THIRD EDITION

jQuery
IN ACTION

Bear Bibeault
Yehuda Katz
Aurelio De Rosa

Forewords by Dave Methvin
John Resig

MANNING

Praise for Earlier Editions of *jQuery in Action*

Every technical book should be like this one…concise but clear, humorous but not silly, and one that answers all the questions it raises, quickly. The reader is never left wondering "But what about…" for more than a sentence or two.

—JRoller Online Book Reviews

Thanks to the authors and their exemplary style, this comprehensive book, or operating manual as it might be called, can be taken in a front-to-back approach to learn from scratch, or as a reference for those already dabbling in jQuery and needing verification of best practices.

—Matthew McCullough
Denver Open Source Users Group

With its capable technical coverage, extensive use of sample code, and approachable style, this book is a valuable resource for any web developer seeking to maximize the power of JavaScript, and a must-have for anyone interested in learning jQuery.

—Michael J. Ross
Web Developer and Slashdot Contributor

An excellent work, a worthy successor to others in Manning's In Action series. It is highly readable and chock-full of working code. The Lab Pages are a marvelous way to explore the library, which should become an important part of every web developer's arsenal. Five stars all 'round!

—David Sills
JavaLobby, DZone

I highly recommend the book for learning the fundamentals of jQuery and then serving as a good reference book as you leverage the power of jQuery more and more in your daily development.

—David Hayden
MVP C#, Codebetter.com

I highly recommend this book to any novice or advanced JavaScript developers who want to get serious about JavaScript and start writing optimized and elegant code without all the hassle of traditional JavaScript code authoring.

—Val's Blog

The Elements of Style *for JavaScript.*

—Joshua Heyer
Trane Inc.

jQuery in Action

THIRD EDITION

BEAR BIBEAULT
YEHUDA KATZ
AURELIO DE ROSA

To Annarita, because you give balance to my life
—Aurelio

brief contents

contents

ix

foreword to the third edition

A decade ago, John Resig imagined a JavaScript library that would simplify the way people built web sites. Today, that library, jQuery, is used by more than 80% of all web sites that use JavaScript, according to BuiltWith.com. It would be hard to call yourself a web developer today without knowing jQuery.

On the technical side, jQuery simplifies the long-winded native method calls that browsers use and shrinks the number of lines of code that it takes to get things done. That's why jQuery's motto is "Write less, do more." jQuery also paves over the differences in behavior—and even some outright bugs—that exist in browsers. That simplifies both development and testing.

From the start, jQuery was designed so that it could be extended by others. The jQuery plugin model lets anyone build specialized functionality on top of what jQuery already offers. There are thousands of jQuery plugins that do everything from lightboxes to form validation. The result is that many people with only a modest amount of programming skill are able to create beautiful and functional web sites by building on the work that others have done.

Still, code alone is not what made jQuery popular. From the beginning, a strong community of helpful developers filled online forums and mailing lists to answer questions for newcomers. The insight gained from those discussions led to better documentation, training classes, and books like this one.

This book is a great way to learn jQuery. Early on, it covers a central tenet of jQuery's API, which is to select some elements on a web page and do something with them. That same pattern applies whether you are hiding, showing, animating, removing, or changing an element's appearance. The selection process uses the standard CSS selector syntax, with some jQuery enhancements that give selection even more power.

I must confess that the chapter on events is my favorite because my first major code contribution to jQuery was the rewrite of the event module in jQuery 1.7. This chapter does a great job of explaining the purpose and usefulness of events on a web page, which are the main way that you can be notified of how the user is interacting with the web page. Nearly every jQuery operation you do is started through an event of some kind.

I'm also glad this book covers some topics often ignored, such as unit testing and organization of large projects. Many small projects eventually turn into large ones, and the advice in these chapters can help you to manage that growth in a way that reduces maintenance headaches.

The chapters building demo applications do a great job of showing how all the parts of jQuery fit together and demonstrate important concepts like templating that are central to all modern JavaScript frameworks and applications. Even today, I feel a bit amazed by demos like this showing it's possible to build something useful with very little code.

Aurelio De Rosa has been a contributor to the jQuery community for several years and is a member of jQuery's content team that ensures jQuery's online documentation is up-to-date. His work on this latest edition of *jQuery in Action* gives you timely information that reflects the most recent version of the library. Aurelio has also made jQuery's online documentation better in the process of writing this book by uncovering inconsistencies and missing information. You, as a reader of this book and soon-to-be jQuery developer, are the lucky beneficiary. Go forward and, "Write less, do more!"

DAVE METHVIN
PRESIDENT, JQUERY FOUNDATION

foreword to the first edition

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 when 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 plug-ins 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 Live-Query 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

It always astonishes me when I think about the amount of work and effort I put into this book. When the people at Manning approached me to write the third edition of *jQuery in Action*, I knew that it wasn't going to be a walk in the park, but I definitely underestimated the task. I thought, "This is going to be a piece of cake. A few months of work and I'll be done." Two years and many nights of work later, I don't regret my choice. Writing this book has been an incredible journey, one that has let me improve my skills in many different ways. I've become a better developer and a better writer, and I've improved my jQuery skills.

Two years ago, I was a web developer with a strong passion for jQuery, and I was grateful that this library solved so many problems for me for free. Before I started this project, my knowledge of jQuery was good, but without a doubt, writing and revising the chapters that you're about to read forced me to dive much deeper and, as a result, I was able to take my skills to the next level. I also had the opportunity to discover new issues regarding the library and its documentation. Revising this book allowed me to contribute to jQuery on a regular basis—so much that I've been invited to join the jQuery team. Needless to say, this has been an unexpected and very welcome achievement, and I'm proud to be part of such an amazing project.

Now that you know how I came to embark on this journey, let's tackle a crucial question: was this third edition really needed? I think it was, and this can be summed up with two basic facts. The previous edition of the book covers jQuery up to version 1.4, while the last version is 1.11, with jQuery 3 (also covered in this book) just around the corner. The second reason is that jQuery is definitely the most used JavaScript library out there. It's employed by 63% of the top one million sites in the world, and

by 17% of the internet. These two facts should lead you to understand that much has changed since the second edition of *jQuery in Action* was published, and that jQuery is not only still relevant, but isn't going to disappear any time soon.

In this third edition of the book, you'll see quite a few changes. First of all, I deleted the chapters about the jQuery UI because both jQuery and the jQuery UI have grown so much that they deserve a book of their own. In addition, as you'll see by turning the pages of this edition, I decided to add some advanced topics that weren't covered in the previous edition. Finally, I've introduced many new examples, lab pages, snippets of code, live demos, and much more to make this edition even better.

Turn this page, delve into the book, and start learning about the most-used JavaScript library in the world. Have fun!

<div align="right">AURELIO DE ROSA</div>

acknowledgments

offered invaluable contributions to the technical accuracy of the text and insightful comments, most of which are included in the volume you're holding in your hands (or the digital copy you're reading).

Sincere thanks to Dave Methvin for penning the foreword to this edition and endorsing my work, and to Bear Bibeault and Yehuda Katz for writing the two best-selling editions that preceded this one.

On a personal level, the most important person I'd like to thank is my soon-to-be-wife Annarita. Your love, patience, and sweetness have been crucial throughout this journey and not only this one. You complained not once during the two years I spent working on this project instead of doing something with you. Your support and understanding have been stunning and that's why I'm dedicating this book to you. You, my dear Annarita, give balance to my life. Thank you for all the lovely moments spent together and those yet to come. I love you.

Big thanks also go to my family: Raffaele, Eufemia, Giusy, Viola, my grandmothers Giuseppina and Anna, and my grandfather Aurelio. Thank you for all your love. You're partly responsible for who I am and what I've done. You have supported me as much as you could, and I owe you a lot.

I also want to thank Francesco Palladino. You're the best friend a person could have. You have always been there for me when I needed it. I wish you all the best life has to offer and may all your dreams come true.

And while I'm speaking about dreams, I also want to dedicate this book to all the people who have a burning passion and believe in their dreams. Don't stop believing in them because others tell you to, even if it's tough to keep going. One day, you'll achieve them. To all the dreamers out there, I wish you good luck.

I want to thank all the people who have contributed to form me and to shape the person that I have become, in one way or another: Albert Einstein, Ludwig van Beethoven, Lucius Annaeus Seneca, Roberto De Rosa, Leonardo Grisolia, and the anonymous umbrella seller.

Finally, I want to thank all the people on the jQuery team. If I've written a good book, it's because of the marvelous work you've been doing all these years. You rock!

AURELIO DE ROSA

about this book

This book is for web developers who want to delve into jQuery, the most popular and adopted JavaScript library on the internet. The goal is to guide you, the reader, through the path of becoming a pro of jQuery regardless of your starting level, beginner or advanced. This tome covers the whole library in depth, including some additional tools and frameworks such as Bower and QUnit, without forgetting to advocate best practices. Each API method is presented in an easy-to-digest syntax block that describes the parameters and the return value of the method.

jQuery in Action, Third Edition covers topics from the simple, such as what's jQuery and how to include it in a web page, to the advanced, such as the way the library implements Promises and how to create jQuery plugins. To help you in this journey, the content features many examples, three plugins, and three sample projects. It also includes what we called 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.

The book assumes a fundamental knowledge of HTML, CSS, and JavaScript. A previous knowledge of jQuery is not required but might come in handy to help you absorb the concepts faster.

Roadmap

We've divided the book into three parts: an introduction to jQuery and what it brings to the table, the jQuery core, where we cover all of its features, and a section on advanced topics.

Chapter 1 is about the philosophy behind jQuery and how it adheres to a principle called unobtrusive JavaScript. It discusses what jQuery is, what problems it tries to solve, and why you might want to employ it in your web projects.

Chapter 2 covers the selection of DOM elements via the use of selectors and how to create your own custom selectors. We'll also introduce you to the term *jQuery collection* (or *jQuery object*), which is used to refer to the JavaScript object returned by jQuery's methods. It contains the set of elements selected on which you can operate with the library.

Chapter 3 expands on chapter 2 by teaching how to refine or create a new selection of elements starting with a previous selection. You'll also learn how to create new DOM elements with jQuery.

Chapter 4 focuses on the many methods jQuery offers for working with attributes and properties, and what their differences are. Moreover, it explains how to store custom data on one or more DOM elements.

Chapter 5 is all about manipulating element class names, cloning and setting the content of DOM elements, and modifying the DOM tree by adding, moving, or replacing elements.

Chapter 6 introduces you to the various event models and how browsers allow you to establish handlers to control what happens when an event occurs. Then, we'll cover how jQuery allows developers to do the same thing while avoiding dealing with browser incompatibilities. In addition, the chapter describes important notions like *event delegation* and *event bubbling*.

Chapter 7 is different from the previous ones because its aim is to walk you through the development of a project, a DVD discs locator, where you can apply the lessons learned up to this point.

Chapter 8 examines the methods used to show and hide elements, and how you can create animations. Function queuing for serially running effects, as well as general functions, are also covered.

Chapter 9 is dedicated to utility functions, functions that are namespaced by jQuery that usually don't operate on DOM elements.

Chapter 10 covers one of the most important concepts of recent years: Ajax. We'll see how jQuery makes it almost brain-dead simple to use Ajax on web pages, shielding us from all the usual pitfalls, while vastly simplifying the most common types of Ajax interactions (such as returning JSON objects).

We set up a new challenge for you in chapter 11. Here, we'll tackle a real-world problem that many developers face: creating a contact form. The project consists of building a working contact form that doesn't require a complete reload of the page to inform the user about the failure or success in sending the message.

Chapter 12 is the first of part 3 where we move onto advanced topics, most of which are not strictly related to the core of the library. In this chapter, we'll discuss how to extend the functionality of jQuery by creating plugins for it. These plugins come in two flavors: methods and utility functions. In this chapter we'll examine both of them.

Chapter 13 explains how to avoid what's known as the *callback hell* by describing jQuery's implementation of Promises. As you'll learn, this is a delicate and controversial topic that has been the subject of discussions for years.

In chapter 14 we introduce you to testing, what it is, and why it's important. We'll focus our attention on one particular kind of testing: unit testing. Then, we'll cover QUnit, a JavaScript testing framework employed by some of the jQuery projects (jQuery, jQuery UI, and jQuery Mobile) to unit test the code.

Chapter 15, the last chapter of the book, starts with tips and tricks to improve the performance of code that uses jQuery by selecting elements the right way. Then, we'll broaden our focus to several tools, frameworks, and patterns not strictly related to jQuery but that can be used to craft fast, solid, and beautiful code. In particular, this chapter explains how to organize your code in modules, how to load modules with RequireJS, and how to manage front-end dependencies with Bower. Finally, we'll give you a taste of how jQuery fits into single-page applications by skimming the surface of Backbone.js.

To top it all off, we have provided an appendix highlighting key JavaScript concepts such as function contexts and closures—essential to make the most effective use of jQuery on our pages—for readers who are unfamiliar with, or would like a refresher on, these concepts.

Source code conventions and downloads

The source code in the book, whether in code listings or snippets, is in a `fixed-width font like this`, which sets it off from the surrounding text. In some listings, the code is annotated to point out key concepts, and numbered bullets are sometimes used in the text to provide additional information about the code. The code is formatted so that it fits within the available page space in the book by adding line breaks and using indentation carefully.

All of the source code for the examples in the book can be found at this GitHub link: https://github.com/AurelioDeRosa/jquery-in-action. The source code is also available for download from the publisher's website at www.manning.com/derosa/ or www.manning.com/jquery-in-action-third-edition.

Software requirements

The code examples for this book are organized in folders, one for each chapter, ready to be easily served by a local web server such as the Apache HTTP Server. With the exception of the projects built in chapters 7 and 10 and a few other ones, the examples don't require the presence of a web server and can be loaded directly into a browser for execution, if you so desire. The project in chapter 10 requires more back-end interaction than Apache can deliver, so running it locally requires setting up PHP for Apache.

All examples were tested in a variety of browsers, including Internet Explorer, Firefox, Safari, Opera, and Chrome.

Author Online

Purchase of *jQuery in Action, Third Edition* includes free access to a private web 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 the forum and subscribe to it, point your web browser to www.manning.com/derosa. This Author Online (AO) page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog among individual readers and between readers and 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 AO remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions, lest their interest stray!

The AO 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 cover illustration

The figure on the cover of *jQuery in Action, Third Edition* is captioned "The Watchman." The illustration is taken from a French travel book, *Encyclopédie des Voyages* by J. G. St. Saveur, published almost 200 years ago. 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 *Encyclopédie des Voyages* speaks vividly of the uniqueness and individuality of the world's towns and provinces just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by how they were speaking or what they were wearing.

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 collections such as this one.

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 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, Works.com, and a handful of other companies. Bear even served in the U.S. Military teaching infantry soldiers how to blow up tanks; skills that come in handy during those daily scrum meetings. Bear is currently a senior web developer for a leading provider of object storage software.

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, ever wedding videography), and helps to moderate JavaRanch.com as a "sheriff" (senior moderator). 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, 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.

AURELIO DE ROSA is a (full-stack) senior web developer with more than 5 years' professional experience programming for the web using the WAMP stack and HTML5, CSS3, Sass, JavaScript, and PHP. He's a member of the jQuery team and the JoindIn team, and an expert on JavaScript and HTML5 APIs. His interests also include web security, accessibility, performance, and SEO.

When not busy writing code, he's a regular writer, speaker, author of books, and coauthor of some academic papers.

Part 1

Starting with jQuery

If you're reading this page, it's because you've heard of jQuery from a fellow developer or read about it in a website or forum, and you're eager to understand what this library is all about. Maybe you're employing this library at work and you want to improve your skills to impress your boss. Or perhaps you've never heard about this jQuery thing and you were just captured by the very nice illustration on the cover of this book. Whatever the reason that brought you to open this book and read this page, the next chapter will (hopefully) give you all the explanations you need.

In the only chapter belonging to part 1, you'll learn more about what jQuery is, what problems it tries to solve, and why you might want to employ it in your web projects. In chapter 1 we'll teach you how to extricate yourself from the different versions of jQuery available and decide which one best fits your needs. If you're into web development and want to become a professional of the most used library in the world, proceed to chapter 1 and start the amazing journey that this book will be.

Introducing jQuery

This chapter covers

- What exactly jQuery is and why you should use it
- The unobtrusive JavaScript strategy
- Choosing the right version of jQuery
- Fundamental elements and concepts of jQuery

"There are only two kinds of languages: the ones people complain about and the ones nobody uses." How well this sentence from Bjarne Stroustrup, who designed and implemented C++, summarizes the sentiments about JavaScript. It, as well as several other languages (most notably PHP), was bemoaned as a "bad" language for several years. Then, something magical happened. Thanks to the rise of Ajax, the release of several libraries such as Prototype, Moo Tools, and jQuery, and the new, highly interactive web applications (which you might also have heard referred to as *single-page applications*), developers started understanding JavaScript's potential. Today JavaScript is also one of the most ubiquitous languages thanks to Node.js, a platform that allows you to use it as a server-side language, and PhoneGap, a framework for creating hybrid mobile applications.

jQuery is a free (licensed under the MIT License), popular JavaScript library, created by John Resig in 2006, that's designed to simplify the client-side scripting of HTML. As stated on the jQuery website,

> jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.

Although you might find this claim a bit self-promotional or presumptuous, it asserts nothing but the truth. jQuery has *really* changed the way millions of developers and designers write their code. Its use is so widespread that, according to the latest BuiltWith statistics (as of April 2015), jQuery is used by 63% of the top million websites (http://trends.builtwith.com/javascript/jQuery). The previously cited Moo Tools library, its nearest competitor, has a usage of just 3% (http://trends .builtwith.com/javascript/MooTools), while Prototype has a mere 2.5% (http:// trends.builtwith.com/javascript/Prototype).

jQuery is used by some of the most important companies and websites in the world, such as Microsoft, Amazon, Dell, Etsy, Netflix, Best Buy, Instagram, Fox News, GoDaddy, and many more. If you had any doubts about jQuery, this data should convince you that it's a stable and reliable library that you can use in your projects.

This book covers many aspects of the library starting from basic concepts, like selectors and the methods to traverse the Document Object Model (DOM), to more advanced ones, like extending the functionalities (creating plugins), improving the performances of your code, and testing. It assumes you have a minimal knowledge of JavaScript. If you need a refresher, take a look at the appendix. If you're unfamiliar with the language, you may find this text too tough, so we encourage you to study it and then come back. We'll wait here.

Are you back? Glad to see you again! Let's start from the beginning—that is, discussing what jQuery has to offer you and how it can help you in your web development process.

1.1   *Write less, do more*

jQuery's motto is "Write less, do more." If you've spent any time at all trying to add dynamic functionality to your pages, you've found that performing simple tasks using raw JavaScript can result in dozens of lines of code (LoC). The creator of jQuery specifically created this library to make common tasks trivial and easy to learn, solving issues caused by browser incompatibilities.

For example, anyone who has dealt with radio groups in JavaScript knows that it's a lesson in tedium to discover which radio element of a radio group is currently checked and to obtain its `value` attribute. The radio group needs to be located, and the resulting set of radio elements must be inspected, one by one, to find out which element has its `checked` attribute set. This element's `value` attribute can then be obtained.

To be compatible with Internet Explorer 6 and above (if you ignore some older browsers, a better approach exists), such code might be implemented as follows:

```
var checkedValue;
var elements = document.getElementsByTagName('input');
for (var i = 0; i < elements.length; i++) {
  if (elements[i].type === 'radio' &&
      elements[i].name === 'some-radio-group' &&
      elements[i].checked) {
    checkedValue = elements[i].value;
    break;
  }
}
```

Contrast that with how it can be done using jQuery:

```
var checkedValue =
  jQuery('input:radio[name="some-radio-group"]:checked').val();
```

Don't worry if that looks a bit cryptic right now. In short order, you'll understand how it works, and you'll be whipping up your own terse—but powerful—jQuery statements to make your pages come alive. The point we want to show here is how the library can turn a lot of lines of code into just one.

What makes the previous jQuery statement so short is the power of the *selector*, an expression used to identify target elements on a page. It allows you to easily locate and grab the elements that you need; in this case, the checked element in the radio group. If you haven't downloaded the example code yet, now would be a great time to do so. It can be obtained from a link on this book's web page at http://www.manning.com/derosa. Unpack the code and load into your browser the HTML page that you find in the file chapter-1/radio.group.html. This page, shown in figure 1.1, uses the jQuery statement that we just examined to determine which radio button has been checked.

This example shows you how simple and concise code written using jQuery can be. This isn't the only real power of jQuery; otherwise we could have thrown it out the window a long time ago. Nonetheless, one of its great strengths is the ability to retrieve elements using complex selectors without worrying about cross-browser compatibility, especially in older browsers.

When you perform a selection, you're relying on two things: a method and a selector. Today the latest versions of all major browsers support native methods for element selection like `document.querySelector()` and `document.querySelectorAll()`. They allow you to use more complex selectors instead of the usual selection by ID or class.

Figure 1.1 Determining which radio button is checked is easy to accomplish in one statement with jQuery!

In addition, the new CSS3 selectors are widely supported among modern browsers. If you aimed to support only modern browsers, and the capabilities of jQuery lay only in selecting elements, you would have enough power to avoid the overhead introduced by the library in your website. The fact that a lot of people still rely on older browsers, which you may have to support, can be a real pain because you have to deal with all the inconsistencies. This is one of the main reasons to employ jQuery. It allows you to reliably use its selectors without the worry of code not working in browsers that don't support them natively.

> **NOTE** If you're wondering what browsers are considered modern today, they are Internet Explorer 10 and above and the latest versions of Chrome, Opera, Firefox, and Safari.

