 global namespace 14
 polluting 198
 Google 218
 Graphical User Interface (GUI) 83
 drag-and-drop 300
 grep 170
 GUI. *See* Graphical User Interface (GUI)

H

halting form submission 91
 hasClass() 67
 hash as parameter 190–192
 height() 63–64, 285
 hide() 127, 135
 hover() 111
 hovering 110–112
 HTML 5 Draft Specification 262
 HTML Specification 270
 html() 7, 68
 HTML, generation 31–32
 HTTP. *See* Hypertext Transfer Protocol (HTTP)
 Hypertext Transfer Protocol (HTTP) 83, 248
 methods 221, 227
 effects on caching 234
 status codes 222

I

idempotent requests 234
 iframe 284
 implementation functions 213
 index() 36
 information overload 128, 130
 inheritance 176
 innerHeight() 286
 innerWidth() 286
 insertAfter() 73
 insertBefore() 73
 inserting dynamic HTML 74
 Internet Explorer 155, 158, 160
 box model 161
 event handling limitations 96
 Event Model 97
 inverting selectors 29–30
 is() 45, 67, 132, 135
 iterating 51
 properties and collections 169–170

J

-
- Java Swing 83
 - JavaScript
 - . operator 323
 - adding select options 158
 - advanced example 258
 - closure variables 335
 - closures 84, 211, 334–337
 - creating objects 320
 - Date 195
 - dot operator 323
 - dynamic creation of
 - properties 321
 - essential concepts 320
 - extending objects 176
 - for loop 169
 - for-in loop 169
 - Function 84
 - function contexts 84, 200
 - function keyword 327
 - functions 326
 - general reference operator
 - 323
 - getAttribute() 53
 - isNaN() 173
 - libraries 14
 - libraries, using with jQuery
 - 163–167
 - NaN 173
 - navigator 155
 - new operator 320
 - nonstandard property
 - names 323
 - Number 174
 - Object 84, 320–326
 - object hash 191
 - Object literals 324–325
 - object properties 321–323
 - object property diagram
 - 322
 - object-oriented 176
 - property references 322
 - prototype property 177
 - regular expressions 172, 198
 - setAttribute() 53
 - String 168, 172, 174
 - top-level scope 325
 - Unobtrusive JavaScript 3–5
 - var keyword explained 325
 - window properties 325
 - XML and 223
 - Jedi mind tricks 327
 - Jesse James Garrett 218
 - jQuery 2–5
 - \$() 6
 - add() 36
 - browser detection flags
 - 157–163
 - chaining 7, 186, 200, 261
 - chains 45–47
 - commands 17
 - contains() 44
 - CSS implementation 20
 - custom selectors 27–30
 - document ready handler 9
 - DOM manipulation 17
 - essential JavaScript
 - concepts 319
 - Event Model 99
 - extending 12–13, 186–216
 - flags 154
 - global Ajax commands 254
 - manipulating objects 167–180
 - plugins 268–317
 - selectors 17–30
 - translating data 172–176
 - trimming strings 168
 - using with other libraries
 - 14, 163–167
 - utility functions 8, 154
 - wrapper 6–7
 - jQuery API
 - \$.ajax() 249
 - \$.ajaxSetup 252
 - \$.boxModel 161
 - \$.browser 157
 - \$.each() 169
 - \$.extend() 177
 - \$.get() 234–236
 - \$.getJSON() 237
 - \$.getScript() 180
 - \$.grep() 170
 - \$.inArray() 175
 - \$.makeArray() 175
 - \$.map() 172
 - \$.noConflict 164
 - \$.post() 248
 - \$.styleFloat 163
 - \$.trim() 9, 168
 - \$.unique() 176
 - addClass() 59
 - after() 73
 - ajaxComplete() 254
 - ajaxError() 254
 - ajaxSend() 254
 - ajaxStart() 254
 - ajaxStop() 254
 - ajaxSuccess() 254
 - andSelf() 47
 - animate() 145
 - append() 70
 - appendTo() 71
 - attr() 52, 54–55
 - before() 73
 - bind() 98
 - blur() 102, 107
 - change() 102
 - children() 43
 - click() 102, 107
 - clone() 78
 - contents() 43
 - css() 61–62
 - dblclick() 102
 - each() 13, 51
 - empty() 77
 - end() 46
 - error() 102
 - fadeIn() 141
 - fadeOut() 141
 - fadeTo() 142
 - filter() 41
 - find() 44
 - focus() 102, 107
 - get() 35
 - hasClass() 67
 - height() 63–64
 - hide() 135
 - hover() 111
 - html() 7, 68
 - index() 36
 - insertAfter() 73
 - insertBefore() 73
 - is() 45
 - jQuery() 6

jQuery API (*continued*)

