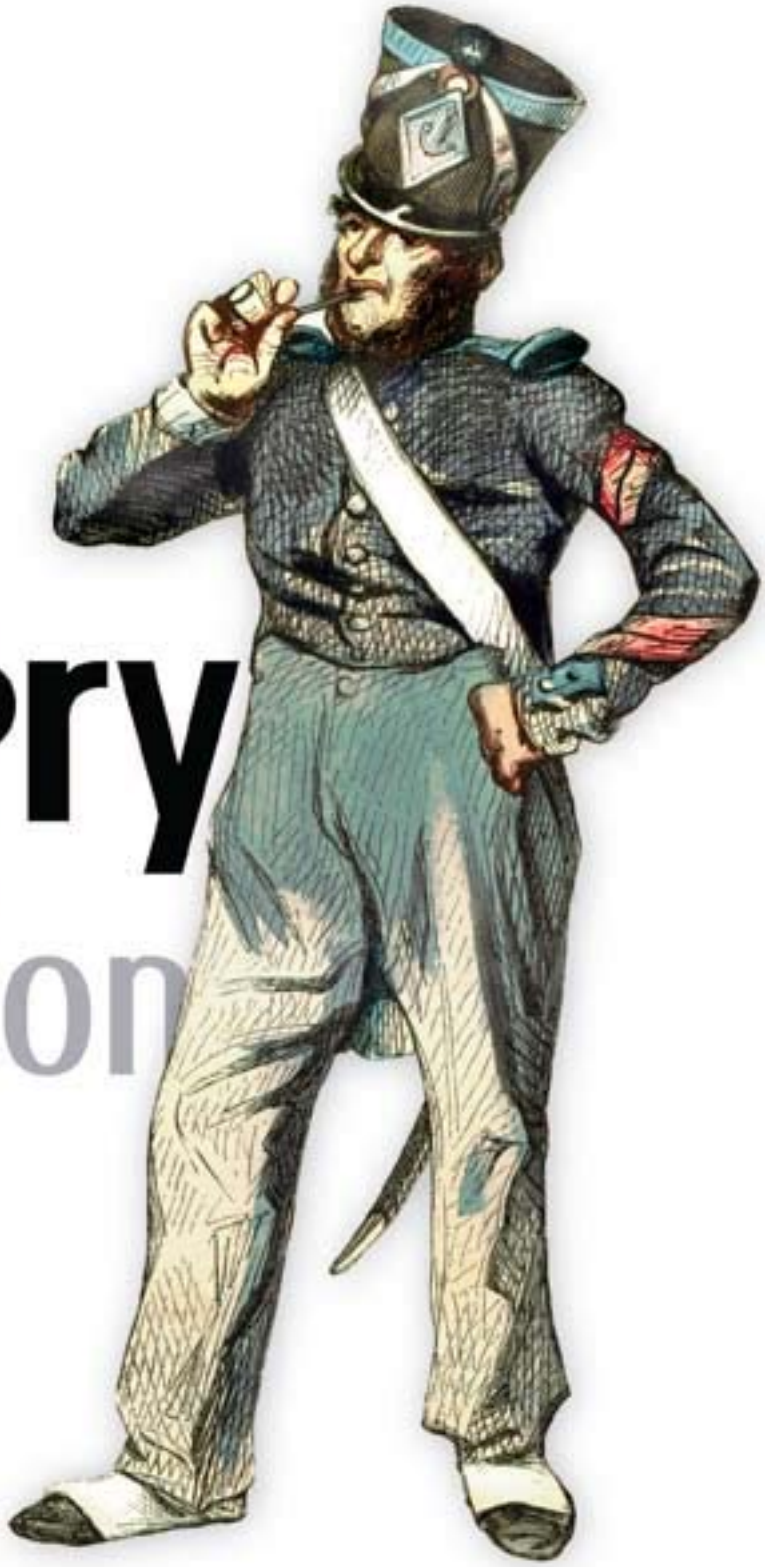
Bear Bibeault
Yehuda Katz

FOREWORD BY John Resig
Creator of jQuery

jQuery in Action



MANNING

jQuery in Action

jQuery in Action

BEAR BIBEAULT
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

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

Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 12 11 10 09 08

contents

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 *copyedit* 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 BIBEAULT 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.