Still not convinced? Here's a list of issues that you'll have to tackle on your own if you don't use jQuery: http://goo.gl/eULyPT. In addition, as we outlined, the library is much more than that, as you'll discover in the rest of the book.

Let's now examine how JavaScript should be used on your pages.

1.2   *Unobtrusive JavaScript*

You may recall the bad-old days before CSS, when you were forced to mix stylistic markup with the document structure markup in your HTML pages. Anyone who's been authoring pages for any amount of time surely does, most likely with less than fondness.

The addition of CSS to your web development toolkits allows you to separate stylistic information from the document structure and gives travesties like the `<font>` tag the well-deserved boot. Not only does the separation of style from structure make your documents easier to manage, but it also gives you the versatility to completely change the stylistic rendering of a page by swapping out different style sheets.

Few of you would voluntarily regress to the days of applying styles with HTML elements, yet markup such as the following is still all too common:

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

You can easily see that the style of this `button` element isn't applied via the use of the `<font>` tag and other deprecated style-oriented markup. It's determined by whatever, if any, CSS rules (not shown here) that are in effect on the page. Although this declaration doesn't mix *style* markup with structure, it does mix *behavior* with structure. It includes the JavaScript to be executed when the button is clicked as part of the markup of the `button` element via the `onclick` attribute (which, in this case, changes the color of a DOM element with the ID value of `xyz` into red). Let's examine how you might improve this situation.

1.2.1 Separating behavior from structure

For all the same reasons that it's desirable to segregate style from structure within an HTML document, it's just as beneficial (if not more so) to separate the *behavior* from the structure. Ideally, an HTML page should be structured as shown in figure 1.2, with structure, style, and behavior each partitioned nicely in its own niche.

This strategy, known as *unobtrusive JavaScript*, is now embraced by every major JavaScript library, helping page authors achieve this useful separation on their pages. As the library that popularized this movement, jQuery's core is well optimized for producing unobtrusive JavaScript easily. Unobtrusive JavaScript considers *any* JavaScript expressions or statements placed within or among HTML tags in the <body> of HTML pages, either as attributes of HTML elements (such as onclick) or in script blocks placed anywhere other than the very end of the body of the page, to be incorrect.

```
<html>
  <head>

    ┌─────────────────────────────┐
    │            STYLE            │
    │     Local style elements    │
    │   and imported style sheets │
    └─────────────────────────────┘

  </head>
  <body>
    ┌─────────────────────────────┐
    │          STRUCTURE          │
    │      HTML structural        │
    │          elements           │
    └─────────────────────────────┘
    ┌─────────────────────────────┐
    │          BEHAVIOR           │
    │     Local script elements   │
    │   and imported script files │
    └─────────────────────────────┘
  </body>
</html>
```

Figure 1.2   With structure, style, and behavior each neatly tucked away within a page, readability and maintainability are maximized.

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

```
<button id="test-button">Click Me</button>
```

Much simpler! But now, you'll note, the button doesn't **do** anything. You can click it all day long, and no behavior will result. Let's fix that.

1.2.2 Segregating the script

Rather than embedding the button's behavior in its markup, you'll segregate the script by moving it to a script block. Following the current best practices, you should place it at the bottom of the page before the closing body tag (</body>):

```
<script>
  document.getElementById('test-button').addEventListener(
    'click',
    function() {
      document.getElementById('xyz').style.color = 'red';
    },
    false
  );
</script>
```

Because you're placing the script at the bottom of the page, you don't need to use a handler attached to the onload event of the window object, like developers (erroneously) use to do in the past, or wait for the DOMContentLoaded event, which is only

available in modern browsers. The DOMContentLoaded event is fired when the HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and so on to finish loading. The load event is fired when an HTML page and its dependent resources have finished loading (we'll return to this topic in section 1.5.3). By placing the script at the bottom of the page, when the browser parses the statement, the button element exists because its markup has been parsed, so you can safely augment it.

> **NOTE**   For performance reasons, script elements should always be placed at the bottom of the document body. The first reason is to allow progressive rendering, and the second is to have greater download parallelization. The motivation behind the first is that rendering is blocked for all content below a script element. The reason behind the second is that the browser won't start any other downloads, even on a different hostname, if a script element is being downloaded.

The previous snippet is another example of code that isn't 100% compatible with the browsers your project might be targeting. It uses a JavaScript method, addEventListener(), that's not supported by Internet Explorer 6–8. As you'll learn later on in this book, jQuery helps you in solving this problem, too.

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

But none of that is bad; anything that persuades you to write your 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, that is.

jQuery is specifically focused on the task of making it easy and delightful for you to code your pages using unobtrusive JavaScript techniques, without paying a hefty price in terms of effort or code bulk. You'll find that making effective use of jQuery will enable you to accomplish much more on your pages while writing *less* code. The motto is still "Write less, do more," isn't it? Without further ado, let's start looking at how jQuery makes it so easy for you to add rich functionality to your pages without the expected pain.

1.3   *Installing jQuery*

Now that you know what jQuery is and what it can do for you, you need to download the library to start getting your hands dirty. To download it, please visit the page http:// jquery.com/download/. Once there, you'll probably be overwhelmed by the plethora of options available. Branch 1.x, 2.x, or 3.x? Compressed or uncompressed? Download it or use a content delivery network (CDN)? Which one to choose depends on several factors. To make a conscious choice, let's uncover the differences.

1.3.1 Choosing the right version

In April 2013, the jQuery team introduced version 2.0 with the intention of looking at the future of the web instead of its past, especially from the browser's perspective. Until that point, jQuery supported all of the latest versions of Chrome, Firefox, Safari, Opera, and Internet Explorer starting from version 6. With the introduction of version 2.0, the team decided to leave behind the older Internet Explorer 6, 7, and 8 browsers to focus on the web as it will be, not as it was.

This decision caused the deletion of a bunch of code created to solve browser incompatibilities and missing features in those prehistoric browsers. The fulfillment of this task resulted in a smaller (-12%) and faster code base. Although 1.x and 2.x are two different branches, they have a strict relation. There's feature parity between jQuery version 1.10 and 2.0, version 1.11 and 2.1, and so on.

In October 2014, Dave Methvin, the president of the jQuery Foundation (the foundation that takes care of jQuery and other projects—https://jquery.org/), published a blog post (http://blog.jquery.com/2014/10/29/jquery-3-0-the-next-generations/) where he publicly announced the plan to release a new major version of jQuery: jQuery 3. In the same way version 1.x supports old browsers while 2.x targets modern browsers, jQuery 3 is split into two versions. jQuery Compat 3 is the successor of 1.x, whereas jQuery 3 is the successor of 2.x. He further explained:

> We'll also be re-aligning our policy for browser support starting with these releases. The main jQuery package remains small and tight by supporting the evergreen browsers (the current and previous versions of a specific browser) that are common at the time of its release. We may support additional browsers in this package based on market share. The jQuery Compat package offers much wider browser support, but at the expense of a larger file size and potentially lower performance.

With the new version, the team also took the opportunity to drop the support for some browsers, fix many bugs, and improve several features.

The first factor to consider when deciding which version to use is which browsers your project must support. Table 1.1 describes the browsers supported by each major version of jQuery.

Table 1.1 An overview of the browsers supported by the major versions of jQuery

Browsers	jQuery 1	jQuery 2	jQuery Compat 3	jQuery 3
Internet Explorer	6+	9+	8+	9+
Chrome	Current and previous	Current and previous	Current and previous	Current and previous
Firefox	Current and previous	Current and previous	Current and previous	Current and previous
Safari	5.1+	5.1+	7.0+	7.0+
Opera	12.1x Current and previous	12.1x Current and previous	Current and previous	Current and previous

Table 1.1   An overview of the browsers supported by the major versions of jQuery *(continued)*

Browsers	jQuery 1	jQuery 2	jQuery Compat 3	jQuery 3
iOS	6.1+	6.1+	7.0+	7.0+
Android	2.3 4.0+	2.3 4.0+	2.3 4.0+	2.3 4.0+

As you can see from the table, there's a certain degree of overlap in regard to the browser versions supported. But keep in mind that what's referred to as "Current and previous" (meaning the current and preceding version of a browser at the time a new version of jQuery is released) changes based on the release date of the new version of jQuery.

Another important factor to base your decision on is where you'll use jQuery. Here are some use cases that can help you in your choice:

- Websites that don't need to support older versions of Internet Explorer, Opera, and other browsers can use branch 3.x. This is the case for websites running in a controlled environment such as a company local network.
- Websites that need to target an audience as wide as possible, such as a government website, should use branch 1.x.
- If you're developing a website that needs to be compatible with a wider audience but you don't have to support Internet Explorer 6–7 and old versions of Opera and Safari, you should use jQuery Compat 3.x.
- If you don't need to support Internet Explorer 8 and below, but you have to support old versions of Opera and Safari, you should use jQuery 2.x.
- Mobile apps developed using PhoneGap or similar frameworks can use jQuery 3.x.
- Firefox OS or Chrome OS apps can use jQuery 3.x.
- Websites that rely on very old plugins, depending on the actual code of the plugins, may be forced to use jQuery 1.x.

In summary, two of the factors are where you're going to use the library and which browsers you intend to support.

Another source of confusion could be the choice between the compressed (also referred to as *minified*) version, intended for the production stage, or the uncompressed version, intended for the development stage (see the comparison in figure 1.3). The advantage of the minified library is the reduction in size that leads to bandwidth savings for the end users. This reduction is achieved by removing the useless spaces (*indentation*), removing the code's comments that are useful for developers but ignored by the JavaScript engines, and shrinking the names of the variables (*obfuscation*). These changes produce code that's harder to read and debug—which is why you shouldn't use this version in development—but smaller in size.

```
       // Handle when the DOM is ready
       ready: function( wait ) {

               // Abort if there are pending holds or we're already ready
               if ( wait === true ? --jQuery.readyWait : jQuery.isReady ) {
                       return;
               }

               // Make sure body exists, at least, in case IE gets a little overzealous (ticket #5443).
               if ( !document.body ) {
                       return setTimeout( jQuery.ready );
               }

               // Remember that the DOM is ready
               jQuery.isReady = true;

               // If a normal DOM Ready event fired, decrement, and wait if need be
               if ( wait !== true && --jQuery.readyWait > 0 ) {
                       return;
               }

               // If there are functions bound, to execute
               readyList.resolveWith( document, [ jQuery ] );

               // Trigger any bound ready events
               if ( jQuery.fn.triggerHandler ) {
                       jQuery( document ).triggerHandler( "ready" );
                       jQuery( document ).off( "ready" );
               }
       }
```

Uncompressed

```
ready:function(a){if(a===!0?!--m.readyWait:!m.isReady){if(!y.body)return
setTimeout(m.ready);m.isReady=!0,a!==!0&&--m.readyWait>0||(H.resolveWith(y,
[m]),m.fn.triggerHandler&&(m(y).triggerHandler("ready"),m(y).off("ready")))}}
```

Compressed

Figure 1.3 At the top, a snippet taken from the jQuery's source code that shows you the uncompressed version format. At the bottom, the same snippet minified to be used in production.

In this book we'll use jQuery 1.x as a base to let you test your code in the widest range of possible browsers, but we'll highlight all the differences introduced by jQuery 3 so that your knowledge will be as up to date as possible.

Choosing the right version of jQuery is important, but we also cited the difference between hosting jQuery locally or using a CDN.

1.3.2 *Improving performances using a CDN*

Today it's common practice to serve files like images and libraries through a *content delivery network* to improve the performance of a website. A CDN is a distributed system of servers created to offer content with high availability and performance. You might be aware that browsers can download a fixed set of contents, usually from four to eight files, from a host at the same time. Because the files served using a CDN are provided from a different host, you can speed up the whole loading process, increasing the number of files downloaded at a time. Besides, a lot of today's websites use CDNs, so there's a higher probability that the required library is already in the user's browser cache. Employing a CDN to load jQuery doesn't guarantee better performance in every situation because there are many factors that come into play. Our advice is to test which configuration best suits your specific case.

Nowadays there are several CDNs you can rely on to include jQuery, but the most reliable are the jQuery CDN (http://code.jquery.com), the Google CDN (https://developers.google.com/speed/libraries/devguide), and the Microsoft CDN (http://www.asp.net/ajaxlibrary/cdn.ashx).

Let's say you want to include the compressed version of jQuery 1.11.3 using the jQuery CDN. You can do that by writing the following code:

```
<script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
```

As you may have noticed, this code doesn't specify the protocol to use (either HTTP or HTTPS). Instead, you're specifying the same protocol used in your website. But keep in mind that using this technique in a page that doesn't run on a web server will cause an error.

Using a CDN isn't all wine and roses, though. No server or network has 100% uptime on the internet, and CDNs are no exception. If you rely on a CDN to load jQuery, in the rare situations where it's down or not accessible and the visitor's browser doesn't have a cached copy, your website's code will stop working. For critical applications this can be a real problem. To avoid it, there's a simple and smart solution you can adopt, employed by a lot of developers. Once again, you want to include the minified version of jQuery 1.11.3, but now you'll use this smart solution:

```
<script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
<script>window.jQuery || document.write('<script src="javascript/jquery-
    1.11.3.min.js"><\/script>');</script>
```

The idea behind this code is to request a copy of the library from a CDN and check if it has been loaded, testing whether the jQuery property of the window object is defined. If the test fails, you inject a code that will load a local hosted copy that, in this specific example, is stored in a folder called javascript. If the jQuery property is present, you can use jQuery's methods safely without the need to load the local hosted copy.

You test for the presence of the jQuery property because, once loaded, the library adds this property. In it you can find all the methods and properties of the library. During the development process, we suggest that you use a local copy of jQuery to avoid any connectivity problems.

In addition to the jQuery property, you'll also find a shortcut called $ that you'll see a lot in the wild and in this book. Although it may seems odd, in JavaScript a variable or a property called $ is allowed. We called $ a shortcut because it's actually the same object of jQuery as proved by this statement taken from the source code:

```
window.jQuery = window.$ = jQuery;
```

So far, you've learned how to include jQuery in your web pages but you know nothing about how it's structured. We'll look at this topic in the next section.

1.4   How jQuery is structured

The jQuery repository (https://github.com/jquery/jquery), hosted on GitHub, is a perfect example of how front-end development has changed over the past years. Although not strictly related to the use of the library itself, it's always important to know how expert developers organize their workflow and the tools they employ.

If you're an experienced front-end developer, chances are you're already aware of some, if not all, of these tools, but a refresher is always worthwhile. The development team adopted the latest and coolest technologies in today's front-end panorama for the development of jQuery, specifically these:

- *Node.js* (http://nodejs.org)—A platform built on Chrome's JavaScript runtime that enables you to run JavaScript as a server-side language.
- *npm* (https://npmjs.org)—The official package manager for Node.js used to install packages like Grunt and its tasks.
- *Grunt* (http://gruntjs.com)—A task runner to automate common and repetitive tasks such as building, testing, and minification.
- *Git* (http://git-scm.com)—A free, distributed version control system to keep track of changes in the code. It allows easy collaboration between developers.

On the other hand, jQuery's source code follows the asynchronous module definition (AMD) format. The AMD format is a proposal for defining modules where both the module and its dependencies can be asynchronously loaded. In practice, this means that although you use jQuery as a unique, single block, its source is split into several files (modules), as shown in figure 1.4. The dependencies relative to these files are managed through the use of a dependencies manager—in this case, RequireJS.
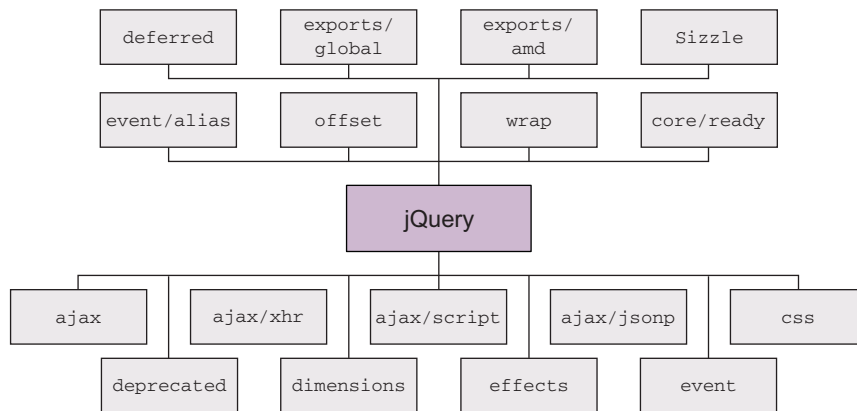
Figure 1.4   A schema representing jQuery's modules: `ajax`, `ajax/xhr`, `ajax/script`, `ajax/jsonp`, `css`, `deprecated`, `dimensions`, `effects`, `event`, `event/alias`, `offset`, `wrap`, `core/ready`, `deferred`, `exports/global`, `exports/amd`, and `Sizzle`

To give you an idea of what's inside the modules, here are some examples:

- `ajax`—Contains all the Ajax functions like `ajax()`, `get()`, and `post()`.
- `deprecated`—Contains all the currently deprecated methods that haven't been removed. What's inside this module depends on the jQuery version.
- `effects`—Contains the methods that allow animations like `animate()` and `slideUp()`.
- `event`—Contains the methods to attach event handlers to browser events like `on()` and `off()`.

The organization into modules of the source leads to another advantage: the possibility of building a custom version of jQuery containing only the modules you need.

1.4.1   *Save space creating your own custom build*

jQuery offers you the possibility of building your own custom version of the library, containing only the functionalities you need. This allows you to reduce the weight of your library, which will lead to a performance improvement because the end user has fewer KBs to download.

The ability to eliminate the modules you don't need is important. Although you might think that you'll need all the power that jQuery brings to the table, it's doubtful that you'll use all of its functions in the same website. Why not remove those useless lines of code to improve the performance of your website?

You can use Grunt to create a custom version. Imagine that you need a minified version of jQuery 1.11.3 with all the functionalities (except the deprecated methods and properties) and the effects. To perform this task, you need to install Node.js, Git, and Grunt on your local machine. After installing them, you have to clone jQuery's repository by running the following command using the command-line interface (CLI):

```
git clone git://github.com/jquery/jquery.git
```

Once the cloning process is complete, enter these last two commands:

```
npm install
grunt custom:-deprecated,-effects
```

You're finished! Inside the folder named dist you'll find your custom jQuery build in both minified and non-minified versions.

This approach doesn't come without drawbacks, though. The first issue arises when a new version of jQuery is released. The second arises when a new functionality of your website requires a feature contained in a module that wasn't previously included. In these cases, you need to again perform the steps described previously (usually only the commands) to create a new custom version that includes the new methods, bug fixes, or the missing module.

Now that you know how to put the library in place and how to create a custom build, it's time to delve into jQuery's fundamentals.

1.5   jQuery fundamentals

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

jQuery places a high priority on ensuring that code will work consistently across all major browsers; many of the harder JavaScript problems have been silently solved for you. Should you find that the library needs a bit more juice, jQuery has a simple but powerful way for extending its functionality via plugins, which we'll discuss in detail in chapter 12.

Let's start by taking a look at the `jQuery` object itself and how you can use your CSS knowledge to produce powerful yet terse code.

1.5.1   Properties, utilities, and methods

As we said before, the jQuery library is exposed through a property called `jQuery` and a shortcut called `$`. Using them gives you access to the properties, methods, and functions that jQuery provides.

One of the properties exposed by the `jQuery` property is `fx.off`. It allows enabling or disabling effects executed using jQuery's methods. We'll discuss this and other properties in detail in chapter 9.

Much more exciting are the *utilities*, also referred to as *utility functions*. You can think of them as a handful of commonly used, general-purpose functions that are included in the library. You could say that jQuery acts as a *namespace* for them.

To give you a basic idea of what they are, let's look at an example. One of the utilities available is the function for trimming strings. Its aim is to remove whitespaces from the beginning and the end of a string. A call to it could look like this:

```
var trimmed = $.trim(someString);
```

If the value of `someString` is " hello ", the result of the `$.trim()` call will be "hello". As you can see, in this example we used the jQuery shortcut (`$`). Remember that it's an identifier like any other in JavaScript. Writing a call to the same function using the `jQuery` identifier, rather than its alias, will result in this code:

```
var trimmed = jQuery.trim(someString);
```

Another example of a utility function is `$.isArray()`, which, as you may guess, tests if a given argument is an array.

In addition to properties and functions, the library also exposes methods that are available once you call the `jQuery()` function. Let's learn more.

1.5.2   The jQuery object

The first function you'll use in your path to learn jQuery is `jQuery()`. It accepts up to two arguments, and, depending on their number and type, performs different tasks.

Like many other (almost all) methods in the library, it allows for *chaining*. Chaining is a programming technique used to call several methods in a single statement. Instead of writing

```
var obj = new Obj();
obj.method();
obj.anotherMethod();
obj.yetAnotherMethod();
```

you can write

```
var obj = new Obj();
obj.method().anotherMethod().yetAnotherMethod();
```

The most common use of `jQuery()` is to select elements from the DOM so you can apply some changes to them. In this case, it accepts two parameters: a selector and (optionally) a context. This function returns an object containing a collection of DOM elements that match the given criteria. But what's a selector?

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 to use selectors, which concisely represent elements based on their type, attributes, or position within the HTML document. Those familiar with XML might be familiar with XPath (more on this here: http://www.w3.org/TR/xpath20/) as a means to select elements within an XML document. CSS selectors represent an equally powerful concept but are tuned for use within HTML pages, are a bit more concise, and are generally considered easier to understand.

jQuery makes use of the same selectors as CSS. It supports not only the widely implemented ones belonging to CSS2.1 but also the more powerful selectors defined in CSS3. This is important because some of them may not be fully implemented by all browsers or may never make their appearance (for example, in older versions of Internet Explorer). As if this were not enough, jQuery also has its own selectors and allows you to create your own custom selectors.