- keydown() 102
- keypress() 102
- keyup() 102
- load() 102, 226
- mousedown() 102
- mousemove() 102
- mouseout() 102
- mouseup() 102
- next() 43
- nextAll() 43
- noConflict() 14
- not() 39
- one() 102
- parent() 43
- parents() 43
- prepend() 73
- prependTo() 73
- prev() 43
- prevAll() 43
- ready() 10
- removeAttr() 56
- removeClass() 59
- resize() 102
- scroll() 102
- select() 102, 107
- serialize() 227
- serializeArray() 228
- show() 136
- siblings() 43
- size() 34
- slice() 42
- slideDown() 143
- slideToggle() 144
- slideUp() 144
- stop() 145
- submit() 102, 107
- toggle() 108, 137
- toggleClass() 59
- trigger() 106
- unbind() 103
- unload() 102
- val() 79–80
- width() 63–64
- wrap() 75
- jQuery Effects Lab 138–144
- JSON 236, 241, 263, 277, 324
 - array example 325
 - array notation 241

- as Ajax response 223
- JavaScript Dates and 324
- object example 324
- JSP 225

K

- keydown() 102
- keypress() 102
- keyup() 102
- Konqueror 161

L

- Laboratory pages
 - ajaxForm Lab 282
 - ajaxSubmit Lab 279
 - Clear and Reset Lab 274
 - Dimensions Plugin Scrolling Lab 287
 - Get Form Value Lab 271
 - Get Form Values 271
 - Live Query Lab 296
 - Move and Copy Lab 72
 - Selectors Lab 17
 - UI Draggables Lab 304
 - UI Droppables Lab 311
 - Wrapped Set Lab 33
- leveraging jQuery 186
- libraries, using with jQuery 14
- listeners 83
 - match and mismatch 294
 - proactive 292–293
- Live Query Lab 296
- Live Query Plugin 292–299
 - \$.livequery.run() 294
 - capabilities 292
 - expire() 295
 - expiring listeners 295
 - forcing evaluation 294
- livequery() 293–294
- load() 102, 226, 231, 252, 263
- loading content
 - Ajax 224–226
 - dynamic data 229–233
 - jQuery 226–228
 - select elements 246

M

- match handler 298
- match listeners 294
- merging objects 179
- merging options 210
- method 334
- Microsoft 218
- MIME type in Ajax response 223
- mismatch handler 298
- mismatch listeners 294
- mousedown() 102
- mousemove() 102
- mouseout() 102
- mouseup() 102
- Move and Copy Lab 72
- moz-opacity 62
- multi-browser support 159
- multipart forms 284
- multiple class names 58

N

- name collisions 187–189
- namespace, global 198
- NaN 174
- navigator object 155
- nesting selectors 23
- .NET framework 83
- Netscape Communications Corporation 84
- Netscape Event Model 84
- Netscape Navigator 84
- next() 43
- nextAll() 43
- noConflict() 14
- NodeList 50, 175
- non-idempotent requests 234
- normalizing event targets 88
- not() 39
- Number.NaN 174