1 Introducing jQuery

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—

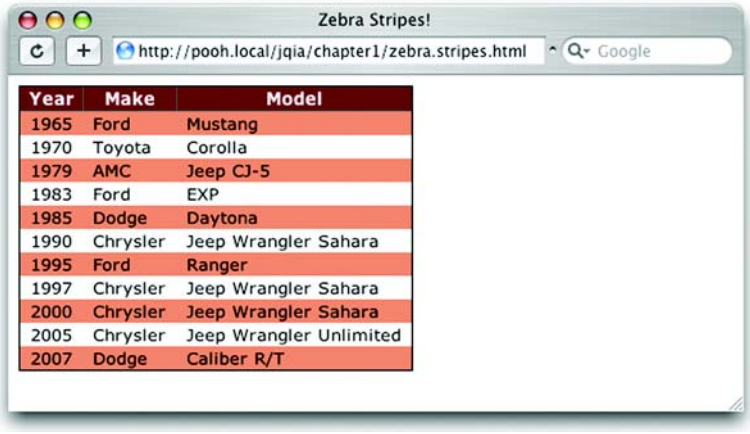


Figure 1.1 Adding "zebra stripes" to a table is easy to accomplish in one statement with jQuery!

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 `<font>` 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 `<font>` 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>
```

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-striping 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-striping 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-striping 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>
```

❶ Ready handler that creates HTML element

❷ Existing element to be followed

This example establishes an existing HTML paragraph element named `followMe` ❷ in the document body. In the script element within the `<head>` section, we establish a ready handler ❶ 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 it 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.

2 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.

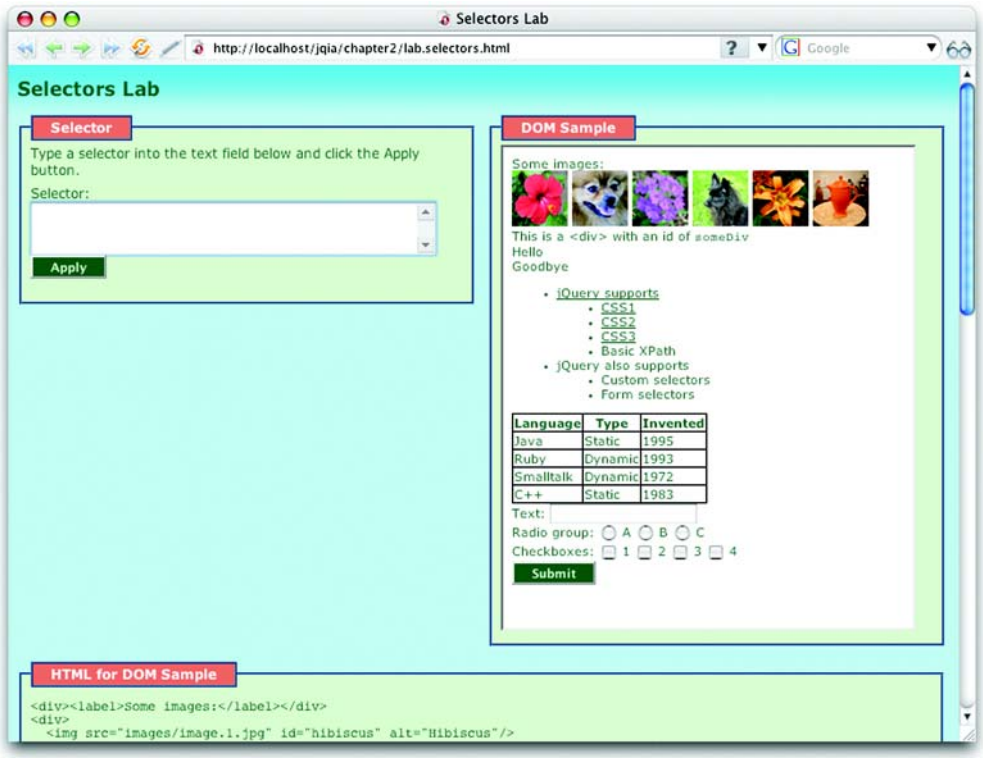


Figure 2.1 The Selectors Lab page allows us to observe the behavior of any selector we choose in real time.