In this book you'll be able to use your existing knowledge of CSS to get up and running fast, and then you'll learn about the more advanced selectors that jQuery supports. If you have little knowledge of them, don't worry. We'll cover jQuery selectors in great detail in chapter 2, and you can find a full list of them on the jQuery site at http://api.jquery.com/category/selectors/.

Let's say you want to select all the <p>s in the page using `jQuery()`. To do this, you can write

```
var paragraphs = jQuery('p');
```

The library searches for matching elements within the DOM starting from the document root, so for a huge number of elements the process can be slow.

In most cases, you can speed up the search using the `context` parameter. It's used to restrict the process to one or more subtrees, depending on the selector used. To understand it, you'll modify the previous example.

Let's say that you want to find all the `<p>`s contained in a `<div>`. *Contained* doesn't mean the `<div>` must be the parent of the `<p>`; it can also be a generic ancestor. You can achieve this task as shown below:

```
var paragraphsInDiv = jQuery('p', 'div');
```

Using the jQuery alias, the same statement will look like this:

```
var paragraphsInDiv = $('p', 'div');
```

When you use the second argument, jQuery first collects elements based on this selector called `context` and then retrieves the descendants that match the first parameter, `selector`. We'll discuss this topic in more detail in chapter 2.

As we said, the `jQuery()` function (and its alias `$()`) returns a JavaScript object containing a set of DOM elements that match the selector, in the order in which they're defined within the document. This object possesses a large number of useful predefined methods that can act on the collected group of elements. We'll use the term *jQuery collection*, *jQuery object*, or *jQuery set* (or other similar expressions) to refer to this returned JavaScript object that contains the set of matched elements that can be operated on with the methods defined by jQuery. Based on this definition, the previous `paragraphsInDiv` variable is a jQuery object containing all the paragraphs that are descendants of a `div` element. You'll use jQuery objects extensively when you need to perform operations, like running a certain animation or applying a style, on several elements in the page.

As mentioned earlier, one important feature of a large number of these methods, which we often refer to as *jQuery methods*, is that they allow for chaining. After a method has completed its work, it returns the same group of elements it acted on, ready for another action. As things get progressively more complicated, making use of jQuery's chainability will continue to reduce the lines of code necessary to produce the results you want.

In the previous section, we highlighted the advantages of placing the JavaScript code at the bottom of the page. For many years now, developers have placed the `scripts` elements in the `<head>` of the page, relying on a jQuery method called `ready()`. This approach is now discouraged, but many developers still use it. In the next section you'll learn more about it and also discover what the suggested approach is today.

1.5.3 *The document ready handler*

When embracing unobtrusive JavaScript, behavior is separated from structure. Applying this principle, you perform operations on the page elements outside the document markup that creates them. In order to achieve this, you need a way to wait until the DOM elements of the page are fully realized before those operations execute.

In the radio group example, the entire body has to be loaded before the behavior 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 this:

```
window.onload = function() {
  // do stuff here
};
```

This causes the defined code to execute *after* the document has fully loaded. Unfortunately, the browser not only delays executing the `onload` code until after the DOM tree is created but also waits until all external resources are fully loaded and the page is displayed in the browser window. This includes resources like images as well as Quick-Time and Flash videos embedded in web pages. As a result, visitors can experience a serious 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 significant time to load, visitors will have to wait for the image loading to complete before the rich behaviors become available. This could make the whole unobtrusive JavaScript proposition a nonstarter 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 resulting DOM tree before executing the script to apply the rich behaviors. Accomplishing this in a cross-browser manner that takes into account older browsers is somewhat difficult, but jQuery provides a simple means to trigger the execution of code once the DOM tree has loaded (without waiting for external resources).

The formal syntax to define such code is as follows:

```
jQuery(document).ready(function() {
  // Your code goes here...
});
```

First, you wrap the `document` object using the `jQuery()` function, and then you call the `ready()` method, passing a function to be executed when the document is ready to be manipulated. This means that inside the function passed to `ready()` you can safely access all of the elements of your page. A schema of the mechanism described is shown in figure 1.5.

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

```
jQuery(function() {
  // your code hoes here...
});
```

By passing a function to `jQuery()` or its alias `$()`, you instruct the browser to wait until the DOM has fully loaded
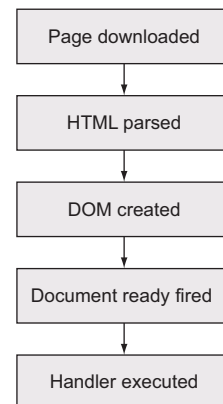
Figure 1.5   **A representation of the steps performed by browsers before the document-ready handler is executed**

(but only the DOM) before executing the code. Even better, you can use this technique multiple times within the same HTML document, and the browser will execute all of the functions you specify in the order in which they're 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 included third-party code uses the `onload` mechanism for its own purpose (not a best-practice approach).

Using the document-ready handler is a good way to embrace the unobtrusive JavaScript technique, but its use isn't mandatory and can be avoided.

Because `ready()` takes care to execute the code after the DOM is loaded, developers used to place the `script` elements in the `<head>` of the page. As we discussed in section 1.2.2, "Segregating the script," you can place them just before the closing body tag (`</body>`). By doing so, you can completely avoid the use of `$(document)` `.ready()` because at that point all of the other elements are already in the DOM. Therefore, you can retrieve and use them safely. If you want to see an example of how `$(document).ready()` can be avoided, take a look at the source code of the file chapter-1/radio.group.html.

In the remainder of this book we'll stick with the current best practices, so you won't use `ready()`.

1.6 Summary

We've covered a great deal of material in this whirlwind introduction to jQuery. To summarize, it's generally useful for any page that needs to perform anything but the most trivial of JavaScript operations. It's 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 on the parameters passed to it.

We mentioned how well the repository of the library and the code in general are organized. We also paid great attention to the several available versions of the library and their differences in order to be able to make a conscious choice. Performance is an important factor to consider, so we described the possibilities you have to reduce the added overhead to a minimum by including a library in your pages. Using CDNs and customizing the modules that you want are an amazing way to speed up the download of jQuery.

In the chapters that follow, we'll explore all the features that jQuery has to offer you as a web developer. We'll begin our tour in the next chapter as you learn how to use jQuery selectors to quickly and easily identify the elements that you wish to act on.

Part 2

Core jQuery

Many years have passed since John Resig presented jQuery to the world. Fewer but still quite a few years are behind us since jQuery was only a library to manipulate the DOM. During this time jQuery has created an entire ecosystem around itself consisting of companion libraries and other projects such as these:

- *jQuery UI*—A library consisting of a set of user interface interactions, effects, widgets, and themes to help you create amazing user interfaces
- *jQuery Mobile*—An HTML5-based user interface system for all popular mobile device platforms, to help you create beautiful designs for mobile devices
- *QUnit*—A JavaScript unit-testing framework used by all the other jQuery projects
- *Plugins*—The plugins published on npm (https://www.npmjs.com/) and the myriad of other plugins spread across the web that people have created to cover those use cases not covered by jQuery or to improve its functionalities

In part 2 of this book, we'll cover the core library from stem to stern. When you finish these chapters, you'll thoroughly know the jQuery library and be ready to tackle any web project armed with one of the most powerful client-side tools available. So turn the page, dig in, and get ready to learn how to breathe life into your web applications in a way that's not only easy but fun!

Selecting elements

2

This chapter covers

- Selecting elements with jQuery by using CSS selectors
- Discovering the unique jQuery-only filters
- Developing custom filters
- Learning the `context` parameter of the `jQuery()` function

In this chapter, we'll examine in great detail how the DOM elements to be acted upon are identified by looking at one of the most powerful and frequently used capabilities of jQuery's `$()` function: the selection of DOM elements via *selectors.* Throughout the pages of this chapter, you'll become familiar with the plethora of selectors available. jQuery not only provides full support for all the CSS selectors but also introduces other ones. We'll also introduce you to *filters,* many of which are special jQuery-only selectors that usually work with other types of selectors to further reduce a set of matched elements. As if this weren't enough, you'll learn how to create *custom filters* (also referred to as *custom selectors* or *custom pseudo-selectors*) in case your pages need one the library doesn't support. We'll also discuss `context`, the second parameter of the `$()` function, and describe the implications of its use.

A good number of the capabilities required by interactive web 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. This and the next chapter provide you with the concepts to select elements. In the previous edition of this book, they were a unique chapter because their contents are highly related, but we decided to split them to help you digest the huge number of concepts described. Note that, despite the split, this chapter is still pretty long and terse. You may expect to go through it several times before mastering all its concepts. With this last note in mind, let's begin our detailed tour of the many ways that jQuery lets you specify which elements are to be targeted for manipulation.

2.1   *Selecting elements for manipulation*

The first thing you need to do when using virtually any jQuery method is to select some document elements to act upon. As you learned in chapter 1, to select elements in a page using jQuery, you need to pass the selector to the `jQuery()` function (or its alias `$()`). The `jQuery()` function and its alias return a jQuery object containing a set of DOM elements that match the given criteria and also expose many of jQuery's methods and properties.

Sometimes the set of elements you 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 `list-element`, contain a link, and are first in the list." Fortunately, jQuery provides a robust selector syntax you can use to easily specify sets 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 means to perform both common and complex selections.

To help you learn about element selection, we've put together a jQuery Selectors Lab Page that's available in the downloadable code examples for this book (in the file chapter-2/lab.selectors.html). The Selectors Lab allows you to enter a jQuery selector string and see (in real time!) which DOM elements get selected. When displayed, the lab should look as shown in figure 2.1.

> **TIP**   If you haven't yet downloaded the example code, you really ought to do so now—the information in this chapter will be much easier to absorb if you follow along with the lab exercises. Visit this book's web page at http:// www.manning.com/derosa to find the download link, or go to https://github .com/AurelioDeRosa/jquery-in-action.

The Selector Panel at the 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.

The selector that you type (in this case `li`) is applied to the HTML fragment loaded into the DOM Sample pane at the upper right. The lab code that executes when you click Apply adds a class named `found-element` to all the matching elements. A CSS declaration defined for the page causes all elements with that class to be highlighted

Figure 2.1 The jQuery Selectors Lab Page allows you to observe the behavior of any selector you choose in real time.

with a black border and gray background. After clicking Apply, you should see the display shown in figure 2.2, in which all `li` elements in the DOM sample are highlighted. In addition, the executed jQuery statement, as well as the tag names of the selected elements, is displayed below the Selector text box. The HTML markup used to render the DOM sample fragment is displayed in the lower pane, labeled "DOM Sample Code." This should help you experiment with writing selectors targeted at the elements in this sample.

Matched elements
highlighted

jQuery
command
executed
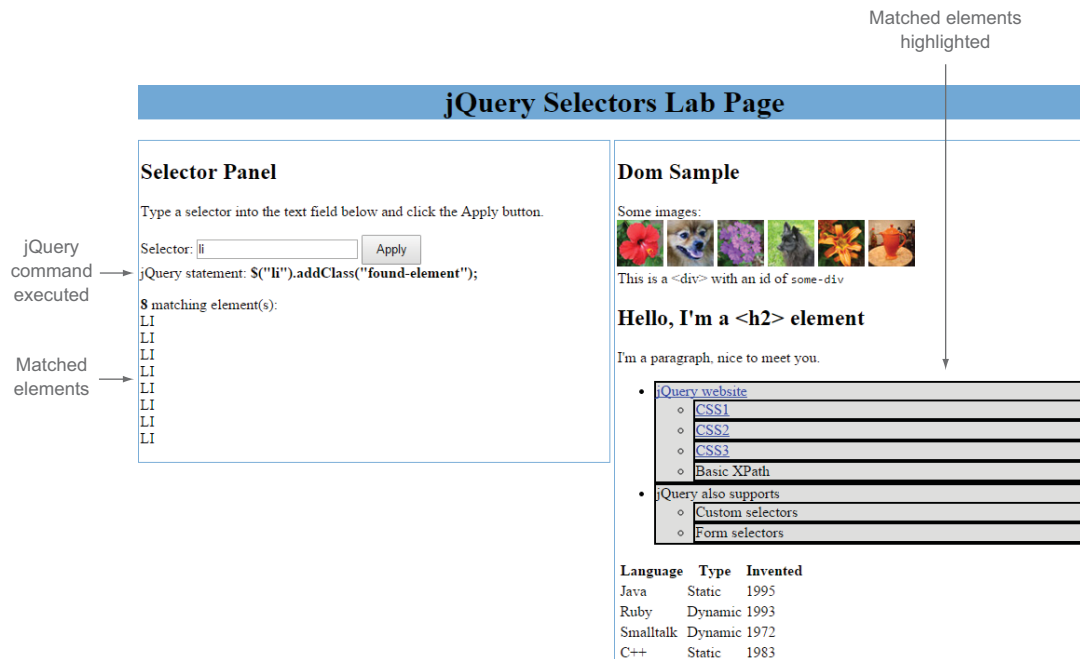
Matched
elements

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

We'll talk more about using this lab as we progress through the chapter. For the moment, let's take a look at how jQuery deals with the basic CSS selectors.

2.2   Basic selectors

For applying styles to page elements, web developers have become familiar with a small but useful group of selection expressions that work across all browsers. Those expressions can select by an element's ID, by CSS class names, and by tag names. A special case of selecting elements by tag name is the Universal selector, which allows you to select all the page elements within the DOM. The selection expressions enable you to perform basic searches in the DOM, and we'll provide the details in the following sections. When combined, these selectors allow you to achieve slightly more complicated selections. Table 2.1 provides a quick refresher of these selectors and how you can combine them.

Table 2.1   Some simple CSS selector examples

Example	Description	In CSS?
`*`	Matches all the elements in the page	✓
`#special-id`	Matches the element with the ID value of `special-id`	✓
`.special-class`	Matches all elements with the class `special-class`	✓

Table 2.1 Some simple CSS selector examples *(continued)*

Example	Description	In CSS?
`a`	Matches all anchor (`a`) elements	✓
`a.special-class`	Matches all anchor (`a`) elements that have the class `special-class`	✓
`.class.special-class`	Matches all elements with the class `class` and class `special-class`	✓

In JavaScript, you have a set of functions, such as `getElementById()` and `get-ElementsByClassName()`, that are designed to work with a specific type of selector to retrieve DOM elements to act upon. Unfortunately, you might have some problems using even such simple functions. For example, `getElementsByClassName()` isn't supported in versions of Internet Explorer prior to 9. If you want to use only native methods, you should pay attention to cross-browser compatibilities.

jQuery to the rescue! If the browser supports the selector or the function natively, jQuery will rely on it to be more efficient; otherwise it'll use its methods to return the expected result. The good news is that you don't have to worry about this difference. jQuery will do its work for you behind the scenes, so you can focus on other aspects of your code.

The jQuery library is fully CSS3 compliant, so selecting elements will present 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. The library does *not* depend on 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 practice, play with the Selectors Lab and run some experiments with some basic CSS selectors until you feel comfortable with them.

Happy to know that jQuery will solve all the cross-browser compatibilities (for the supported browsers) for us, we can now delve into the plethora of selectors available.

2.2.1 *The All (or Universal) selector*

The first selector available is the All (or Universal) selector, which is represented by an asterisk (*). As the name suggests, it allows you to retrieve all of the DOM elements of a web page, even the `head` element and its children. To reinforce this concept, let's say you have the following HTML page:

```
<!DOCTYPE html>
<html>
  <head>
    <title>jQuery in Action, 3rd edition</title>
  </head>
  <body>
```

```
    <p>I'm a paragraph</p>
  </body>
</html>
```

To retrieve all the elements of the page you need to use the Universal selector and pass it to the `jQuery()` function (or its alias `$()`) in a statement like the following:

```
var allElements = $('*');
```

Before moving on, there's an established convention we want to mention. When saving the result of a selection made with jQuery in a variable, a widely adopted convention is to prepend or (less commonly) append a dollar sign to the name of the variable. It doesn't have a special meaning; it's used as a reminder of what the variable is storing. Another reason to adopt one of these conventions is to be sure not to invoke `$()` on a set of DOM elements on which we've already called this method. For example, you may erroneously write the following:

```
var allElements = $('*');
// Other code here...
$(allElements);
```

Using the aforementioned conventions, you can rewrite the previous statement prepending the dollar sign to the variable name, as shown here:

```
var $allElements = $('*');
```

Alternatively, you also can write it this way:

```
var allElements$ = $('*');
```

We recommend adopting one of these conventions and sticking with it. Throughout the rest of the book, we'll use the first one: the dollar sign prepended.

Let's now see the first complete example of using jQuery in a web page. In this example, shown in listing 2.1, we'll use a CDN to include jQuery using the fallback technique learned in chapter 1, and the Universal selector to select all the elements of the page. You can find the code for this listing in the file chapter-2/listing-2.1.html in the source provided with the book. In the remainder of the book, the examples will only include a reference to a local version of the jQuery library, avoiding the use of any CDN. There are two main reasons for this choice: brevity (that is, writing less code) and avoiding an additional HTTP request (that fails if you're running the examples while offline).

> **Listing 2.1   Using the Universal selector with jQuery**

```
<!DOCTYPE html>
<html>
    <head>
        <title>jQuery in Action, 3rd edition</title>
    </head>
    <body>
        <p>I'm a paragraph</p>
        <script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
```
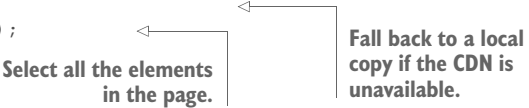
Request jQuery from the jQuery CDN.

```
    <script>
        window.jQuery || document.write('<script src="../js/jquery-
    1.11.3.min.js"><\/script>');
        var $allElements = $('*');
    </script>
  </body>
</html>
```

Fall back to a local copy if the CDN is unavailable.

Select all the elements in the page.

We told you that the previous listing was created to select all the elements of the page, but what are these elements? If you inspect the variable using a debugger or with the help of the console (where available), you'll see that they are `html`, `head`, `title`, `body`, `p`, `script` (the first in the page), and `script` (the second in the page).

> **WARNING**   We want to point out that the `console.log()` method is not supported by old versions of Internet Explorer (IE 6–7). In the examples in this book we'll ignore this issue and we'll use this method heavily to avoid resorting to the very annoying `window.alert()` method. But you should keep in mind this lack of support in case your code needs to work in these browsers.

Remember, the elements are retrieved and stored in the same order in which they appear on the page.

Developer tools

Trying to develop a DOM-scripted application without the aid of a debugging tool is like trying to play concert piano while wearing welding gloves. Why would you do that to yourself?

Depending on the browser you're using, there are different options you can choose to inspect your code. All major modern browsers have a set of built-in tools for this purpose, although with a different name, that you can adopt. For example, in Chrome these tools are called the *Chrome Developer Tools* (https://developers.google.com/chrome-developer-tools/), whereas in Internet Explorer they're called the *F12 developer tools* (http://msdn.microsoft.com/en-us/library/bg182326(v=vs.85).aspx). Firefox has its own built-in tools as well, but developers usually use a plugin called *Firebug* (http://getfirebug.com). These tools not only let you inspect the JavaScript console, but they also allow you to inspect the live DOM, the CSS, the scripts, and many other aspects of your page as you work through its development.

As you've seen, the use of the All selector forces jQuery to traverse all of the DOM's nodes. With a lot of elements in the DOM, the process might be very slow; therefore its use is discouraged. In addition, it's unlikely that you'll need to retrieve all the elements of a page, although you could need to collect those belonging to a specific subtree of the DOM, as you'll see later.

If you've ever played with JavaScript and a browser, you know that one of the most-used selections is performed using the ID of a given element. Let's discover more about this topic.

2.2.2   *The ID selector*

The ID selector is one of the most used selectors, not only in jQuery but also in plain JavaScript. In JavaScript, to select an element by its ID, you pass the ID to the native `document.getElementById()` function. If you have some knowledge of CSS, you'll recall that the ID selector is characterized by a sharp (#) sign (in some countries this symbol is known with a different name like *number sign* or *pound sign*) prepended to the element's ID. If you have this paragraph in your page

```
<p id="description">jQuery in Action is a book about jQuery</p>
```

you can retrieve it using the ID selector and jQuery by writing

```
$('#description');
```

When used with the ID selector, jQuery returns a collection of either zero or one DOM element. In case you have more than one element on a page with the same ID, the library retrieves only the first matched element encountered. Although you can have more than one element with the same ID, it's invalid and you should not do that.

> **NOTE**   The W3C specifications of HTML5 assert that the value of an ID "must not contain any space characters. There are no other restrictions on what form an ID can take; in particular, IDs can consist of just digits, start with a digit, start with an underscore, consist of just punctuation, etc." It's possible to use characters such as the period (.) that have a special meaning in CSS and jQuery (because it follows the CSS conventions). Because of this, they must be escaped by prepending two backslashes to the special character. Thus, if you want to select an element with ID of `.description`, you have to write `$('#\\.description')`.

It isn't accidental that we compared how to select elements by their ID in jQuery and in JavaScript at the beginning of this section, using the `getElementById()` function. In jQuery the selection by ID is the fastest one, regardless of the browser used, because behind the scenes the library uses `getElementById()`, which is very fast.

Using the ID selector you're able to quickly retrieve one element in the DOM. Often, you need to retrieve elements based on the class names used. How can you select elements that share the same style?