O

- Object
 - literals 324–325
 - properties 321–323

- object detection 156, 247
 - feasibility 158
 - for Ajax 219
 - object hash 190–192
 - object orientation 176
 - object-oriented JavaScript 176
 - objects, extending 176
 - Observable pattern 92
 - offset parent 289
 - offset() 290
 - offsetParent() 290
 - offsets 289–291
 - OmniWeb 155, 157, 160
 - one() 102
 - online retailers 229
 - onload handler, alternative to 9
 - onresize 65
 - Opera 155, 157, 160, 247
 - operator 323
 - options hash 190–192, 261
 - extensive example 208
 - used for wrapper method 261
 - OS X, dragging images from browser 305
 - outerHeight() 286
 - outerWidth() 286
 - Outlook Web Access 218
 - overflow 289
- P**
-
- page rendering 161
 - parameters 190–192
 - parent() 43
 - parents() 43
 - parsing ZIP codes 172
 - patterns
 - necessity in Rich Internet Applications 5
 - Observable 92
 - options hash 192
 - progressive disclosure 113
 - repetitiveness 186
 - responsibilities and 5
 - structure and 4
 - Photomatic jQuery extension 206–216
 - PHP 225
 - plain text response 223
 - plugins 268–317
 - creating 186–216
 - Dimensions 149
 - Dimensions Plugin 285–291
 - Easing 147
 - Form Plugin 269–284
 - Live Query Plugin 292–299
 - on the web 269
 - strategy 269
 - UI Plugin 299–316
 - PNG files 266
 - position() 290
 - positional selectors 24–27
 - positioning 289
 - positions 289–291
 - POST 221, 227, 234, 248–249
 - prepend() 73
 - prependTo() 73
 - prev() 43
 - prevAll() 43
 - proactive event handling 292
 - product description page 231–233
 - progressive disclosure 113, 128
 - propagation of events 94–96
 - properties 49
 - diagrammed 50
 - of JavaScript objects 321–323
 - property referencing 322
 - Prototype 192
 - using with jQuery 14, 163–167
 - prototype 177, 199
- Q**
-
- query string 221, 227, 274
 - quirks mode 161
- R**
-
- read-only status, applying 201–206
 - ready handler 165
 - ready state handler 222–223
 - ready() 10
 - real-time data 229
 - regular expression 170
 - relational selectors 20
 - removeAttr() 56
 - removeClass() 59
 - removing elements 77
 - removing wrapped set elements 39
 - rendering
 - quirks mode 161
 - strict mode 161
 - rendering engine 158
 - replacing elements 77
 - reporting Ajax errors 253
 - request header 155
 - request parameters 221, 227, 236, 264
 - requests
 - idempotent 234
 - non-idempotent 234, 248
 - resetForm() 275
 - resize() 102
 - resources
 - Form Plugin download 269
 - jQuery plugins 188
 - plugins 269
 - quirksmode.org 161
 - UI Plugin URL 300
 - response 223–224
 - JSON 236
 - responseText 220, 223, 226
 - responseXML 223
 - retail web sites 229
 - reusability 186
 - reusable components 167
 - RIA. *See* Rich Internet Application (RIA)
 - Rich Internet Application (RIA) 2, 17, 92, 218, 237
- S**
-
- Safari 155, 160, 247
 - problems loading scripts 181
 - scripts, dynamic loading 180–183

scroll() 102
 scrollbars 288
 scrolling dimensions 287–289
 scrollLeft() 287
 scrollTop() 287
 select() 102, 107
 selecting check boxes 27
 selectors 3, 6, 17–30, 258

- attribute 20, 22–24
- basic 19–20
- child 20–24
- container 20, 23–24
- CSS syntax 17
- custom 27–30
- filter 29
- find 29
- form-related 29
- inverting 29–30
- nesting 23
- positional 24–27
- regular expression syntax 22
- relational 20–24
- XPath plugin 30

 Selectors Lab 17
 self-disabling form 57
 semantic actions 83, 106
 serialize() 227
 serializeArray() 228
 serializeForm() 276
 server setup 225
 server-side

- resources 225
- state 248
- templating 230

 servlet 225
 setting width 63
 show() 127, 136
 siblings() 43
 size() 34
 slice() 42
 slideDown() 143
 slideshows 206–209
 slideToggle() 144
 slideUp() 144
 sniffing 155–161
 spoofing 155
 stop() 145
 strict mode 161

string trimming 168
 styles, setting 61–67
 styling 58–68
 submit() 102, 107
 submitting forms

- Ajax 276–284

 subsetting wrapped sets 42–43

T

Termifier, The 259
 test-driven development 208
 text() 69
 this 330–334
 thumbnail images 206
 timers 83
 title attribute 258, 261
 toggle() 108, 134–137
 toggleClass() 59
 toggling display state 134
 Tomcat web server 225
 tooltip 258
 top-level flags and functions 154
 translating data 173
 translation 172–176
 Trash 300
 trigger() 106
 trimming 168