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 `<li>` elements in the page are highlighted.

Note that the `<li>` 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.

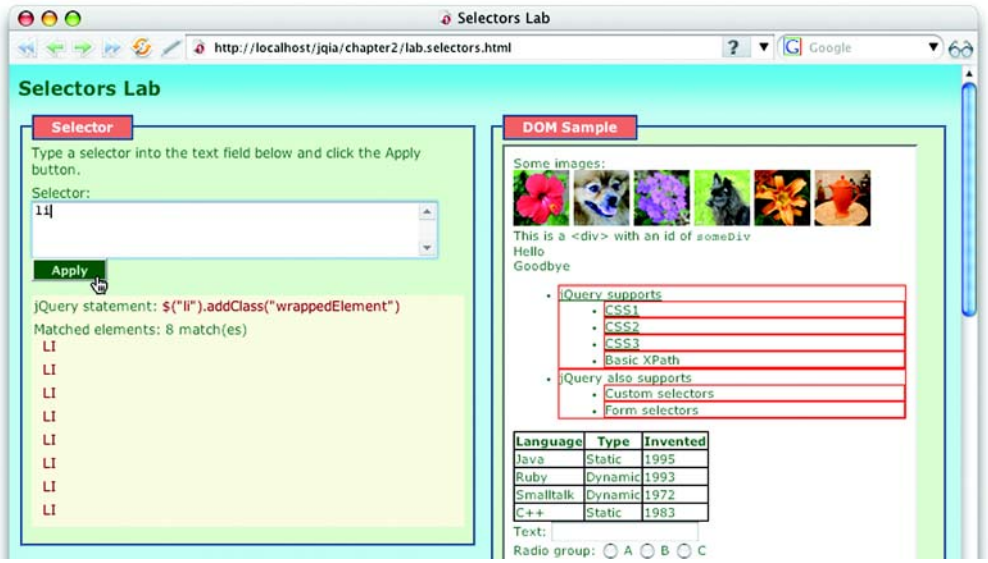


Figure 2.2 A selector value of `li` matches all `<li>` elements when applied as shown by the display results.

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 `<span>` 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 `<ul>` elements that have the class `myList`. The links contained in the sublists are excluded because the `<ul>` elements serving as the

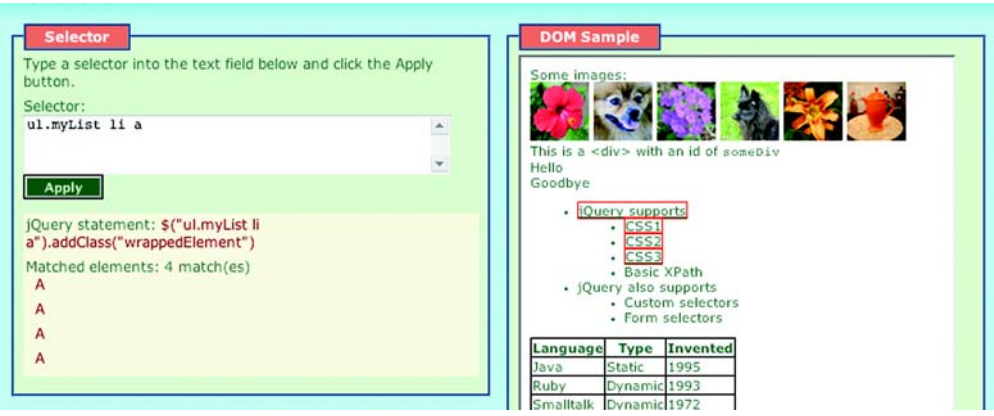


Figure 2.3 All anchor tags that are descendents, at any depth, of an `<li>` element are selected by `ul.myList li a`.