2.2.3   *The Class selector*

The Class selector is used to retrieve elements by the CSS class names used. As a JavaScript developer, you should be familiar with this kind of selection through the use of the native `getElementsByClassName()` function. jQuery follows the CSS conventions, so you have to prepend a dot before the chosen class name. For example, if you have the following HTML code inside the `<body>` of a page

```
<div>
  <h1 class="green">A title</h1>
  <p class="description">I'm a paragraph</p>
```

```
</div>
<div>
  <h1 class="green">Another title</h1>
  <p class="description blue">I'm yet another paragraph</p>
</div>
```

and you want to select the elements that have the class `description`, you need to pass `.description` to the `$()` function by writing the following statement:

```
var $descriptions = $('.description');
```

The result of this statement is an object, often referred to by the documentation as a *jQuery   object* or a *jQuery collection* (other names you can find in the wild are *set of matched elements*, or simply *set* or *collection*) containing the two paragraphs of the HTML snippet. The library will also select the nodes having multiple classes where one of them matches the given name (like the second paragraph).

In jQuery, like in CSS, it's also possible to combine more class name selectors. If you want to select all the elements having the classes `description` *and* `blue`, you can concatenate them, resulting in `$('.description.blue')`.

The Class selector is surely one of the most used among JavaScript and CSS, but there's another basic selector we still need to discuss.

2.2.4 *The Element selector*

The Element selector allows you to pick up elements based on their tag name. Because of its support in almost any browser (including IE6), jQuery uses `get-ElementsByTagName()` to select elements by tag name behind the scenes. To understand what kind of selection you can perform with the Element selector, let's say you want all the `<div>`s in a page. To achieve this task, you have to write

```
var $divs = $('div');
```

It's common to use it in conjunction with other selectors because you'll usually have a lot of elements of the same type in your pages. In such cases, it must be written *before* the other selectors. Hence, if you want all `<div>`s having class `clearfix`, you have to write the following statement:

```
var $clearfixDivs = $('div.clearfix');
```

You can also combine it with the ID selector, but we strongly encourage you to not do that for two reasons: performance and usefulness. Using a complex selector, jQuery will perform a search using its own methods, usually avoiding the use of native functions, and this leads to slower execution. In addition, as we pointed out in the section on the ID selector, jQuery will retrieve the first (if any) element having the searched ID. Therefore, if you're searching for just one element, there's no need to add complexity to your selector by mixing two types.

jQuery also enables you to use different types in a single selection, providing a performance gain because the DOM is traversed only once. To use it, you have to add a comma after each selector but the last one (spaces after the comma are ignored, so

their use is a matter of code style). To select all the <div>s and the <span>s in a page, you can write

```
$('div, span');
```

In case a given element matches more than one of the comma-separated selectors (which is not possible when you use only Element selectors because, for example, an element is a div *or* a span), the library will retrieve it only once, removing all the duplicates for you.

Thanks to the selectors discussed in these sections, you're able to perform basic searches in the DOM. But you often need to select elements using more complex criteria. You may need to retrieve DOM nodes based on their relation with other nodes like "all the links inside an unordered list." What you're doing here is specifying a selection based on the hierarchy of the elements. How to perform such a search is the topic of the next section.

2.3   *Retrieving elements by their hierarchy*

Retrieving a set of elements by their class name is a nice feature, but often you don't want to search the whole page. Sometimes you may want to select only the direct children of a certain element. Consider the following HTML fragment from the sample DOM in the Selectors Lab:

```
<ul class="my-list">
  <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 that you wanted to select the a element pointing to the jQuery website but not those to various local pages describing the different CSS specifications. To achieve this goal you can use the *Child selector*, in which a parent and its direct child are separated by the right angle bracket character (>). You can write

```
ul.my-list > li > a
```

This selector will collect only links that are direct children of list elements, which are in turn direct children of the <ul> that have class my-list. The links contained in the sublists are excluded because the ul element serving as their parent doesn't have the class my-list. Running this selector in the lab page gives the result shown in figure 2.3.

Figure 2.3   With the selector `ul.my-list > li > a`, only the direct children of parent nodes are matched.

The Child selector isn't the only one available to express a relation between two or more elements based on the DOM tree's hierarchy. Table 2.2 provides an overview of the selectors of this type.

Table 2.2   The CSS hierarchy selectors supported by jQuery

Selector	Description	In CSS?
E F	Matches all elements with tag name F that are descendants of E	✓
E>F	Matches all elements with tag name F that are direct children of E	✓
E+F	Matches all elements with tag name F that are immediately preceded by sibling E	✓
E~F	Matches all elements with tag name F preceded by any sibling E	✓

All the selectors described in the table but the first one are part of the CSS2.1 specifications, so they aren't supported by Internet Explorer 6. But you can use all of them safely in jQuery because the library deals with these kinds of problems for you.

These selectors improved your ability to precisely target the DOM nodes you want to act upon. Over time a lot of other CSS selectors have been created to place more power in your hands. One of the features introduced was the ability to select elements based on their attributes. These selectors are the topic of the next section.

2.4   *Selecting elements using attributes*

Attribute selectors are extremely powerful and allow you to select elements based on their attributes. You can easily recognize these selectors because they're wrapped with square brackets (for example, [selector]).

To see them in action, let's take another look at a portion of the lab page:

```
<ul>
   <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>
</ul>
```

What usually makes the link pointing to an external site unique is the http:// at the beginning of the string value for the link's href attribute. Actually, an external link may also be prefixed by https://, ftp://, and many other protocols. Besides, a link pointing to a page of the same website might still start with http://. But for the sake of simplicity we'll take into account http:// only and we'll pretend that internal links use only relative paths.

In CSS, you could select links that have an href value starting with http:// with the following selector:

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

Using jQuery, the latter can be employed in a statement like the following:

```
var $externalLinks = $("a[href^='http://']");
```

This matches all links with an href value beginning with the exact string http://. The caret character (^) is used to specify that the match has to occur at the beginning of a value. Because 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 again (from which the previous HTML fragment was lifted), type a[href^='http://'] into the text box, and click Apply. Note that only the jQuery link is highlighted.

Single and double quotes

Pay attention to single and double quotes when you use the attribute selectors. A wrong combination of the latter will result in an invalid statement. If your style of code adopts the use of double quotes for strings and you want to use the same quotes for wrapping the attributes value, you must escape them. If you feel it's easier for you to read a selection without escaped characters, you can mix the quote types. Using the selector `a[href^="http://"]` will result in the following equivalent statements:

```
$("a[href^=\"http://\"]");
$('a[href^=\'http://\']');
$("a[href^='http://']");
$('a[href^="http://"]');
```

Now imagine you want all the links but those pointing to the jQuery website's homepage. Using our lovely library, you can write

```
$("a[href!='http://jquery.com']")
```

This statement, using the "not equal attribute" selector, gives you the expected result. Because this selector isn't part of the CSS specifications, behind the scenes jQuery can't take advantage of the native `querySelectorAll()` method, so this results in a slower execution.

These symbols can't be combined with other ones to create even more powerful selectors. For example, if you want to select all links but the externals (assuming only those starting with `http://`), you can't write

```
$("a[href!^='http://']");
```

At this point you may think that selecting elements by their attribute is possible only in conjunction with the Element selector. But this isn't the case. You can use whatever selector you like, and even no other selectors at all, resulting in a selector like `[href^='http://']`. In this case, the use of the Universal selector (*) is implicitly assumed.

There are other ways to use attribute selectors. To match an element—for example, a `form`—that possesses a specific attribute, regardless of its value, you can use

```
form[method]
```

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

To match a specific attribute value, you use something like

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

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

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

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

This selects all `div` elements with a `title` attribute 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 here's a selector, called "attribute contains," for locating elements whose attributes contain arbitrary strings anywhere in the attribute value:

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

As you'd expect, this selector matches all `a` elements that reference the jQuery site.

Another selector is the "contain prefix." It selects elements with a given attribute's value equal to a specified string or equal to a specified string followed by a hyphen. If you write

```
div[class|='main']
```

this selector will find all the `<div>`s having `class="main"` or having a class name starting with `main-`, like `class="main-footer"`.

The last selector we're going to discuss is similar to the previous one, except it's used to search for a word within an attribute's value. Let's say you're using the HTML5 data-* attribute—for example, `data-technologies`—to specify a list of values in some `<span>`s of your page. You want to perform a search to find if one of them contains the value "javascript". You can perform this selection using the following selector:

```
span[data-technologies~="javascript"]
```

This selects `<span>`s having an attribute like `data-technologies="javascript"` but also `data-technologies="jquery javascript qunit"`. You can think of it as the equivalent of the Class selector but for a generic attribute.

The presented selectors can also be chained in case you need to retrieve nodes that match more criteria. You can chain as many selectors as you like; there isn't a fixed limit. For example, you can write

```
input[type="text"][required]
```

This selector retrieves all the `<input>`s that are required (the `required` attribute has been introduced in HTML5) and are of type `text`.

Table 2.3 summarizes the CSS selectors that deal with attributes that you can use in jQuery.

With all this knowledge in hand, head over to the jQuery Selectors Lab Page and spend some more time running experiments using selectors of various types from table 2.3. Try to make some targeted selections like the `input` element having the `type` of `checkbox` and the value of `1` (hint: you'll need to use a combination of selectors to get the job done).

Selectors aren't used only to retrieve elements using the `$()` function. As you'll discover later in this chapter, they're one of the most used parameters to pass to jQuery's methods. For example, once you've made a selection, you can use a jQuery method and a new selector to add new elements to the previous set or to filter some elements.

Another case is to find all the descendants of the elements in a previous stored set that match a given selector.

Table 2.3   The attribute selectors supported by jQuery

Selector	Description	In CSS?
`E[A]`	Matches all elements with tag name `E` that have attribute `A` of any value	✓
`E[A='V']`	Matches all elements with tag name `E` that have attribute `A` whose value is exactly `V`	✓
`E[A^='V']`	Matches all elements with tag name `E` that have attribute `A` whose value starts with `V`	✓
`E[A$='V']`	Matches all elements with tag name `E` that have attribute `A` whose value ends with `V`	✓
`E[A!='V']`	Matches all elements with tag name `E` that have attribute `A` whose value doesn't match `V` (are not equal to `V`) or that lack attribute `A` completely	
`E[A*='V']`	Matches all elements with tag name `E` that have attribute `A` whose value contains `V`	✓
`E[A|='V']`	Matches all elements with tag name `E` that have attribute `A` whose value is equal to `V` or to `V-` (`V` followed by a hyphen)	✓
`E[A~='V']`	Matches all elements with tag name `E` that have attribute `A` whose value is equal to `V` or contains `V` delimited by spaces	✓
`E[C1][C2]`	Matches all elements with tag name `E` that have attributes that satisfy the criteria `C1` and `C2`	✓

As if the power of the selectors that we've discussed so far isn't enough, there are some more options that offer an even finer ability to slice and dice the DOM. In the next section we'll introduce other types of selectors known as *filters*. In the CSS specification these types of selectors are referred as *pseudo-classes*.

2.5   *Introducing filters*

Filters are selectors that usually work with other types of selectors to reduce a set of matched elements. You can recognize them easily because they always start with a colon (:). Just as you've seen for the attributes, if another selector isn't specified, the use of the Universal selector is implicitly assumed. One of the peculiarities of these selectors is that some of them accept an argument passed inside the parentheses; for example, `p:nth-child(2)`. In the next sections, we'll discuss all the available filters in jQuery broken down into different categories.

2.5.1   *Position filters*

Sometimes you'll need to select elements by their position on the page. You might want to select the first or last link on the page or from the third paragraph. jQuery supports mechanisms for achieving these specific selections.

For example, consider

```
a:first
```

This format of selector matches the first <a> on the page. Now, let's say you want to retrieve links starting from the third one on the page. To achieve this goal, you can write

```
a:gt(1)
```

This selector is really interesting because it gives us the chance to discuss a few points. First, we're using a selector called Greater than (gt) because there isn't one called Greater than or equal. Also, unlike the selectors you've seen so far, it accepts an argument (1 in this case) that specifies the index from which to start. Why do you pass 1 if you want to start from the third element? Shouldn't it be 2? The answer comes from our programming background where indexes usually start at 0. The first element has index 0, the second has index 1, and so on.

These selectors specific to jQuery provide surprisingly elegant solutions to sometimes tough problems. See table 2.4 for a list of these Position filters (which the jQuery documentation collocates inside the basic filters category).

Table 2.4   The Position filters supported by jQuery

Selector	Description	In CSS?
:first	Selects the first match within the context. li a:first returns the first anchor that's a descendant of a list item.	
:last	Selects the last match within the context. li a:last returns the last anchor that's a descendant of a list item.	
:even	Selects even elements within the context. li:even returns every even-indexed list item.	
:odd	Selects odd elements within the context. li:odd returns every odd-indexed list item.	
:eq(n)	Selects the *n*th matching element.	
:gt(n)	Selects elements after the *n*th matching element (the *n*th element is excluded).	
:lt(n)	Selects elements before the *n*th matching element (the *n*th element is excluded).	

As we noted, the first index in a set of elements is always 0. For this reason, the :even selector will counterintuitively retrieve the odd-positioned elements because of their

even indexes. For example, `:even` will collect the first, third, and so on elements of a set because they have even indexes (0, 2, and so on). The takeaway lesson is `:even` and `:odd` are related to the index of the elements within the set, not their position.

Another fact to highlight is that you can also pass to `:eq()`, `:gt()`, and `:lt()` a negative index. In this case the elements are filtered counting backward from the last element. If you write `p:gt(-2)`, you're collecting only the last paragraph in the page. Considering that the last paragraph has index -1, the penultimate has index -2, and so on, basically you're asking for all the paragraphs that come after the penultimate.

In some situations you don't want to select only the first or last element in the whole page but each first or last element relative to a given parent in the page. Let's discover how.

2.5.2   Child filters

We said that jQuery embraces the CSS selectors and specifications. Thus, it shouldn't be surprising that you can use the child pseudo-classes introduced in CSS3. They allow you to select elements based on their position inside a parent element. Where the latter is omitted, the Universal selector is assumed. Let's say you want to retrieve elements based on their position inside a given element. For example,

```
ul li:last-child
```

selects the last child of parent elements. In this example, the last `<li>` child of each `<ul>` element is matched.

You may also need to select elements of a type only if they're the fourth child of a given parent. For example,

```
div p:nth-child(4)
```

retrieves all `<p>`s inside a `<div>` that are the fourth child of their parent element.

The `:nth-child()` pseudo-class is different from `:eq()` although they're often confused. Using the former, all the children of a containing element are counted, regardless of their type. Using the latter, only the elements corresponding to the selector attached to the pseudo-class are counted, regardless of how many siblings they have before them. Another important difference is that `:nth-child()` is derived from the CSS specifications; therefore it assumes the index starts from 1 instead of 0.

Another use case we can think of is "retrieve all the second elements having class `description` inside a `<div>`." This request is accomplished using the selector

```
div .description:nth-of-type(2)
```

As you're going through this section you should realize that the available selectors are many and powerful. Table 2.5 shows all the Child filters described so far and many more. Please note that when a selector allows for more syntaxes, like `:nth-child()`, a check mark in the In CSS? column of table 2.5 means that all the syntaxes are supported.

Table 2.5   The Child filters of jQuery

Selector	Description	In CSS?
`:first-child`	Matches the first child element within the context	✓
`:last-child`	Matches the last child element within the context	✓
`:first-of-type`	Matches the first child element of the given type	✓
`:last-of-type`	Matches the last child element of the given type	✓

`:nth-child(n)`
`:nth-child(even|odd)`
`:nth-child(Xn+Y)`
	Matches the *n*th child element, even or odd child elements, or *n*th child element computed by the supplied formula within the context based on the given parameter	✓

`:nth-last-child(n)`
`:nth-last-child(even|odd)`
`:nth-last-child(Xn+Y)`
	Matches the *n*th child element, even or odd child elements, or *n*th child element computed by the supplied formula within the context, counting from the last to the first element, based on the given parameter	✓

`:nth-of-type(n)`
`:nth-of-type(even|odd)`
`:nth-of-type(Xn+Y)`
	Matches the *n*th child element, even or odd child elements, or *n*th child element of their parent in relation to siblings with the same element name	✓

`:nth-last-of-type(n)`
`:nth-last-of-type(even|odd)`
`:nth-last-of-type(Xn+Y)`
	Matches the *n*th child element, even or odd child elements, or *n*th child element of their parent in relation to siblings with the same element name, counting from the last to the first element	✓

`:only-child`	Matches the elements that have no siblings	✓
`:only-of-type`	Matches the elements that have no siblings of the same type	✓

As table 2.5 points out, `:nth-child()`, `:nth-last-child()`, `:nth-last-of-type()`, and `:nth-of-type()` accept different types of parameters. The parameter can be an index, the word "even," the word "odd," or an equation. The latter is a formula where you can have an unknown variable as n. If you want to target the element at any position that's a multiple of 3 (for example 3, 6, 9, and so on), you have to write 3n. If you need to select all the elements at a position that's a multiple of 3 plus 1 (like 1, 4, 7, and so on), you have to write 3n+1.

Because things are becoming more complicated, it's best to see some examples. Consider the following table from the lab's sample DOM. It contains a list of programming languages and some basic information about 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 that you wanted to get all of the table cells that contain the names of programming languages. Because they're all the first cells in their rows, you could use

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

You could also use

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

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

To grab the language type cells, you'd change the selector to use `:nth-child(2)`, and for the year they were invented, you'd use `:nth-child(3)` or `:last-child`. If you wanted the absolute last table cell (the one containing the text *1983*), you'd use the `:last` pseudo-class seen in the previous section, resulting in `td:last`.

To test your ability, you can imagine another situation. Let's say that you want to retrieve the name of the languages and their year of creation using `:nth-child()`. Basically, what you're asking here is to take for each table row (`<tr>`) the first and the third columns (`<td>`). The first and easier solution is to pass `odd` as the argument to the filter, resulting in

```
#languages td:nth-child(odd)
```

Just to have some more fun, let's make the previous selection harder, assuming that you want to perform the same selection passing a formula to the `:nth-child()` filter. Recalling that the index for `:nth-child()` starts at 1, you can turn the previous selector into

```
#languages td:nth-child(2n+1)
```

This last example should reinforce in you the idea that jQuery puts great power in your hands.

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 to get a feel for the difference between the `:nth-child()` type of filters and the absolute position selectors.

Even though the CSS selectors we've examined so far are incredibly powerful, we'll discuss ways of squeezing even more power out of jQuery's selectors that are specifically designed to target form elements or their status.

2.5.3   *Form filters*

The CSS selectors that you've seen so far give you a great deal of power and flexibility to match the desired DOM elements, but there are even more selectors that give you greater ability to filter the selections.

As an example, you might want to match all check boxes that are in a checked state. You might be tempted to try something along these lines:

```
$('input[type="checkbox"][checked]');
```

But trying to match by attribute will check only the initial state of the control as specified in the HTML markup. What you really want to check is the real-time state of the controls. CSS offers a pseudo-class, `:checked`, that matches elements that are in a checked state. For example, whereas the `input[type="checkbox"]` selector selects all `input` elements that are check boxes, the `input[type="checkbox"]:checked` selector narrows the search to only `input` elements that are check boxes and are currently checked. When rewriting your previous statement to select all the check boxes that are currently checked using the filter, you can write

```
$('input[type="checkbox"]:checked');
```

jQuery also provides a handful of powerful custom filter selectors, not specified by CSS, that make identifying target elements even easier. For example, the custom `:checkbox` selector identifies all check box elements. Combining these custom selectors can be powerful and shrink your selectors even more. Consider rewriting once again our example using filters only:

```
$('input:checkbox:checked');
```

As we discussed earlier, jQuery supports the CSS filter selectors and also defines a number of custom selectors. They're described in table 2.6.

Table 2.6   The CSS and custom jQuery filter selectors

Selector	Description	In CSS?
`:button`	Selects only button elements (`input[type=submit]`, `input[type=reset]`, `input[type=button]`, or `button`)	

Table 2.6 The CSS and custom jQuery filter selectors *(continued)*

Selector	Description	In CSS?
`:checkbox`	Selects only check box elements (`input[type=checkbox]`)	
`:checked`	Selects check boxes or radio elements in the checked state or `option`s of `select` elements that are in a selected state	✓
`:disabled`	Selects only elements in the disabled state	✓
`:enabled`	Selects only elements in the enabled state	✓
`:file`	Selects only file input elements (`input[type=file]`)	
`:focus`	Selects elements that have the focus at the time the selector is run	✓
`:image`	Selects only image input elements (`input[type=image]`)	
`:input`	Selects only form elements (`input`, `select`, `textarea`, `button`)	
`:password`	Selects only password elements (`input[type=password]`)	
`:radio`	Selects only radio elements (`input[type=radio]`)	
`:reset`	Selects only reset buttons (`input[type=reset]` or `button[type=reset]`)	
`:selected`	Selects only `option` elements that are in the selected state	
`:submit`	Selects only submit buttons (`button[type=submit]` or `input[type=submit]`)	
`:text`	Selects only text elements (`input[type=text]`) or `input` without a `type` specified (because `type=text` is the default)	

These CSS and custom jQuery filter selectors can be combined, too. For example, if you want to select only enabled and checked check boxes, you could use

```
$('input:checkbox:checked:enabled');
```

Try out as many of these filters as you like in the Selectors Lab Page 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 your disposal, but did you think, even for one moment, that the selectors ended here? No way!

2.5.4 Content filters

Another of the categories that you can find in the jQuery documentation is the one containing *Content filters.* As the name suggests, these filters are designed to select elements based on their content. For example, you can choose elements if they contain a given word or if the content is completely empty. Note that by *content* we mean not only raw text but also child elements.

As you saw earlier, CSS defines a useful selector for selecting elements that are descendants of specific parents. For example, this selector