U

UI Draggables Lab 304–305
 UI Droppables Lab 311–313
 UI Plugin 299–316

- accordian 316
- calendar 316
- dialog 316
- download location 300
- draggable() 300
- draggableDestroy() 303
- draggableDisable() 304
- draggableEnable() 304
- draggables 300–308
- drop shadow 316
- drop zones 312
- draggable() 308

draggableDestroy() 314
 draggableDisable() 315
 draggableEnable() 315
 droppables 308–315
 Droppables Lab 311
 magnify 316
 mouse interactions 300–315
 required files 301
 resizables 315
 selectables 315
 slider 316
 sortables 315
 table 316
 tabs 316
 UI Draggables Lab 304
 widgets 316
 UI principles

- gradual transition 135
- progressive disclosure 128

 unbind() 103, 295
 United States Postal Codes 172
 unload() 102
 Unobtrusive JavaScript 3, 87, 115, 230, 292

- practical application 209

 unsuccessful form elements 79
 uploading via Ajax 284
 URI encoding 221, 227
 URL 221, 227

- encoding 272

 user agent detection 155–161
 user interface, annoyances 49
 utility functions 8, 154

V

val() 79–80, 231, 270
 variables as part of closure 335
 viewport scrolling 287

W

W3C 159

- box model 161

 W3C DOM Specification 84
 Wastebasket 300

- web server 225
- width and height 64
- width() 63–64, 285
- wiki 57
- window
 - origin 289
 - properties 325
 - scrolling 288
- window.event 87, 97, 104
- window.setInterval() 335
- window.setTimeout() 329
- wrap() 75
- wrapped set 7, 32, 43–44
 - adding elements 36–39
 - as array 34
 - augmenting 36–43
 - determining size 34
 - filtering 40
 - iterating over 51
 - manipulation 32

- obtaining elements from
 - 34–36
 - removing elements 39–42
 - subsetting 42–43
- Wrapped Set Lab 33
- wrapper 7
- wrapper methods
 - applying multiple operations 201–206
 - defining 199–216
 - implementation functions 212–214
 - maintaining state 210

X

- X11 83
- XHR. *See* XMLHttpRequest (XHR)
- XHTML 53

- XML 236, 277
- XML DOM 223
- XMLHTTP 218
- XMLHttpRequest (XHR) 218
 - instance creation 219
 - instantiating 219
 - making the request 221–222
 - methods and properties 219
 - ready state handler 221
 - responses 223–224
 - responseText 223
 - responseXML 223
 - status 222
- XPath selectors plugin 30

Z

- zebra-striping 2
- ZIP Codes 172

jQuery in Action

Bear Bibeault • Yehuda Katz

A really good web development framework anticipates your needs. jQuery does more—it practically reads your mind. Developers fall in love with this JavaScript library the moment they see 20 lines of code reduced to three. jQuery is concise and readable. Its unique “chaining” model lets you perform multiple operations on a page element in succession, as in `$(“div.elements”).addClass(“myClass”).load(“ajax_url”).fadeIn()`

jQuery in Action is a fast-paced introduction and guide. It shows you how to traverse HTML documents, handle events, perform animations, and add Ajax to your web pages. The book’s unique “lab pages” anchor the explanation of each new concept in a practical example. You’ll learn how jQuery interacts with other tools and frameworks and how to build jQuery plugins. This book requires a modest knowledge of JavaScript and Ajax.

What’s Inside

- Countless practical examples
- DOM manipulation and event handling
- Animation and UI effects
- Painless Ajax
- Based on jQuery 1.2

Bear Bibeault is a JavaRanch senior moderator and coauthor of Manning’s *Ajax in Practice* and *Prototype and Scriptaculous in Action*. **Yehuda Katz** is a developer with Engine Yard. He heads the jQuery plugin development team and runs Visual jQuery.

“The best-thought-out and researched piece of literature on the jQuery library.”

—FROM THE FOREWORD BY
John Resig, Creator of jQuery

“Solve your complex UI problems in no time. A must read.”

—Jonathan Bloomer, Soap Creative

“A fantastic journey through jQuery.”

—John C. Tyler
UBS Investment Bank

“A superior guide to the superior JavaScript library.”

—Gregg Bolinger, VML

“This book is like jQuery itself—fast, effective, efficient.”

—Eric Pascarello
Author of *Ajax in Action*

“*jQuery in Action* has become my best friend. A great read.”

—Andrew Siemer, OTX Research

“*The Elements of Style* for JavaScript.”

—Joshua Heyer, Trane Inc.

For more information, code samples, and to purchase an ebook visit manning.com/jqueryinAction