parent of the sublists `<li>` 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]
```

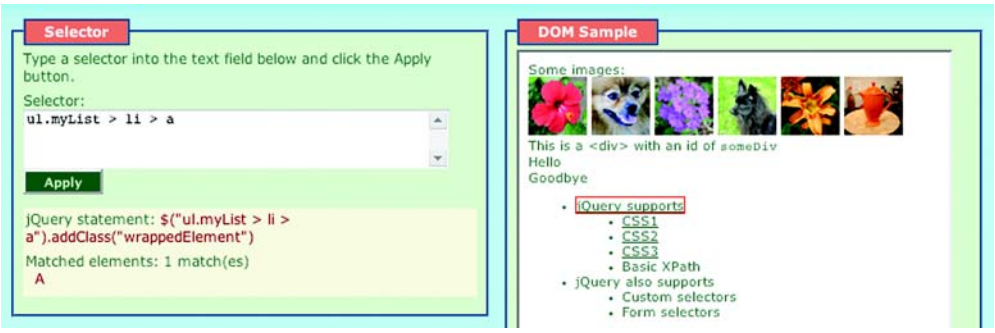


Figure 2.4 With the selector `ul.myList > li > a`, only the direct children of parent nodes are matched.

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 `<li>` 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 `<li>` 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 <li> child of each <ul> 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
`:first`	The first match of the page. `li a:first` returns the first link also under a list item.
`:last`	The last match of the page. `li a:last` returns the last link also under a list item.
`:first-child`	The first child element. `li:first-child` returns the first item of each list.
`:last-child`	The last child element. `li:last-child` returns the last item of each list.
`:only-child`	Returns all elements that have no siblings.
`:nth-child(`*n*`)`	The *n*th child element. `li:nth-child(2)` returns the second list item of each list.
`:nth-child(even|odd)`	Even or odd children. `li:nth-child(even)` 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
`:nth-child(Xn+Y)`	The *n*th child element computed by the supplied formula. If `Y` is `0`, it may be omitted. `li:nth-child(3n)` returns every third item, whereas `li:nth-child(5n+1)` returns the item after every fifth element.
`:even and :odd`	Even and odd matching elements page-wide. `li:even` returns every even list item.
`:eq(n)`	The *n*th matching element.
`:gt(n)`	Matching elements after (and excluding) the *n*th matching element.
`:lt(n)`	Matching elements before (and excluding) the *n*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
`:animated`	Selects elements that are currently under animated control. Chapter 5 will cover animations and effects.
`:button`	Selects any button (`input[type=submit]`, `input[type=reset]`, `input[type=button]`, or `button`).
`:checkbox`	Selects only check box elements (`input[type=checkbox]`).
`:checked`	Selects only check boxes or radio buttons that are checked (supported by CSS).
`:contains(foo)`	Selects only elements containing the text *foo*.
`:disabled`	Selects only form elements that are disabled in the interface (supported by CSS).
`:enabled`	Selects only form elements that are enabled in the interface (supported by CSS).
`:file`	Selects all file elements (`input[type=file]`).
`:header`	Selects only elements that are headers; for example: `<h1>` through `<h6>` elements.
`:hidden`	Selects only elements that are hidden.
`:image`	Selects form images (`input[type=image]`).
`:input`	Selects only form elements (`input`, `select`, `textarea`, `button`).
`:not(filter)`	Negates the specified filter.
`:parent`	Selects only elements that have children (including text), but not empty elements.
`:password`	Selects only password elements (`input[type=password]`).
`:radio`	Selects only radio elements (`input[type=radio]`).
`:reset`	Selects reset buttons (`input[type=reset]` or `button[type=reset]`).

continued on next page

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

Selector	Description
`:selected`	Selects option elements that are selected.
`:submit`	Selects submit buttons (`button[type=submit]` or `input[type=submit]`).
`:text`	Selects only text elements (`input[type=text]`).
`:visible`	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 $(" ") and $(" ")

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