```
div span
```

will select all span elements that are descendants of div elements.

But what if you wanted the opposite? What if you wanted to select all <div>s that contained span elements? That's the job of the :has() filter. Consider this selector

```
div:has(span)
```

which selects the div ancestor elements as opposed to the span descendant elements.

This can be a powerful mechanism when you get to the point where you want to select elements that represent complex constructs. For example, let's say that you want to find which table row contains a particular image element that can be uniquely identified using its src attribute. You might use a selector such as this

```
$('tr:has(img[src="puppy.png"])');
```

which would return any table row element containing the identified image anywhere in its descendant hierarchy.

A complete list of the Content filters is shown in table 2.7.

Table 2.7   The Content filters supported by jQuery

Selector	Description	In CSS?
:contains(text)	Selects only elements containing the specified text (the text of the children and the descendants is also evaluated).	
:empty	Selects only elements that have no children (including text nodes).	✓
:has(selector)	Selects only elements that contain at least one element that matches the specified selector.	
:parent	Selects only elements that have at least one child node (either an element or text).	

If you're starting to feel overwhelmed by all these selectors and filters, we suggest that you take a small break, because you aren't finished yet!

2.5.5   *Other filters*

You've seen an incredible number of selectors and filters (special selectors) that you probably didn't even know existed. Your journey into the world of selectors hasn't ended, and in this section we'll discuss the remaining ones. A couple of them, :visible and :hidden, are categorized in the library's documentation under *Visibility filters*, but for brevity we decided to include them here.

If you want to negate a selector—let's say to match any input element that's *not* a check box—you can use the :not() filter. For example, to select non–check box input elements, you could use

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

But be careful! It's easy to go astray and get some unexpected results!

Let's say that you wanted to select all images except those whose `src` attribute contained the text *dog*. You might quickly come up with the following selector:

```
$(':not(img[src*="dog"])');
```

But if you used this selector, you'd find that not only did you get all the image elements that don't reference *dog* in their `src`, but in general, every element in the DOM that isn't an image element with such `src` attribute's value!

Whoops! Remember that when a base selector is omitted, it defaults to the Universal selector. Your errant selector actually reads as "fetch all elements that aren't images that reference 'dog' in their `src` attributes." What you intended was "fetch all image elements that don't reference 'dog' in their `src` attributes," which would be expressed like this:

```
$('img:not([src*="dog"])');
```

Again, use the lab page to conduct experiments until you're comfortable with how to use the `:not()` filter to invert selections.

When working with jQuery it's common to use its methods to hide one or more elements on a page. To retrieve these elements you can use the `:hidden` filter. An element is considered hidden not only if it has

```
display: none;
```

applied but also if it doesn't occupy space. For example, a hidden element is also one that has its width and height set to zero. Using this selector

```
input:hidden
```

you're targeting all the `input` elements of the page that are hidden.

---

jQuery 3: Feature changed

jQuery 3 slightly modifies the meaning of the `:visible` (and therefore of `:hidden`) filter. Starting from jQuery 3, elements will be considered `:visible` if they have any layout boxes, including those of zero width and/or height. For example, `br` elements and inline elements with no content will now be selected by the `:visible` filter.

When creating web pages, you often use foreign words. If you write correct, semantic HTML, you'll find yourself tagging those words using the `em` element, adding the `lang` attribute to specify the language. Let's say that you have a page about pizza; you could have markup like the following:

```
<p>The first pizza was called <em lang="it">Margherita</em>, and it was
    created in <em lang="it">Napoli</em> (Italy).</p>
```

You can select all those foreign words of this example using the `:lang()` filter in this way:

```
var $foreignWords = $('em:lang(it)');
```

A complete list of the remaining filters is shown in table 2.8.

Table 2.8   The remaining filters supported by jQuery

Selector	Description	In CSS?
:animated	Selects only elements that are currently under animated control	
:header	Selects only elements that are headers: `<h1>` through `<h6>`	
:hidden	Selects only elements that are hidden	
:lang(language)	Selects elements in a specified language	✓
:not(selector)	Negates the specified selector	✓
:root	Selects the element that's the root of the document	✓
:target	Selects the target element indicated by the fragment identifier of the document's URI	✓
:visible	Selects only elements that are visible	

Although jQuery offers an incredible number of selectors, it doesn't cover all the possible use cases, and the development team behind the library knows it. For this reason, they gave you the option to create your own filters. Let's look at how you can do this.

2.5.6 *How to create custom filters*

In the previous sections, you learned all the selectors and filters supported by jQuery. Regardless of their number, you may deal with use cases not covered. You may also find yourself doing the same selection and then the same filtering on the retrieved set over and over again, using loops and selection constructs. In situations like these you can create a shortcut to collect nodes of the DOM or, to better phrase it, you can create a *custom filter* (also referred as a *custom selector* or *custom pseudo-selector*).

In jQuery there are two ways to create a custom filter. The first is simpler to write but its use is discouraged because it has been replaced, starting from jQuery 1.8, by the second one. In this book we'll describe the newer method only, but if you want to take a look at the old way, we've prepared a JS Bin just for you (http://jsbin.com/ImIboXAz/edit?html,js,console,output). The example is also available in the file chapter-2/custom.filter.old.html of the source provided with this book. Keep in mind that when using the new approach, you're developing a custom filter that won't work in versions of jQuery prior to 1.8. However, this shouldn't be a problem in many cases as this version is obsolete.

To explain the new way to create a custom filter, we'll start with an example. Pretend you're developing a tech game where you have a list of levels to complete with a certain grade of difficulty, the number of points the user can earn, and a list of technologies to employ to complete it. Your hypothetical list could resemble this:

```
<ul class="levels">
  <li data-level="1" data-points="1" data-technologies="javascript node
     grunt">Level 1</li>
  <li data-level="2" data-points="10" data-technologies="php composer">Level
     2</li>
  <li data-level="3" data-points="100" data-technologies="jquery
     requirejs">Level 3</li>
  <li data-level="4" data-points="1000" data-technologies="javascript jquery
     backbone">Level 4</li>
</ul>
```

Now imagine you often need to retrieve levels (data-level) higher than 2 but only if they allow you to earn more than 100 points (data-points) and have jQuery in the list of the technologies to employ (data-technologies). Using the knowledge you've acquired so far, you know how to search li elements having the word *jquery* inside the attribute data-technologies (li[data-technologies~="jquery"]). But how do you perform a number comparison using selectors? The truth is you can't. To accomplish this task, you must loop over your initial selection and then retain only the elements you need, as shown here:

> **Initial selection using the attribute selector**

```
var $levels = $('.levels li[data-technologies~="jquery"]');    ⊲
var matchedLevels = [];
for(var i = 0; i < $levels.length; i++) {      ⊲      Loop over the matched
  if ($levels[i].getAttribute('data-level') > 2 &&         set of elements.
      $levels[i].getAttribute('data-points') > 100) {   ⊲
    matchedLevels.push($levels[i]);   ⊲               Test if the current
  }                                                    element matches
}                          Add to the final            the requirements.
                           set of elements.
```

Instead of repeating these lines every time, you can create a custom filter:

> **Declare the filter using the createPseudo() function.** ❶

```
$.expr[':'].requiredLevel = $.expr.createPseudo(function(filterParam) {   ⊲
  return function(element, context, isXml) {      ⊲        Return the
    return element.getAttribute('data-level') > 2 &&        anonymous
           element.getAttribute('data-points') > 100;       function called to
  };                                                  ❷     perform the tests.
});
```

Tests the current element. ❸

As you can see, a filter is nothing but a function added to a property called :, which belongs to jQuery's expr attribute. That's no mistake, dear reader. It's a property called "colon." The latter is a property containing jQuery's native filters, and you can use it to add your own.

You call your custom filter requiredLevel, and instead of passing the function directly, you use a jQuery utility (actually it belongs to the underlying Sizzle selectors engine) called createPseudo() ❶.

To the `createPseudo()` function, you pass an anonymous function where you declare a parameter called `filterParam`. The name of the latter, standing for "filter parameter," is arbitrary and you can choose a different one if you prefer. This parameter represents an optional parameter you can pass to the filter, just like filters such as `:eq()` and `:nth-child()`, that you won't use for the moment. Inside this anonymous function, you create another anonymous function that will be returned and that's responsible for performing the filtering. To this inner function, jQuery passes the elements to be processed one at a time (`element` parameter), the DOMElement or DOMDocument from which selection will occur (`context` parameter), and a Boolean that specifies if you're working on an XML document or not (`isXML` parameter) ❷. Inside the innermost function, you write the code to test whether the element should be kept or not ❸. In your case, you test whether the level is higher than 2 and the points the user can earn are more than 100.

In the previous example, we introduced an argument called `filterParam` that you can use to pass a parameter to your custom filter. Due to the fixed nature of our requirements, we didn't use it. Let's have some fun seeing how it can help you.

Imagine you want to retrieve levels based on the offered number of points—something like "select all the levels with a number of points higher than X." That big X is a good opportunity to use a parameter to pass to your pseudo-selector. Based on this requirement, you can create a new filter:

```
$.expr[':'].pointsHigherThan = $.expr.createPseudo(function(filterParam) {
  var points = parseInt(filterParam, 10);
  return function(element, context, isXml) {
    return element.getAttribute('data-points') > points;
  };
});
```

Cache argument to be available in the inner function's closure. ❶

Use the cached argument in the test. ❷

There are a few differences compared to the previous example. You use the `createPseudo()` function as before, but you call the filter `pointsHigherThan`. Before declaring the second function, you need to save the argument in a variable called `points` ❶ so it'll be available in its closure (if you don't know what a closure is, read the section on closures in the appendix). At this point, you can use the given argument through the use of the stored variable ❷.

Let's put this new filter into action. If you want to retrieve all the levels that allow you to earn more than 50 points, you can write

```
var $elements = $('.levels li:pointsHigherThan(50)');
```

obtaining the last two list items.

Both the custom filters presented in this section are available in the file chapter-2/ custom.filter.html and also as a JS Bin (http://jsbin.com/mucigo/edit?html,js,console ,output).

So far, you've used half the power of the jQuery() function used to select elements because you used just one of the two parameters you can pass. It's time to fix this.

2.6 *Enhancing performances using context*

Up to this point, we've been acting as if there were only one argument that we can pass to jQuery's $() function, but this was just a bit of hand waving to keep things simple at the start. In chapter 1 we briefly introduced a second parameter called context. It's used to restrict the selection to one or more subtrees of the DOM, depending on the selector used. This argument is helpful when you have a large number of elements in a page because it can narrow down the subtree(s) where jQuery will perform the second phase of the search.

As you'll see with many of jQuery's methods, when an optional argument is omitted, a reasonable default is assumed. And so it is with context. When a selector is passed as the first parameter, context defaults to document, applying that selector to every element in the DOM tree.

That's often exactly what you want, so it's a nice default. But there may be times when you want to limit your search to a subset of the entire DOM. In such cases, you can identify a subset of the DOM that serves as the root of the subtree to which the selector is applied.

The Selectors Lab offers a good example of this scenario. When that page applies the selector that you typed into the text field, the selector is applied only to the subset of the DOM that's loaded into the DOM Sample pane.

You can use a DOM element reference as context but also a string that contains a jQuery selector or a jQuery collection. (Yes, that means that you can pass the result of one $() invocation to another—don't let that make your head explode yet; it's not as confusing as it may seem at first.)

When a selector or jQuery collection is provided as context, the identified elements serve as the context for the application of the selector. Because there can be multiple such elements, this is a nice way to provide disparate subtrees in the DOM to serve as the context for the selection process.

Let's take the lab page as an example. We'll assume that the selector string is stored in a variable conveniently named selector. When you apply this submitted selector, you want to apply it only to the sample DOM, which is contained within a div element with an ID of sample-dom.

If you were to code the call to the jQuery function like this

```
$(selector);
```

the selector would be applied to the entire DOM tree, including the form in which the selector was specified. That's not what you want. What you want is to limit the selection process to the subtree of the DOM rooted at the div element with the ID of sample-dom; so instead you write

```
$(selector, '#sample-dom');
```

which limits the application of the selector to the desired portion of the DOM.

When you use context, jQuery first retrieves elements based on it and then selects the descendants that match the selector provided as the first argument. In other words, you search for elements that match selector that need to have context as their ancestor. Therefore, the *Descendant selector* can be replaced by the use of context. Consider the following selection where you select the <p>s inside a <div>:

```
$('div p');
```

It can be turned into

```
$('p', 'div');
```

giving the same result.

With this section we've completed the discussion of jQuery selectors. We know how hard it has been to go through all these selectors, and you shouldn't feel discouraged. Take your time to absorb the described concepts, and when you feel ready, move on.

Before we look at the methods of chapter 3, we'll test your skills with some exercises focused on the concepts described so far.

2.7   *Testing your skills with some exercises*

In this section you'll practice doing some exercises targeting the selectors and the filters described in this chapter. If you want to test your solutions, you can run them using the jQuery Selectors Lab Page. In addition, we'll provide you our solutions to allow you to compare them with yours.

2.7.1   *Exercises*

Here's the list of exercises:

1 Select all the links in the page.
2 Select all the direct child links of a <div> having the class wrapper.
3 Select all the links and the paragraphs that have as their ancestor a <div>.
4 Select all the <span>s that have the attribute data-level equal to hard but not the attribute data-completed equal to true.
5 Select all the elements on the page having the class name wrapper without using the class selector.
6 Select the third list item inside the list having the ID list, at any level.
7 Select all the list items (li) inside the list having the ID list, after the second.
8 Select the paragraphs that are the multiple of 3 plus 1 (1, 4, 7, and so on) child of their parent, having the class description.
9 Select the <input>s of type password only if they're required (required attribute of HTML5) and are the first child of a <form>.
10 Select all the <div>s in the page that have no children, have an odd position (hint: not index!), and don't have the class wrapper.
11 Create a custom filter to select elements having only numbers, letters, or the underscore (_) as their text.

2.7.2   *Solutions*

Here's the list of solutions:

```
1  $('a')
2  $('div.wrapper > a')
3  $('div a, div p') or even better, using the context parameter, $('a, p',
   'div')
4  $('span[data-level="hard"][data-completed!="true"]')
5  $('[class~="wrapper"]')
6  $('#list li:eq(2)') or even better $('li:eq(2)', '#list')
7  $('li:gt(1)', '#list')
8  $('p.description:nth-child(3n+1)')
9  $('input[required]:password:first-child', 'form')
10 $('div:empty:even:not(.wrapper)')
11 $.expr[":"].onlyText = $.expr.createPseudo(function(filterParam) {
     return function(element, context, isXml) {
       return element.innerHTML.match(/^\w+$/);
     }
   });
```

How did you do? Do you feel comfortable with the ideas outlined so far? Good! With this section we've completed the overview of the selectors available and how you can create your own.

2.8   *Summary*

This chapter focused on creating and adjusting sets of elements (referred to in this chapter and beyond as a *jQuery collection* or *set of matched elements*) via the many means 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 the CSS3 syntax currently supported by most modern browsers. jQuery not only supports all the CSS selectors but also expands them with its own set of selectors, offering you even more expressive power to collect elements in a web page. As if this wasn't enough, jQuery is so flexible that it also allows you to create your own filters.

In this chapter we covered all the selectors available in jQuery. In the next chapter we'll take a look at how to use the $() function to create *new* HTML elements. You'll also discover methods that accept a selector as a parameter to perform some operations on a set of matched elements.

Operating on a
jQuery collection

In this chapter you'll discover how to create new DOM elements using the highly flexibile `jQuery()` function. The need to create new elements will occur frequently in your practice with the library. You'll find yourself using this capability especially when we start discussing how to inject external data into a web page using JSON and the XML format and jQuery's methods to work with Ajax.

In addition, you'll learn other methods that are different from `jQuery()`. We'll divide these methods into two parts. First, we'll describe the methods that, starting from a jQuery collection, accept a selector as a parameter to create a new set of elements. For example, you'll see a method that, starting from a set, creates a new set containing all the children of the elements in the initial set, optionally filtered using the selector passed as its argument. Then we'll cover methods that aren't strictly related to selectors but that allow you to iterate over the elements in a set or perform a test on them. Let's get started!

3.1 *Generating new HTML*

On many occasions, you'll want to generate new fragments of HTML to insert into a page. Such dynamic elements could be as simple as extra text you want to display or as complicated as creating a table of database results you've obtained from a server. A typical situation where this feature comes in handy is when you need to fetch external data, usually served as JSON or XML, using Ajax.

With jQuery, creating dynamic elements is a simple matter. You can create a jQuery object containing DOM elements on the fly by passing to the `$()` function a string that contains the HTML markup for those elements. Consider this line:

```
$('<div>Hello</div>');
```

This expression creates a new jQuery object containing a `div` element that's ready to be added to the page (at this point it isn't injected in the DOM). Any jQuery method that you could run on a set of existing elements can be run on the newly created HTML fragment. This may not seem impressive at first glance, but when you throw event handlers, Ajax, and effects into the mix (as you will in the upcoming chapters), you'll discover how powerful it is.

Note that if you want to create an empty `div` element, you can get away with this shortcut:

```
$('<div>');
```

This is identical to `$('<div></div>')` and `$('<div />')`, although it's highly recommended that you use well-formed markup and include the opening and closing tags for any element types that can contain other elements. From a performance perspective these three alternatives are equivalent, as you can see from the benchmark shown in figure 3.1 (live test at http://jsperf.com/jquery-create-markup/4).
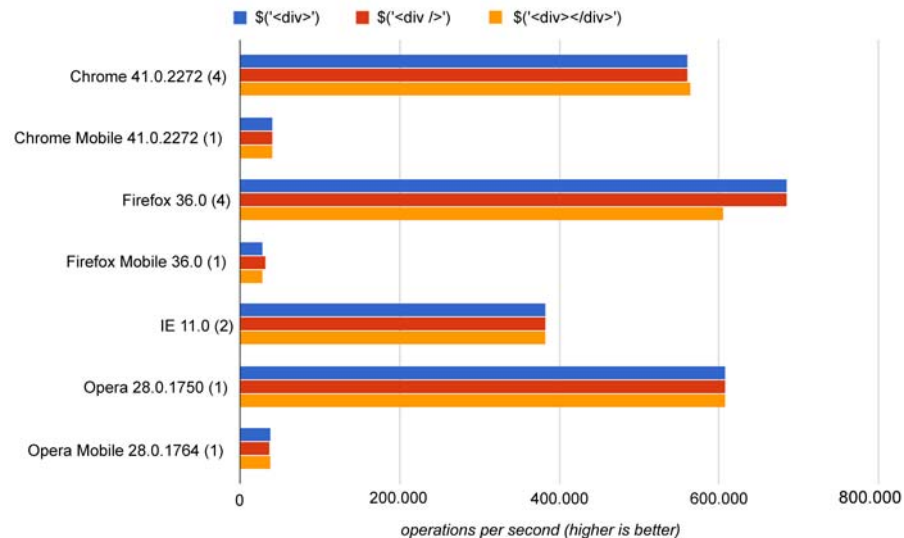
Figure 3.1 A benchmark comparing the three ways of creating a new element using `jQuery()`. It proves that they are equivalent from a performance point of view in almost every browser.

It's easy to create such simple HTML elements, and thanks to the chainability of jQuery methods, creating more complex elements isn't much harder. You can apply any jQuery method to the jQuery collection containing the newly created element. You could also create attributes on the element with jQuery's `attr()` method (we'll cover that in a later chapter), but jQuery provides an even better means to do so.

In the previous chapter we introduced you to the context parameter of the `$()` function. When creating a new element with the `$()` function, you use the context parameter to specify the attributes and their values for the element you're creating in the form of a JavaScript object. The properties of such an object serve as the name of the attributes to be applied to the element, whereas the values serve as the values of the attributes.

Let's say that you want to create an `img` element complete with multiple attributes and make it clickable to boot. Take a look at the code in the following listing.

Listing 3.1   Dynamically creating a full-featured `img` element

```
$('<img>',                                        ➊ Creates the basic img element
  {
    src: 'images/little.bear.png',
    alt: 'Little Bear',                           ➋ Assigns various attributes
    title:'I woof in your general direction',
    click: function() {
      alert($(this).attr('title'));
    }
  })
  .appendTo('body');
```

Establishes the click handler ➌

➍ Add the element to the DOM by appending it at the end of the `body` element.

The single jQuery statement in the listing creates the basic `img` element ➊; gives it important attributes using the second parameter, such as its source, alternate text, and flyout title ➋; and attaches it to the DOM tree (as a child of the `body` element) ➍. In the example shown, you append the element to the DOM using jQuery's `appendTo()` method. We haven't covered this method yet but it appends the elements in the jQuery collection—in this case only the newly created image—to the element specified in the argument, which in our example is the `body` element.

We're also throwing a bit of a curve ball at you here. In this example you also use the second parameter to establish an event handler that issues an alert (garnered from the image's `title` attribute) when the image is clicked ➌.

Regardless of how you arrange the code, that's a pretty hefty statement—which is spread across multiple lines and with logical indentation for readability—but it also does a heck of a lot. Such statements aren't uncommon in jQuery-enabled pages, and if you find it a bit overwhelming, don't worry. We'll cover every method used in this statement over the next few chapters. Writing such compound statements will be second nature before too long.

Figure 3.2a   Creating complex elements on the fly(including this image, which generates an alert when it's clicked) is easy as pie.
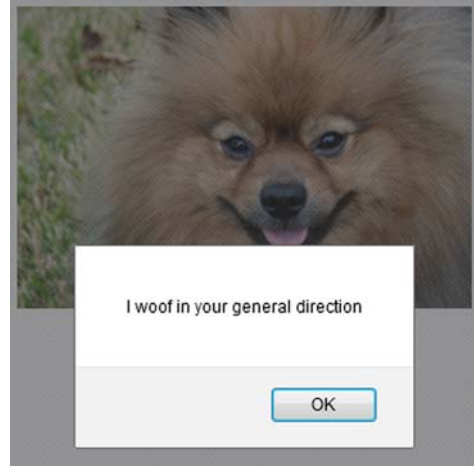
Figure 3.2b   The dynamically generated image possesses all expected styles and attributes, including the mouse click behavior of issuing an alert.

Figure 3.2 shows the result of this code, both when the page is first loaded (3.2a) and after the image has been clicked (3.2b). The full code for this example can be found in the book's project code at chapter-3/listing-3.1.html.

Up until now, you've applied methods to the entire set of matched elements, but there may be times when you want to further manipulate that set before acting upon it.

3.2   *Managing the jQuery collection*

Once you have a jQuery set, whether identified from existing DOM elements with selectors or created as new elements using HTML snippets (or a combination of both), you're ready to manipulate those elements using the powerful set of jQuery methods. We'll start looking at those methods in the next chapter, but what if you want to further refine the jQuery set? In this section, we'll explore the many ways that you can refine, extend, or filter the jQuery set that you wish to operate upon.

In order to help you in this endeavor, we've included another lab in the downloadable project code for this chapter: the jQuery Operations Lab Page (chapter-3/lab.operations.html). This page, which looks a lot like the Selectors Lab we employed in chapter 2, is shown in figure 3.3.

This new lab page not only looks like the Selectors Lab, but it also operates in a similar fashion. But in this lab, rather than typing a selector, you can type in any complete jQuery operation that results in a jQuery collection. The operation is executed in the context of the DOM Sample, and, as with the Selectors Lab, the results are displayed.

Figure 3.3   The jQuery Operations Lab Page lets you compose jQuery collections in real time to help you see how collections can be created and managed.

NOTE   This lab page loads the elements upon which it acts inside an `iframe`. Due to the security restrictions of some browsers, this operation may fail. To avoid this issue, you can either execute the page under a web server like Apache, Tomcat, or IIS or search for a specific solution for your browser. For example, in WebKit-based browsers, you can run them through the command-line interface (CLI) using the flag `--allow-file-access-from-files`. It's important that the command creates a new process, so it must not open a new tab but a new window.

The jQuery Operations Lab allows you to enter any expression that results in a jQuery set. Because of the way jQuery chaining works, this expression can also include jQuery methods, making this a powerful lab for examining the operations of jQuery.

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

To get a feel for the lab, load it in your browser and enter this text into the Operation field:

```
$('img').hide();
```

Then click the Execute button. This operation is executed within the context of the DOM Sample, and you'll see how the images disappear from the sample.

After any operation, you can restore the DOM Sample to its original condition by clicking the Restore button. Although we haven't treated it yet, the `hide()` method belongs to jQuery, and you'll see it in detail later in this book. For the moment, what you need to know is that this function allows you to hide all the elements in a set. We used it because we wanted to give you a concrete example of what you can do in the Operations Lab Page. You'll see this new lab in action as you work your way through the sections that follow, and you might even find it helpful in later chapters to test various jQuery operations.

3.2.1 *Determining the size of a set*

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

Let's say you want to know the number of all the paragraphs in your page and show it on the screen; you can write the following statement:

```
alert($('p').length);
```

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

3.2.2 *Obtaining elements from a set*

Once you have a jQuery set, you often use jQuery methods to perform some sort of operation upon it as a whole. There may be times when you want a direct reference to an element or elements to perform raw JavaScript operations upon them. Let's look at some of the ways that jQuery allows you to do just that.

FETCHING ELEMENTS BY INDEX
Because jQuery allows you to treat a jQuery collection as a JavaScript array, you can use simple array indexing to obtain any element in the list by position. For example, to obtain the first element in the set of all `<img>`s with an `alt` attribute on the page, you could write

```
var imgElement = $('img[alt]')[0];
```

The most observant of you might have noticed that we didn't prepend the dollar sign ($) in front of the variable name (imgElement). We didn't forget it. This jQuery set contains an array of DOM elements, so if you retrieve a single element, it isn't a jQuery set made of one element itself but a plain DOM element.

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

Method syntax: get

get([index])

Obtains one or all of the matched elements in the set. If no parameter is specified, all elements in the jQuery object are returned as a JavaScript array. If an index parameter is provided, the indexed element is returned. index can be negative, in which case the count is performed starting from the end of the matched set.

Parameters

index   (Number) The index of the single element to return. If omitted, the entire set is returned as an array. If a negative integer is given, the count starts from the end of the set. If the index is out of bounds, which is less than the negative number of elements or equal to or greater than the number of elements, the method returns undefined.

Returns

A DOM element, or an array of DOM elements, or undefined.

The fragment

```
var imgElement = $('img[alt]').get(0);
```

is equivalent to the previous example that used array indexing.

The get() method also accepts a negative index. Using get(-1) will retrieve the last element in the set, get(-2) the second to last, and so on. In addition to obtaining a single element, get() can also return an array of all the elements of the set if used without a parameter.

Sometimes you'll want a jQuery object containing a specific element rather than the plain element itself. It would look weird (although valid) to write something like this:

```
$($('p').get(2))
```

For this purpose, jQuery provides the eq() method. The latter mimics the action of the :eq() selector filter discussed in the previous chapter. To see their differences in terms of code, let's say you want to select the second element in a set containing all the <div>s of a page. Here's how you can perform this task, reporting the alternatives side by side:

```
var $secondDiv = $('div').eq(1);     var $secondDiv = $('div:eq(1)');
```

Selecting using the
eq() method

Selecting using the
:eq() filter

The difference between the statements is minimal, but for performance reasons (more details in chapter 15) it's better to stick with the first form (the eq() method). As a rule of thumb, we suggest you use methods over filters because they usually lead to better performance.

Now that we've highlighted the difference between the method and the filter, it's time to dive into the details of the former.

Method syntax: eq

eq(index)

Obtains the indexed element in the set and returns a new set containing just that element.

Parameters

index   (Number) The index of the single element to return. A negative index can be specified to select the element starting from the end of the set.

Returns

A jQuery collection containing one or zero elements.

Obtaining the first element of a set is such a common operation that there's a convenience method that makes it even easier: the first() method.

Method syntax: first

first()

Obtains the first element in the set and returns a new set containing just that element. If the original set is empty, so is the returned set.

Parameters

Returns

A jQuery collection containing one or zero elements.

The first() method has its filter counterpart in the :first filter. Once again, we want to show an example of the two alternatives. Say that you want to retrieve the first paragraph of the page; you can write one of the following:

```
var $firstPar = $('p').first();          var $firstPar = $('p:first');
```

Selecting using the
first() method

Selecting using the
:first filter

Not surprisingly, the difference in terms of code is minimal, but the first() method should be preferred to the :first filter.

As you might expect, there's a corresponding method to obtain the last element in a set as well, which is the counterpart of the `:last` filter.

Method syntax: `last`

`last()`

Obtains the last element in the set and returns a new set containing just that element. If the original set is empty, so is the returned set.

Parameters

None

Returns

A jQuery collection containing one or zero elements.

If you want to practice with these methods, you can use the jQuery Operations Lab Page. For example, if you want to retrieve the first list item of the list shown in the page, you can write

```
$('li', '.my-list').first();
```

Now let's examine the other method to obtain an array of elements in the set.

FETCHING ALL THE ELEMENTS AS AN ARRAY

If you wish to obtain all of the elements in a jQuery object as a JavaScript array of DOM elements, jQuery provides the `toArray()` method.

Method syntax: `toArray`

`toArray()`

Returns the elements in the set as an array of DOM elements.

Parameters

None

Returns

A JavaScript array of the DOM elements within the sets.

Consider this example:

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

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

FINDING THE INDEX OF AN ELEMENT

Whereas `get()` finds an element given an index, you can use an inverse operation, `index()`, to find the index of a particular element in the set. The syntax of the `index()` method is as follows.

Method syntax: `index`

`index([element])`

Finds the specified element in the set and returns its ordinal index within the set, or finds the ordinal index of the first element of the set within its siblings. If the element isn't found, the value -1 is returned.

Parameters

element   (Selector|Element|jQuery) A string containing a selector, a reference to the element, or a jQuery object whose ordinal value is to be determined. In case a jQuery object is given, the first element of the set is searched. If no argument is given, the index returned is that of the first element of the set within its list of siblings.

Returns

The ordinal value of the specified element within the set or its siblings or -1 if not found.

To help you understand this method, let's say that you have the following HTML code:

```
<ul id="main-menu">
  <li id="home-link"><a href="/">Homepage</a></li>
  <li id="projects-link"><a href="/projects">Projects</a></li>
  <li id="blog-link"><a href="/blog">Blog</a></li>
  <li id="about-link"><a href="/about">About</a></li>
</ul>
```

For some reason you want to know the ordinal index of the list item (`<li>`) containing the link to the blog, which is the element having the ID of `blog-link`, within the unordered list having the ID of `main-menu`.

> **NOTE** It's not a best practice to fill your pages with so many IDs because in large applications it's hard to manage them and assure that there won't be duplicates. We used them for the sake of the example.

You can obtain this value with this statement:

```
var index = $('#main-menu > li').index($('#blog-link'));
```

Based on what you learned about the parameters accepted by `index()`, you can also write this statement like this:

```
var index = $('#main-menu > li').index(document.getElementById('blog-link'));
```

Remember that the index is zero-based. The first element has index 0, the second has index 1, and so on. Thus, the value you'll obtain is 2 because the element is the third in the list. This code is available in the file chapter-3/jquery.index.html and also as a JS Bin (http://jsbin.com/notice/edit?html,js,console).

The `index()` method can also be used to find the index of an element within its parent (that is, among its siblings). This case can be a bit hard to understand, so let's drill down to see its meaning. The parent of the list item with ID of `blog-link` is the unordered list `main-menu`. The siblings are the elements at the same level from the DOM tree point of view (siblings) of `blog-link` that share the same parent (the unordered list). Given our markup, these elements are all the other list items. The links are

excluded because they're inside `main-menu` but not at the same level of `blog-link`. Writing this

```
var index = $('#blog-link').index();
```

will set `index`, once again, to 2.

To understand why calling `index()` without a parameter is interesting, consider the following markup:

```
<div id="container">
  <p>This is a text</p>
  <img src="image.jpg" />
  <a href="/">Homepage</a>
  <img src="anotherImage.jpg" />
  <p>Yet another text</p>
</div>
```

This time, the markup contains several different elements. Let's say you want to know the ordinal index of the first `img` element within its parent (the `<div>` with ID of `container`). You can write

```
var index = $('#container > img').index();
```

The value of `index` will be set to 1 because, among the children of `container`, the first `<img>` found is the second element (coming after the `<p>`).

In addition to retrieving the index of an element, jQuery also gives you the ability to obtain subsets of a set, based on the relationship of the items in a jQuery collection to other elements in the DOM. Let's see how.

3.2.3 *Getting sets using relationships*

jQuery allows you to get new sets from an existing one, based on the hierarchical relationships of the elements to the other elements within the DOM.

Let's say you have a paragraph having an ID of `description` and you want to know the number of its ancestors that are a `<div>`. With your current knowledge of selectors and methods this isn't possible. That's where a function like `parents()` comes into play. Consider the following code:

```
var count = $('#description').parents('div').length;
```

Using `parents()`, you're able to retrieve the information you want. This method retrieves the ancestors of each element in the current set of matched elements (which consists of the only paragraph having `description` as its ID). You can optionally filter the ancestors using a selector, as in the example. Because in your jQuery collection you have just one element (assuming it exists in your page), the result is what you expect.

What if you want to know the number of children of your hypothetical paragraph? This can be easily achieved using selectors:

```
var count = $('#description > *').length;
```

But wait! Are you using the same Universal selector we highly discouraged in the previous sections? Unfortunately, yes. A better approach from a performance point of view is to express the same statement using the `children()` method as follows:

```
var count = $('#description').children().length;
```

This method, however, doesn't return text nodes. How can you deal with this case?

For such situations where you have to work with text nodes, you can employ `contents()`. The `contents()` method and `children()` differ in another detail: the former doesn't accept any parameters. Returning to our counting example, you can write

```
var count = $('#description').contents().length;
```

You know that just counting elements isn't very useful and we know that you're impatient to get your hands dirty and start creating awesome effects using jQuery. We ask you to wait a few pages in order to allow us to provide you with a solid knowledge base.

The `find()` method is probably one of the most used methods. It lets you search through the descendants of the elements (using a depth-first search) in a set and returns a new jQuery object. This new jQuery object contains all the elements that match a passed selector expression. For example, given a set of matched elements in a variable called `$set`, you can get another jQuery set of all the citations (`<cite>`) within paragraphs (`<p>`) that are descendants of elements in the original set:

```
$set.find('p cite');
```

Like many other jQuery methods, the `find()` method's power comes when it's used within a jQuery chain of operations. This method becomes handy when you need to constrain a search for descendant elements in the middle of a jQuery method chain, where you can't employ any other context or constraining mechanism.

Before listing all the methods belonging to this category, we want to show you another example. Imagine you have the following HTML snippet:

```
<ul>
  <li class="awesome">First</li>
  <li>Second</li>
  <li class="useless">Third</li>
  <li class="good">Fourth</li>
  <li class="brilliant amazing">Fifth</li>
</ul>
```

You want to retrieve all the siblings of the list item having class `awesome` up to but excluding the one having both the classes `brilliant` and `amazing` (the fifth). To perform this task, you can use `nextUntil()`. It accepts a selector as its first argument and retrieves all the following siblings of the elements in the set until it reaches an element matching the given selector. Hence, you can write

```
var $listItems = $('.awesome').nextUntil('.brilliant.amazing');
```

What if you want to perform the same task but retrieve only those having the class `good`? The function accepts an optional second argument, called `filter`, that allows you to achieve this goal. You can update the previous statement, resulting in the following:

```
var $listItems = $('.awesome').nextUntil('.brilliant.amazing', '.good');
```

You can execute this example in your browser by loading the file chapter-3/jquery.nextuntil.html or by accessing the relative JS Bin (http://jsbin.com/fuhen/edit?html,js,console).

Table 3.1 shows this and the other methods that belong to this category and that allow you to get a new jQuery object from an existing one. Most of these methods accept an optional argument, which will be specified by adopting the usual convention of wrapping it with square brackets.

Table 3.1   Methods for obtaining a new set based on relationships to other HTML DOM elements

Method	Description
`children([selector])`	Returns a set consisting of all the children of the elements in the set, optionally filtered by a selector.
`closest(selector [, context])`	Returns a set containing the single nearest ancestor of each element in the set that matches the specified selector, starting from the element itself. As the first argument, an element or a jQuery object can be passed as well. In this case, it will be tested against the ancestors. If found, a set containing it will be returned; an empty array will be returned otherwise. A DOM element can be optionally specified as `context`. In this case, to have a match the ancestor must also be a descendant of this element.
`contents()`	Returns a set of the contents of the elements in the set, which may include text nodes.
`find(selector)`	Returns a set of the descendants of each element in the set, filtered by a given selector, jQuery object, or element.
`next([selector])`	Returns a set consisting of the immediately following sibling of each element in the set of matched elements. If an element is the sibling of more than one element, it's taken only once. If a selector is provided, it retrieves the next sibling only if it matches that selector.
`nextAll([selector])`	Returns a set containing all the following siblings of the elements in the set. If a selector is provided, it retrieves elements only if they match the selector.

Table 3.1   Methods for obtaining a new set based on relationships to other HTML DOM elements *(continued)*

Method	Description
`nextUntil([selector[, filter]])`	Returns a set of all the following siblings of the elements in the set up to but not including the element matched by the selector. If no matches are made to the selector, or if the selector is omitted, all following siblings are selected. In this case, `selector` can be a string containing a selector expression, a DOM node, or a jQuery object. The method optionally accepts another selector expression, `filter`, as its second argument. If it's provided, the elements will be filtered by testing whether they match it.
`offsetParent()`	Returns a set containing the closest relatively, absolutely, or fixedly positioned (in the CSS sense of the terms) parent of the elements in the set.
`parent([selector])`	Returns a set consisting of the direct parent of all the elements in the set. If an element is the parent of more than one element, it's taken only once. If a selector is given, parents are collected only if they match it.
`parents([selector])`	Returns a set consisting of the unique ancestors (an element is selected only once, even if it matches multiple times) of all the elements in the collection. This includes the direct parents as well as the remaining ancestors all the way up to but not including the document root. If a selector is given, ancestors are collected only if they match it.
`parentsUntil( [selector[, filter]])`	Returns a set of all ancestors of the elements in the collection up to but not including the element matched by the selector. If the selector isn't matched or isn't supplied, all ancestors are selected. In this case, `selector` can be a string containing a selector expression, a DOM node, or a jQuery object. The method optionally accepts another selector expression, `filter`, as its second argument. If it's provided, the elements will be filtered by testing whether they match it.
`prev([selector])`	Returns a set consisting of the immediately previous sibling of each element in the set of matched elements. If an element is the sibling of more than one element, it's taken only once. If a selector is provided, it retrieves the previous sibling only if it matches that selector.
`prevAll([selector])`	Returns a set containing all the previous siblings of the elements in the set. Elements can be optionally filtered by a selector.
`prevUntil([selector [, filter]])`	Returns a set of all preceding siblings of the elements in the collection up to but not including the element matched by the selector. If the selector is not matched or is not supplied, all the previous siblings are selected. In this case, `selector` can be a string containing a selector expression, a DOM node, or a jQuery object. The method optionally accepts another selector expression, `filter`, as its second argument. If it's provided, the elements will be filtered by testing whether they match it.
`siblings([selector])`	Returns a set consisting of all siblings of the elements in the set taken only once. Elements can be optionally filtered by a selector.

Now that we've described all these methods, let's look at a concrete example of some of them.

Consider a situation where a button's event handler (which we'll explore in great detail in chapter 6) is triggered with the button element referenced by the `this` keyword within the handler. Such a situation occurs when you want to execute some JavaScript code (for example, a calculation or an Ajax call) when a button is clicked. Further, let's say that you want to find the `<div>` block within which the button is defined. The `closest()` method makes it a breeze:

```
$(this).closest('div');
```

But this would find only the most immediate ancestor `<div>`; what if the `<div>` you seek is higher in the ancestor tree? No problem. You can refine the selector you pass to `closest()` to discriminate which element is selected:

```
$(this).closest('div.my-container');
```

Now the first ancestor `<div>` with the class `my-container` will be selected.

The remainder of these methods work in a similar fashion. Take, for example, a situation in which you want to find a sibling button with a particular `title` attribute:

```
$(this).siblings('button[title="Close"]');
```

What you're doing here is retrieving all the siblings that are `<button>`s and have the title "Close." If you want to ensure that only the first sibling is retrieved, you can employ the `first()` method you learned in this chapter:

```
$(this).siblings('button[title="Close"]').first();
```

These methods give you a large degree of freedom to select elements from the DOM based on their relationships to the other DOM elements. How would you go about adjusting the set of elements that are in a jQuery collection?

3.2.4   Slicing and dicing a set

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

ADDING MORE ELEMENTS TO A SET

You may often find yourself wanting to add more elements to an existing jQuery collection. This capability is more useful when you want to add more elements after applying some method to the original set. Remember, jQuery chaining makes it possible to perform an enormous amount of work in a single statement.

We'll look at some concrete examples of such situations in a moment, but first, let's start with a simpler scenario. Let's say that you want to match all `<img>`s that have either an `alt` or a `title` attribute. The powerful jQuery selectors allow you to express this as a single selector, such as

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

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

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

Using the `add()` method in this fashion allows you to chain a bunch of selectors together, creating a union of the elements that satisfy either of the selectors.

 Methods such as `add()` are also significant (and more flexible than aggregate selectors) within jQuery method chains because they don't augment the original set but create a new set with the result. You'll see in a bit how this can be extremely useful in conjunction with methods such as `end()` (which we'll examine in section 3.2.5) that can be used to back out operations that augment original sets.

 This is the syntax of the `add()` method.

Method syntax: `add`

`add(selector[, context])`

Creates a new jQuery object and adds elements specified by the `selector` parameter to it. The `selector` parameter can be a string containing a selector, an HTML fragment, a DOM element, an array of DOM elements, or a jQuery object.

Parameters

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

`context`   (Selector|Element|jQuery) Specifies a context to limit the search for elements that match the first parameter. This is the same parameter that can be passed to the `jQuery()` function.

Returns

A copy of the original set with the additional elements.

Bring up the jQuery Operations Lab page in your browser and enter this expression:

```
$('td').add('th');
```

Then click the Execute button. This will execute the jQuery operation and select all the cells of the table. Figure 3.4 shows a screen capture of the results.

 In figure 3.4 you can see that all the table cells, including the header ones (`<th>`), were added to the set. The black outline and the gray background highlight the elements captured by your selection. This style is assigned to every element you'll act upon by automatically adding a class called `found-element`.

 Now let's take a look at a more realistic use of the `add()` method. Let's say that you want to apply a red border to all `<img>`s that have an `alt` attribute, adding to them a class called `red-border`. Then you want to apply a level of transparency to all `img`

Figure 3.4 The cells of the table have been matched by the jQuery expression.

elements that have either an `alt` or `title` attribute, adding a class called `opaque`. The comma operator (`,`) of CSS selectors won't help you with this one because you want to apply an operation to a set and then add more elements to it before applying another operation. You 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 expression. To add the cited classes, you'll use a jQuery function called `addClass()`. In its simplest form, it takes a class name as an argument and adds it to the elements in the set.

What we described results in a single statement such as this:

```
$('img[alt]')
  .addClass('red-border')
  .add('img[title]')
  .addClass('opaque');
```

Here you create a set of all `<img>`s that have an `alt` attribute and apply a predefined class that applies a red border. Then you add the `img` elements that have a `title` attribute and finally apply a class that establishes a level of transparency to the newly augmented set.

Enter this statement into the jQuery Operations Lab Page (which has predefined the referenced classes), click the Execute button, and view the results, as shown in figure 3.5.

Figure 3.5   jQuery chaining allows you to perform complex operations in a single statement, as shown in these results.

In these results, you can see that the flower images (those with `alt`) have a red border. In addition, all the 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. As you may note, the images that aren't flowers (except the coffee pot) also have a black border. The reason is that in addition to the `opaque` class, the previously mentioned `found-element` class has been automatically added. Actually, the `found-element` class has been added to *all* the selected images, but the black border for the flower images has been overridden by the style declarations of the `red-border` class.

The `add()` method can also be used to add elements to an existing set, given direct references to those elements. Passing an element reference, or an array of element references, to the `add()` method adds the elements to the set. If you had an element reference in a variable named `someElement`, you could add it to the set of all images containing an `alt` property with this statement:

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

As if that wasn't flexible enough, not only does the `add()` method allow you to add existing elements to the set, but you can also use it to add new elements by passing it a string containing HTML markup. Consider this:

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

This fragment creates a set of all `p` elements in the document and then creates a new set that includes the `<div>` created on the fly.

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

HONING THE CONTENTS OF A SET

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

Let's say that you want to select all `img` elements in a page that have a `title` attribute except for those that contain the text "puppy" as their value. You 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 you forgot about the `:not` filter. By using the `not()` method, which removes any elements from a set that match the specified selector, you can express an *except* type of relationship. To perform the described match, you can write

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

Type this expression into the jQuery Operations Lab Page and execute it. You'll see that only the first dog image has the highlight applied. The black puppy, which is included in the original set because it possesses a `title` attribute, is removed by the `not()` invocation because its `title` contains the text "puppy".

<div align="center">

Method syntax: `not`

</div>

`not(selector)`

Creates a copy of the set without the elements that match the criteria specified by the value of the `selector` parameter.

Parameters

  selector   (Selector|Element|Array|jQuery|Function) Specifies which elements are to be
                    removed. If the parameter is a jQuery selector, the matching elements are
                    removed. If an element reference, array of elements, or a jQuery set is
                    passed, those elements are removed from the set. If a function is passed,
                    the function is invoked for each item in the set (with `this` set to the item),
                    and returning `true` from the invocation causes the item to be removed from
                    the set. In addition, jQuery passes the index of the element inside the set
                    as the first argument of the function, and the current element as the second
                    argument.

Returns

A copy of the original set without the removed elements.

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

When maximum flexibility is needed, you can pass a function to `not()` and make a determination of whether to keep or remove the element on an element-by-element basis. jQuery passes to the function an argument that specifies the index of the element inside the set. Consider this example:

```
$('div').not(function(index) {
  return $(this).children().length > 2 && index % 2 === 0;
});
```

This statement will select all <div>s of the page and then remove those that have more than two children and have an odd index (not position) inside the set. If you want to see a live result, you can try it in the Lab Page and you'll receive four matches.

This method allows you to filter the set in ways that are difficult or impossible to express with a selector expression by resorting to programmatic filtering of the set elements.

For those times when the test applied within the function passed to not() seems to be the opposite of what you want to express, not() has an inverse method, filter(). It works in a similar fashion, except that it removes elements when the function returns false.

For example, let's say that you want to create a set of all <td>s that contain a positive integer value. For such situations, you can employ the filter() method, as follows:

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

This statement creates a set of all td elements and then invokes the function passed to the filter() method for each of them, with the current matched element as the this value for the invocation. The function used employs a regular expression to determine whether the element content matches the described pattern (a sequence of one or more digits), returning null if not. Elements whose filter function invocation returns false, or a *falsy* value in general (null, undefined, and so on), aren't included in the returned set.

The syntax of the filter() method is the following.

Method syntax: `filter`

`filter(selector)`
Creates a copy of the set and removes elements from the new set that don't match the criteria specified by the value of the `selector` parameter.

Parameters

selector	(Selector|Element|Array|jQuery|Function) Specifies which elements are to be removed. If the parameter is a string containing a selector, any elements that don't match are removed. If an element reference, array of elements, or jQuery object is passed, all but those elements are removed from the set. If a function is passed, the function is invoked for each element in the set (with `this` referencing the current element), and returning `false` from the invocation causes the element to be removed from the set. In addition, jQuery passes the index of the element inside the set as the first argument of the function and the current element as the second argument.

Returns
A copy of the original set without the removed elements.

Again, bring up the jQuery Operations Lab Page, type the previous expression in, and execute it. You'll 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 selector expression. When used in this manner, it operates inversely to the corresponding `not()` method, removing any elements that don't match the passed selector. This isn't a super-powerful method, because it's usually easier to use a more restrictive selector in the first place, but it can be useful within a chain of jQuery methods. Consider, for example,

```
$('img')
  .addClass('opaque')
  .filter('[title*="dog"]')
  .addClass('red-border');
```

This chained statement selects all images of a page, applies the `opaque` 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 `red-border`. The result is that all the images end up semitransparent, but only the tan dog gets the red border treatment.

The `not()` and `filter()` methods give you powerful means to adjust a set of elements in a collection on the fly, based on almost any criteria concerning the elements in the set. But you can also subset the set, based on the position of the elements within the set. Let's look at those methods next.

OBTAINING SUBSETS OF A SET

Sometimes you may wish to obtain a subset of a set based on the position of the elements within the set. jQuery provides a `slice()` method to do that. This method creates and returns a new set from any contiguous portion, or a slice, of an original set.

Method syntax: `slice`

`slice(start[, end])`

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

Parameters

start	(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 negative, it indicates an offset from the end of the set. If omitted, the slice extends to the end of the set.

Returns

The newly created set.

If you want to obtain a set that contains a single element from another set, based on its position in the original set, you can employ the `slice()` method. For example, to obtain the third element of a previous selection, you could write

```
$('img, div.wrapper', 'div').slice(2, 3);
```

This statement selects all the `<img>`s, and the `<div>`s having the class `wrapper`, within a `<div>`, and then generates a new set containing only the third element in the matched set. As you can see, your previously acquired knowledge comes back again and again as you advance in the book.

Note that this is different from using `get(2)`, which returns the third DOM element in the set, but is the same as using `eq(2)`.

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 up to the end of the set, the statement

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

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

Another method you can use to obtain a subset of a set is `has()`. Like the `:has` filter, this method tests the children of the elements in a jQuery object, using this check to choose the elements to become part of the subset.

Method syntax: `has`

`has(selector)`

Creates and returns a new set containing only elements from the original set that contain descendants that match the passed `selector` expression.

Parameters

selector (Selector|Element) A string containing a selector to be applied to all descendants of the elements in the set, or a DOM element to be tested. Only elements within the set possessing an element that matches the selector, or the passed element, are included in the returned set.

Returns

A jQuery object.

For example, consider this line:

```
$('div').has('img[alt]');
```

This expression will create a set of all `<div>`s and then create and return a second set that contains only those `<div>`s that contain at least one descendant `<img>` that possesses an `alt` attribute.

TRANSLATING ELEMENTS OF A SET

Often you'll want to perform transformations on the elements of a set. For example, you may want to collect all the IDs of the elements in the set or perhaps collect the values of a set of `form` elements in order to create a query string from them. The `map()` method comes in handy for such occasions.

<div align="center">

Method syntax: `map`

</div>

`map(callback)`

Invokes the `callback` function for each element in the set and collects the returned values into a jQuery object.

Parameters

`callback`	(Function) A `callback` function that's invoked for each element in the set. Two parameters are passed to this function: the zero-based index of the element within the set and the element itself. The element is also established as the function context (the `this` keyword). To add an element into the new set, a value different from `null` or `undefined` must be returned.

Returns

The set of translated values.

For example, the following code will collect all the IDs of all the <div>s on the page:

```
var $allIDs = $('div').map(function() {
  return this.id;
});
```

With this statement you're actually retrieving a jQuery object containing the IDs, which is usually not what you want. If you want to work with a plain JavaScript array, you can append the `toArray()` method to the chain as such:

```
var allIDs = $('div').map(function() {
  return this.id;
})
.toArray();
```

There are other methods that we want to introduce in this chapter. Let's discover more.

TRAVERSING A SET'S ELEMENTS

The `map()` method is useful for iterating over the elements of a set in order to collect values or translate the elements in some other way, but you'll have many occasions where you'll want to iterate over the elements for more general purposes. For these occasions, the jQuery `each()` method is invaluable.

<div align="center">

Method syntax: `each`

</div>

`each(iterator)`

Traverses all the elements in the matched set, invoking the passed `iterator` function for each of them.

Parameters

`iterator`	(Function) A function called for each element in the matched set. Two parameters are passed to this function: the zero-based index of the element within the set and the element itself. The element is also established as the function context (the `this` reference).

Returns

The jQuery collection.

An example of using this method could be to set a property value on all elements in a matched set. For example, consider this:

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

This statement will invoke the passed function for each `img` element on the page, modifying its `alt` property using the index of the element within the set and its ID.

So far you've seen a lot of methods you can use with a jQuery object, but you're not finished yet! Let's see more about how jQuery deals with them.

3.2.5   *Even more ways to use a set*

There are still a few more tricks that jQuery has up its sleeve to let you refine your collections of objects.

Another method that we'll examine allows you to test a set to verify 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` otherwise. Take a look at this example:

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

This statement sets the value of the `hasImage` variable to `true` if the current page has at least one image.

Method syntax: `is`

`is(selector)`
Determines if any element in the set matches the passed selector expression.

Parameters

selector   (Selector|Element|Array|jQuery|Function) The selector expression, the element, an array of elements, or the jQuery object to test against the elements of the set. If a function is provided, it's invoked for each element in the jQuery collection (with `this` set to the item), and returning `true` from the invocation causes the whole function to return `true`. In addition, jQuery passes the index of the element inside the set as the first argument of the function and the current element as the second argument.

Returns

`true` if at least one element matches the passed selector; `false` otherwise.

This is a highly optimized and fast operation within jQuery and can be used without hesitation in areas where performance is of high concern.

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

Now consider the following statement:

```
$('img').filter('[title]').hide();
```

Two sets are generated within this statement: the original set of all the `<img>`s in the DOM and a second set consisting of only those that possess the `title` attribute. (Yes, you could have done this with a single selector, but bear with us for illustration of the concept. Imagine that you do something important in the chain before the call to `filter()`.) Then you hide all the elements in the set (those having the `title` attribute).

But ponder this: what if you subsequently want to apply a method, such as adding a class name, to the original set after it's been filtered? You can't tack it onto the end of the existing chain; that would affect the titled images, not the original set of images.

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

For example, take a look at this statement:

```
$('img')
  .filter('[title]')
  .hide()
  .end()
  .addClass('my-class');
```

The `filter()` method returns the set of the images possessing a `title` attribute. By calling `end()` you back up to the previous set of matched elements (the original set of all images), which gets operated on by the `addClass()` method. Without the intervening `end()` method, `addClass()` would have operated only on the set of the images with the `title` attribute. To avoid that, you should store the initial set in a variable and then write two statements. The `end()` method allows you to get rid of such a variable and perform all the operations in a single statement.

The syntax of the `end()` method is as follows.

Method syntax: `end`

`end()`

Used within a chain of jQuery methods; ends the most recent filtering operation in the current chain and returns the set of matched elements to its previous state

Parameters

Returns

The previous jQuery collection

---

jQuery objects maintain an internal stack that keeps track of changes to the matched set of elements. When a method like those shown so far is invoked, the new set is pushed into the stack. Once jQuery's `end()` method is called, the topmost (most recent) set is popped from the stack, leaving the previous set exposed for subsequent methods to operate upon.

Another handy jQuery method that modifies the cited stack is `addBack()`, which adds the previous set of elements on the stack to the current set, optionally filtered by a selector.

Method syntax: `addBack`

`addBack([selector])`

Adds the previous set of elements on the stack to the current set, optionally filtered by a selector

Parameters

selector   (Selector) A string containing a selector expression to match the current set of
            elements against

Returns

The merged jQuery collection

Consider this:

```
$('div')
  .addClass('my-class')
  .find('img')
  .addClass('red-border')
  .addBack()
  .addClass('opaque');
```

This statement selects all the `div` elements of the page, adds the class `my-class` to them, and creates a new set consisting of all `img` elements that are descendants of those `div` elements. Then it applies class `red-border` to them and creates a third set that's a merger of the `div` elements (because it was the topmost set on the stack) and their descendant `img` elements. Finally, it applies class `opaque` to them.

Whew! At the end of it all, the <div>s end up with classes `my-class` and `opaque`, whereas the images that are descendants of those elements are given classes `red-border` and `opaque`.

This chapter should have proved to you that mastering selectors and the methods to operate on them is important. These features, together with those used to traverse the DOM, allow you to precisely select elements regardless of the complexity of your requirements. Now that you have a solid understanding of this topic, we can delve into more exciting topics.

3.3 *Summary*

This chapter described how to create and augment a set of matched elements using HTML fragments to create new elements on the fly. These orphaned elements can be manipulated, along with any other elements in the set, and eventually attached to parts of the page document.

A set of methods to adjust the set in order to refine the contents of the set, either immediately after creation or midway through a set of chained methods, is available.

Applying filtering criteria to an existing set can also easily create new jQuery collections.

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

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

Working with properties, attributes, and data

This chapter covers
- Getting and setting element attributes
- Working with element properties
- Storing custom data on elements

Everyone who has approached software development for the first time has learned a very important lesson: even huge and complex software consists of a mix of elementary instructions. Summing numbers, counting items, and iterating over elements are just a few examples of these basic operations. In the same way, you can create a nice visual with jQuery by manipulating attributes, properties, classes, styles, and so on.

You can manipulate attributes and properties of elements with JavaScript's native functions, but some tasks aren't as easy as you'd like them to be. In addition, using those functions leaves you with the burden of dealing with browsers' incompatibilities. jQuery provides a full set of methods to easily work with attributes and properties, solving all the compatibility issues for you.

Another key concept when working with DOM elements and the creation of effects is the possibility of storing custom data on the elements you're operating

upon. jQuery allows you to save the state an element is in at a given time. This feature is crucial for creating plugins, as you'll discover in part 3 of this book.

This chapter focuses on the many methods jQuery offers for working with attributes, properties, and data.

4.1   Defining element properties and attributes

When it comes to DOM elements, some of the most basic components you can manipulate are the properties and the attributes assigned to those elements. These properties and attributes are initially assigned to the JavaScript object instances that represent the DOM elements as a result of parsing the HTML markup, and they can be changed dynamically under script control. Let's make sure that you have the terminology and concepts straight.

Properties are intrinsic to JavaScript objects, and each has a name and a value. The dynamic nature of JavaScript allows you to create properties on JavaScript objects under script control. (The appendix goes into great detail on this concept if you're new to JavaScript.)

When referring to *attributes*, we mean the values that are specified on the markup of DOM elements, not the properties of an object instance. Consider the following HTML markup for an image element:

```
<img id="my-image" src="image.gif" alt="An image" class="some-class"
    title="This is an image"/>
```

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. The browser reads and interprets this element markup to create the JavaScript object instance of type HTMLElement that represents this element in the DOM.

The first difference between these two concepts is that the properties' values may be different from their related attributes' values. Whereas the latter are always strings, the corresponding properties' values may be strings, Booleans, numbers, or even objects. For example, trying to retrieve tabindex as an HTML attribute gives you a string (composed of all digits, but still a string). But retrieving its related property gives you a number. Another example is style, which if retrieved as an attribute is a string but if retrieved as a property is an object (of type CSSStyleDeclaration). To see this difference in action, let's say that you have the following HTML element in a web page:

```
<input id="surname" tabindex="1" style="color:red; margin:2px;" />
```

Now create a script in the same page made of the following statements or type them directly in a browser's console:

```
var element = document.getElementById('surname');
console.log(typeof element.getAttribute('tabindex'));   ⟵——— Prints "string"
console.log(typeof element.tabIndex);   ⟵——— Prints "number"
```

```
console.log(element.getAttribute('style'));
console.log(element.style);
```

**Prints the string
"color:red; margin:2px;"**

**Prints the CSSStyleDeclaration object
containing all the styles applied to the element**

All the attributes of an element are gathered into an object, which is stored as a property named, reasonably enough, `attributes` on the DOM element instance. In addition, the object representing the element is given a number of properties, including some that represent the attributes of the element's markup.

As such, the attribute values are stored not only in the `attributes` property but also in a handful of other properties. Figure 4.1 shows a simplified overview of this process.

HTML markup

```
<img id="my-image" src="image.gif" alt="An image" class="some-class" title="This is an image" />
```

img **element**

| attributes |
| id:'my-image' |
| src:'http://localhost/image.gif' |
| alt:'An image' |
| className:'some-class' |
| title:'This is an image' |

Other properties ...

NodeList

| id='my-image' |
| src='image.gif' |
| alt='An image' |
| class='some-class' |
| title='This is an image' |

Other implicit or defaulted attributes ...

Legend

— → Direct reference

◄ ------ ►► Value correspondence
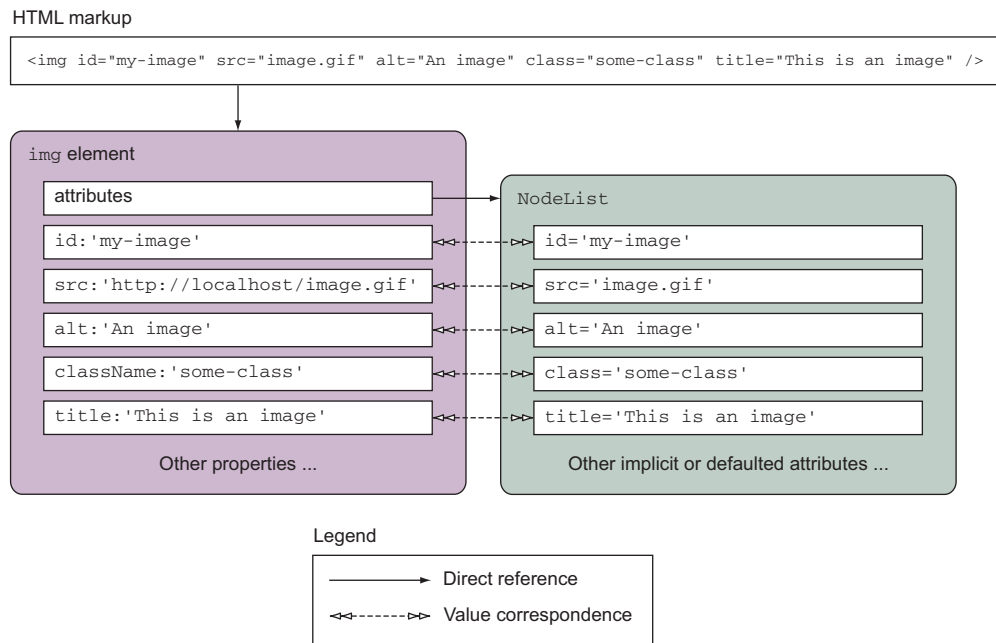
Figure 4.1   HTML markup is translated into DOM elements, including the attributes of the tag and properties created from them. The browser creates a correspondence between the attributes and properties of the elements.

An active connection remains between the attribute values stored in the `attributes` object and the corresponding properties. Changing an attribute value usually results in a change in the corresponding property value and vice versa. To be more specific, consider the following `input` element (a `checkbox`, to be precise) with an additional, nonstandard attribute (in bold):

```
<input type="checkbox" id="book" name="book" title="Check this!"
       book="jQuery in Action" />
```

The following are true:

- If the attribute exists as a built-in (native) property of the corresponding DOM object, the value is synchronized. For example, `title` is a standard attribute and exists in the DOM element representing an image. Therefore, any change of its value will result in an update in the related property and vice versa.
- If the attribute exists as a built-in property but it's a Boolean, the value isn't synchronized. For example, `checked` retrieved as an attribute gives you the initial state of the check box element (`null` if not defined, as in our element). If retrieved as a property, regardless of whether it was defined or not, you obtain a Boolean (`true` if checked, `false` otherwise) of the current state of the element.
- If the attribute doesn't exist as a built-in property, it won't be created and the value won't be synchronized. For example, the `book` attribute won't be created as a property of the DOM element.

To test this synchronization idea, consider the previous `checkbox` element and the following statements:

```
var checkbox = document.getElementById('book');
console.log(checkbox.getAttribute('title') === checkbox.title);

checkbox.title = 'New title!';
console.log(checkbox.getAttribute('title') === checkbox.title);

checkbox.setAttribute('title', 'Another title!');
console.log(checkbox.getAttribute('title') === checkbox.title);

console.log(checkbox.getAttribute('checked') === checkbox.checked);
```

All of the `console.log()` calls but the last one print `true` on the console, confirming what we asserted.

> **WARNING**   Once again we want to highlight that the `console.log()` method isn't supported by old versions of Internet Explorer (IE 6–7). In the examples of this book we'll ignore this issue and we'll use it heavily to avoid resorting to the annoying `window.alert()` method. But you should keep in mind this lack of support in case your code needs to support these browsers.

The previous example can be found in the file name chapter-4/attributes.and .properties.html of the source provided with this book and as a JS Bin (http:// jsbin.com/ soqexa/edit?html,js,console).

In addition to the properties, values are not always identical, either. 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 (that is, `http://www.yourdomain .com/image.gif`).

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

Detecting support for an attribute

HTML5 introduced several new attributes that you can add into your page's elements. The difference between attributes defined in the markup but not present in the DOM element generated by the browser comes in handy when you need to detect the support for these attributes. Take as an example the new HTML5 `required` attribute. When the user adds it to a form field, the browser requires the user to enter some data into that field before the form can be submitted. You can detect if the browser supports this attribute by writing

```
if ("required" in document.createElement("input")) {
    // Attribute supported
}
```

This test will return `true` if the browser supports the feature; `false` otherwise.

If you need to detect the support for many features, we suggest you use Modernizr, the library that we mentioned in chapter 1.

jQuery gives you the means to easily manipulate an element's attributes and provides access to the element instance so that you can also change its properties. Which of these you choose to manipulate depends on what you want to do. Let's start by looking at getting and setting element attributes.

4.2 Working with attributes

In this section we'll delve into the world of attributes and the methods jQuery offers to work with them.

4.2.1 Fetching attribute values

In jQuery, to get and set a value of an attribute you can use the `attr()` method. This is a typical peculiarity of jQuery. As you'll often see, the same method can be used either to read or to write a value. In other words, the method can work as a getter or a setter. What action jQuery will perform depends on the number and types of the parameters passed. The `attr()` method can be used to either fetch the value of an attribute from the first element in the matched set or to set attribute values onto all matched elements.

The syntax for the fetch variant of the `attr()` method (the method as a getter) is as follows.

Method syntax: attr

`attr(name)`

Obtains the value 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 you usually think of element attributes as those predefined by HTML, you can use `attr()` with custom attributes set through JavaScript or HTML markup. As you already saw in chapter 2, to add a custom attribute you can use the new HTML5 `data-*` attribute. To illustrate this, consider the following `img` element with a custom attribute (highlighted in bold):

```
<img id="my-image" src="image.gif" alt="An image" alt="A beautiful image"
    data-custom="some value" />
```

Note that we've added a custom attribute, unimaginatively named `data-custom`, to the element. You can retrieve that attribute's value as if it was any of the standard attributes, with

```
$('#my-image').attr('data-custom');
```

Attribute names aren't case-sensitive in HTML, so regardless of how an attribute such as `title` is declared in the markup, you can access (or set, as you'll see) attributes using any variant of case: `Title`, `TITLE`, `TiTlE`, and any other combinations are all equivalent. In XHTML, even though attribute names must be lowercase in the markup, you can retrieve them using any case variant.

The set variant of `attr()` has some handy features of its own. Let's take a look.

4.2.2   *Setting attribute values*

There are two ways to set attributes onto selected elements with jQuery. Let's start with the most straightforward, which allows you to set a single attribute at a time for all the elements retrieved. Its syntax is as follows.

Method syntax: attr

`attr(name, value)`
Sets the named attribute to the passed value for all elements in the jQuery object.

Parameters

name	(String) The name of the attribute to be set.
value	(String|Number|Boolean|Function) Specifies the value of the attribute. This can be any JavaScript expression that results in a value of the type specified. Unless a function is provided, any other value will be converted to a string. The function is invoked for each element in the set, passing the index of the element and the current value of the named attribute on the element. The return value of the function becomes the attribute value.

Returns

The jQuery collection.

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

In its most basic form, the `value` parameter can be any JavaScript expression that results in a value that will be converted into a string. Things get more interesting when the `value` parameter is an inline function or a reference of a function. In such cases,

the function is invoked for each element retrieved, with the return value of the function used as the attribute value. When the function is invoked, it's passed two parameters: one that contains the zero-based index of the element within the set and one that contains the current value of the named attribute of the element. Additionally, the element is established as the function context (`this`) 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:

```
$('[title]').attr('title', function(index, previousValue) {
  return previousValue + ' I am element ' + index +
         ' and my name is ' + (this.id || 'unset');
});
```

This method will run through all the elements on the page having a `title` attribute. It'll modify the `title` attribute of each element by appending to the previous value a string composed using the index of the element within the DOM and the `id` attribute of each specific element, if available, or the string `'unset'` otherwise.

You'd use this means of specifying the attribute value whenever that value is dependent on other aspects of the element, when you need the *original* value to compute the new value, or whenever you have other reasons to set the values individually.

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

Method syntax: attr

attr(attributes)
Uses the properties and values specified by the passed object to set corresponding attributes on all elements of the matched set

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

Returns
The jQuery collection

This format is a quick and easy way to set multiple attributes on all the elements of a 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 this:

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

This statement sets the `value` of all `input` elements to the empty string and 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 similarly to the previous format of `attr()`; the function is invoked for each individual element in the jQuery object.

> **WARNING**   Attempting to change the `type` attribute on an `input` or `button` element created via `document.createElement()` will throw an exception on Internet Explorer 6–8.

Now that you know how to get and set attributes, what about getting rid of them?

4.2.3   *Removing attributes*

In order to remove attributes from DOM elements, jQuery provides the `removeAttr()` method. Its syntax is as follows.

Method syntax: removeAttr

`removeAttr(name)`

Removes the specified attribute or attributes from every matched element

Parameters

name   (String) The name of the attribute or list of attributes' names separated by a
              space to remove

Returns

The jQuery collection

Let's see an example of this method in use. The goal is to remove the `title` and `alt` attributes from all the images in a page. To perform this task, you can write the following statement:

```
$('img').removeAttr('title alt');
```

The `removeAttr()` method internally uses the JavaScript `removeAttribute()` function. But it has the advantage of being called directly on every element in a jQuery object and allowing the use of chaining.

Removing an attribute doesn't remove any corresponding property from the JavaScript DOM element, although 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 wouldn't be removed from the element.

Now let's look at some examples of how you might apply this knowledge to your pages.

4.2.4   *Fun with attributes*

Let's see how these methods can be used to fiddle with the element attributes in various ways.

EXAMPLE #1—FORCING LINKS TO OPEN IN A NEW WINDOW

Imagine that you want to make all the links on your website that point to external domains open in new windows. This is fairly trivial if you're in total control of the entire markup and can add a `target` attribute, as shown here:

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

That's all well and good, but what if you're not in control of the markup? You could be running a content management system or a wiki, where end users are able to add content, and you can't rely on them to add the `target="_blank"` to all external links. First you need to determine what you want: you want all links whose `href` attributes begin with `http://` to open in a new window, which you've determined can be done by setting the `target` attribute to `_blank`. In addition, you don't want to take into account those links that already have the `target` attribute set with the right value. For the sake of simplicity, in this example we're deliberately ignoring other protocols such as FTP, HTTPS, and the like.

You can use the techniques you've learned in this section to do this concisely:

```
$('a[href^="http://"]')
    .not('[target="_blank"]')
    .attr('target', '_blank');
```

First you select all the links with an `href` attribute starting with `http://`, which indicates that the reference is external (assuming the page isn't using absolute URLs for internal resources). Then you exclude all those links that already have their `target` attribute set to `_blank`. Finally, you set the attribute to the remaining elements. Mission accomplished with a single line of jQuery code! You can see this code in action by opening the file chapter-4/new.window.links.html.

EXAMPLE #2—SIMULATING THE PLACEHOLDER ATTRIBUTE

Another excellent use for jQuery's attribute functionality is to simulate the new HTML5 `placeholder` attribute. In this example, we'll show you a basic implementation to simulate it that well serves to show the use of the `attr()` method, but it isn't intended for use in production. There are many reasons why you shouldn't use this code in your projects. The first reason is that you won't test for browser support, so running this code will perform the operation on browsers that support the `placeholder` attribute as well. Another reason is that the text won't be hidden once the field is focused, so the user has to delete the value manually. What a pain!

The placeholder attribute

The `placeholder` attribute shows text, which ideally specifies what the field is for or an example of a possible value, inside a field. The text is shown until the field is focused or a value has been inserted by the user, depending on the browser, in which case the text is hidden. This attribute applies to `<input>`s and `<textarea>`s. On older browsers that don't support it (Internet Explorer 6–9) this attribute is ignored and nothing happens, like you never wrote the attribute in first place.

What you'll do in this example is copy the value of the placeholder attribute and set it as the value of the value attribute. By doing so, every browser will show the text of the placeholder even if placeholder isn't supported. For the sake of the example, we'll cover only the case of <input> elements. The reason is that in order to target <textarea>s too, you'd need jQuery's text() method, which we haven't covered yet (but will do so shortly).

Assume that you have the following form:

```
<form>
   <label for="username">Username:</label>
   <input id="username" name="username" placeholder="JohnDoe" />
   <label for="email">Email:</label>
   <input type="email" id="email" name="email" placeholder="email@fake.com" />
   <input type="submit" value="Login" />
</form>
```

To satisfy the requirements, you can write this simple code:

```
$('input').each(function(index, element) {         ◁——— Selects all <input>s and
   var $element = $(element);                              iterates over them
   $element.attr('value', $element.attr('placeholder'));◁——— Creates a new
});                                                            jQuery object
                                                              containing the
                                                              current element
              Copies the value of placeholder as the         and stores it in a
                   value of the value attribute              variable
```

The result of this execution is shown in figure 4.2. In addition, the code can be found in the file chapter-4/placeholder.html and as a JS Bin (http://jsbin.com/onuMiDU/edit?html,js,output).

Username: JohnDoe   Email: email@fake.com   Login

Figure 4.2   The fields of the form are filled with the values specified in the placeholder attribute using the attr() method.

Now that you've seen the ins and the outs of dealing with attributes, let's take a look at how to manipulate the properties of an element.

4.3   *Manipulating element properties*

As with the attr() method, to get and set a value of a property in jQuery you can use a similar approach: the prop() method. attr() and prop() are similar in their capabilities and in the parameters they accept. Besides, for those of you who like anecdotes, the prop() method was extracted from attr() as a way to reorganize the latter and to let it focus solely on attributes. Prior to version 1.6, attr() dealt with both attributes and properties. But that's a very old story.

The syntax of the `prop()` method to retrieve the value of a property is as follows.

Method syntax: prop

`prop(name)`

Obtains the value of the given property for the first element in the matched set.

Parameters

name (String) The name of the property whose value has to be retrieved.

Returns

The value of the property for the first matched element. The value `undefined` is returned if the value of a property hasn't been set or if the matched set has no elements.

Consider once again a `checkbox` element:

```
<input type="checkbox" id="legal-age" name="legal-age" title="Check this!" />
```

You can verify if it's checked with the use of the `prop()` method. In this example pretend that you completely forgot about the `:checked` filter and the `is()` method:

```
if ($('#legal-age').prop('checked') === false) {
  console.log('The checkbox is currently unchecked');
}
```

The jQuery `prop()` method 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, which is also used by `attr()`, is shown in table 4.1.

Table 4.1 jQuery `prop()` normalized-access names

jQuery normalized name	DOM name
cellspacing	cellSpacing
cellpadding	cellPadding
class	className
colspan	colSpan
contenteditable	contentEditable
for	htmlFor
frameborder	frameBorder
maxlength	maxLength
readonly	readOnly
rowspan	rowSpan
tabindex	tabIndex
usemap	useMap

Knowing if a certain check box is checked or not is nice, but what if you want to pro-grammatically mark it as checked? You can achieve this goal employing the `prop()` method as a setter. The syntax of its first variant as a setter is shown here.

Method syntax: prop

`prop(name, value)`

Sets the named property to the given value for all elements in the jQuery collection.

Parameters

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

value   (Any|Function) Specifies the value of the property. This can be any JavaScript
            expression that results in a value or it can be a function. The function is
            invoked for each element, passing the index of the element in the jQuery
            collection and the current value of the named attribute for that particular
            element. The return value of the function becomes the property value.

Returns

The jQuery collection.

You can programmatically mark as checked a given check box as follows:

```
$('#legal-age').prop('checked', true);
```

As we pointed out, `attr()` and `prop()` have a lot in common, and the variant where you can specify multiple properties at once is no exception. The syntax of this varia-tion is presented here.

Method syntax: prop

`prop(properties)`

Uses the properties and values specified by the given object to set corresponding properties on all elements of the matched set

Parameters

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

Returns

The jQuery collection

This format allows you to quickly set multiple properties, avoiding a long chain of sin-gle calls to `prop()`. For example, you can write the following:

```
$('input:checkbox').prop({
    disabled: true,
    checked: true
});
```

The last method to discuss in regard to property management is `removeProp()`. It removes a property set by using the `prop()` method for all the elements selected. Its syntax is shown here.

Method syntax: removeProp

`removeProp(name)`

Removes the specified property from every element in the jQuery collection

Parameters

`name`   (String) The name of the property to remove

Returns

The jQuery collection

Unlike `removeAttr()`, this method doesn't allow for a list of space-separated names. This method shouldn't be used to remove native properties such as `checked` or `required` because it'll completely remove the property. Once removed, it can't be added again to the element. If you wish to change the current status of one of those properties, set the value to `false` using `prop()`.

Element attributes and properties are useful concepts for data as defined by HTML and the W3C, but in the course of page authoring, you frequently need to store your own custom data. Let's see what jQuery can do for you on that front.

4.4   *Storing custom data on elements*

Let's just come right out and say it: global variables suck. Except for the infrequent, truly global values, it's hard to imagine a worse place to store information that you'll need while defining and implementing the complex behavior of your pages. Not only do you run into scope issues, but they also don't scale well when you have multiple operations occurring simultaneously (menus opening and closing, Ajax requests firing, animations executing, and so on).

The functional nature of JavaScript can help mitigate this through the use of closures (if you need a refresher on this topic, read the appendix), but closures aren't appropriate for every situation.

Because your page behaviors are so element-focused, it makes sense to use the elements themselves as storage scopes. Again, the nature of JavaScript, with its ability to dynamically create custom properties on objects, can help you out here. But you must proceed with caution. Because DOM elements are represented by JavaScript object instances, they, like all other object instances, can be extended with custom properties of your own choosing. But dragons await!

These custom properties, so-called *expandos*, aren't without risk. In particular, it's easy to create circular references that can lead to serious memory leaks—for example, keeping a reference to an element you don't need anymore. In traditional web applications, where the DOM is dropped frequently as new pages are loaded, memory leaks may not be as big an issue. But for authors of highly interactive web applications that employ lots of script on pages that may hang around for quite some time, memory leaks can be a huge problem.

jQuery comes to your rescue by providing a means to tack data onto any DOM element that you choose, in a controlled fashion, without relying on potentially problematic expandos. You can place any arbitrary JavaScript value, even arrays and objects, on DOM elements by using the cleverly named `data()` method. This is the syntax.

Method syntax: data

`data(name, value)`

Adds the passed value to the jQuery-managed data store for all elements in the set

Parameters

   name   (String) The name of the data to be stored

   value   (Any) The value to be stored except `undefined`

Returns

The jQuery collection

The `data()` method treats camel-case names the same way as dashed (or hyphenated) names. The following statement

```
$('.class').data('lastValue');
```

is the equivalent of

```
$('.class').data('last-value');
```

Besides, unlike the `attr()` method that stores values always as strings, `data()` is able to keep the value's type. `data()` also tries to convert an attribute's value to its native type when used as a getter. Let's look at an example.

Let's say you have this code in your page:

```
$('.class').attr('last-value', 10);
console.log(typeof $('.class').attr('last-value'));     ⟵———— Prints "string"
```

You'll see `"string"` on the console because `attr()` has converted the number 10 into its string equivalent (`"10"`). On the other hand, if you write

```
$('.class').data('last-value', 10);
console.log(typeof $('.class').data('last-value'));     ⟵———— Prints "number"
```

you'll see `"number"` on the console because `data()` keeps the value's type as is.

Now imagine you have the following HTML code:

```
<input id="name" name="name" data-mandatory="true" />
```

You'll obtain different results using `attr()` and `data()`, as follows:

```
console.log(typeof $('#name').attr('data-mandatory'));   ⟵———— Prints "string"
console.log(typeof $('#name').data('mandatory'));        ⟵———— Prints "boolean"
```

The `attr()` method retrieves the value as a string, so by printing its type you obtain `"string"`. On the other hand, `data()` is able to convert the value to a Boolean (the same applies to `number`, `null`, and so on), so you see `"boolean"`.

It's worth noting that `undefined` isn't recognized as a value but still returns a jQuery object. Therefore, a statement such as

```
$('#name').data('mandatory', undefined);
```

doesn't modify the value of `mandatory`, but it returns the jQuery object that it was called on, which allows for method chaining.

Having the ability to add new data to an element is nice, but so far you're limited to adding one item of data at a time, which isn't very practical. Fortunately, jQuery has a variant of `data()` as a setter that accepts an object of key-value pairs.

Method syntax: data

`data(object)`
Adds the key-value pairs of the given object to the jQuery-managed data store for all elements in the set

Parameters
`object`	(Object) An object of key-value pairs to be stored

Returns
The jQuery collection

We'll continue our exploration of the `data()` method soon, but first we want to mention that jQuery also provides a utility function of the `jQuery` object that's called and acts in the same way as the previously explained `data()` method.

`jQuery.data()` (or equivalently `$.data()`) is a low-level method because it acts on a DOM element instead of a jQuery object. This method accepts the same parameters of `data()` but introduces a new one (first in the list of parameters) where you can pass the DOM element on which you want to store the data.

To give you an idea of their differences, let's say that you have an element having `book` as its ID and you want to store a given value using `$.data()` (the method that doesn't work with a jQuery object). You can do this as shown here:

```
$.data(document.getElementById('book'), 'price', 10);
```

If using `data()` (the method that works with a jQuery object), you'd call it as follows:

```
$('#book').data('price', 10);
```

It should be no surprise that the `data()` method is also able to read data, not just write it. Here's the syntax for retrieving data using the `data()` method.

Method syntax: data

`data([name])`

Retrieves any previously stored data or an HTML5 `data-*` attribute with the specified name on the first element of the set. If a name isn't specified, the method returns an object containing all the previously stored data.

Parameters

name   (String) The name of the data to be retrieved.

Returns

The retrieved data, or `undefined` if not found.

The behavior of the `data()` method as a getter is interesting and may lead to some confusion, so it deserves further discussion.

The getter form of this method helps in retrieving data stored in memory using its setter version. If the previously stored data isn't found, the method searches for a `data-*` attribute of the HTML element with the same name given. Once `data()` retrieves a value from a `data-*` attribute, the method stores that value in the jQuery-managed data store (consider it as internal memory jQuery uses to keep track of all sorts of things). Therefore, any following call to the method will no longer retrieve the value from the attribute, even if you changed the latter using the `attr()` method, because it's now stored in the jQuery memory. In case the attribute isn't found, `undefined` is returned. Figure 4.3 shows the described flow to help you visualize this.
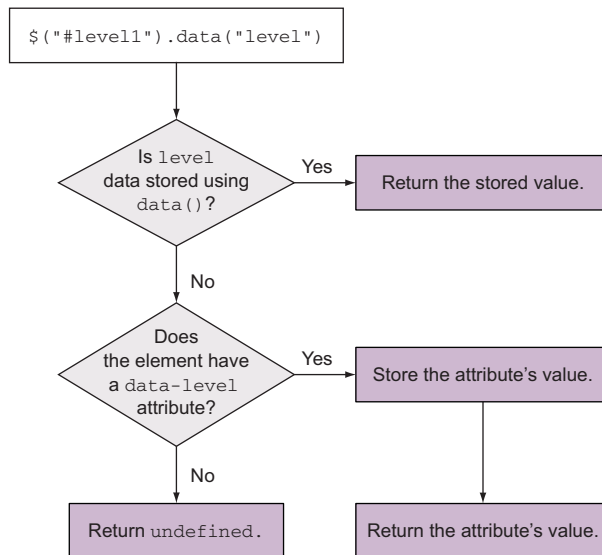
Figure 4.3   How `data()` searches for stored information. The getter form of this method helps you retrieve data stored using its setter version. If the required stored data isn't found, the method searches for a `data-*` attribute of the HTML element with the same name given. If this attribute isn't found, `undefined` is returned.

This process can be tricky, so we'll look at some examples. Let's say that you have the following element:

```
<input id="level1" type="text" value="I'm a text!" data-custom="foo" />
```

You want to retrieve the value of data-custom. You can do that using either the data() or the attr() method but with different parameters. Here's how:

```
console.log($('#level1').data('custom'));
console.log($('#level1').attr('data-custom'));
```
Both print "foo", but the methods need different parameters.

Both previous statements print the string "foo" on the console. But what happens if you use the data() method as shown here?

```
$('#level1').data('custom', 'new value');
```
⟵ Updates the value of the data attribute

```
console.log($('#level1').data('custom'));
console.log($('#level1').attr('data-custom'));
```
Prints the value using data(). The string printed is "new value".

Prints the value using attr(). The string printed is "foo".

You changed the value of custom using the data() method and tried to retrieve it using both the getter version of data() and attr(). This time the result is different! Sounds crazy? You're not finished yet.

As our last example, imagine that you have the previous element without any previously added data because you've just loaded the page. This time you want to prove that once the value of an attribute of an HTML element is retrieved for the first time using data(), the value managed by data() is completely independent from the attribute and thus any future call to attr(). To see this behavior in action, you'll retrieve the value of the attribute using data(), then you'll change the attribute using attr(), and finally you'll invoke attr() and data() to highlight the difference. This description is implemented by the following code:

Both these lines print "foo".

```
console.log($('#level1').data('custom'));
console.log($('#level1').attr('data-custom'));

$('#level1').attr('data-custom', 'new value');

console.log($('#level1').data('custom'));
console.log($('#level1').attr('data-custom'));
```
Sets the value of the data-custom attribute to "new value"

Prints "foo" because it was stored in memory (after invoking data() as a getter the first time)

Prints the value "new value"

If you want to play with this example to better understand the concept, you can open the file chapter-4/getting.and.setting.data.html or access the relative JS Bin we've created (http://jsbin.com/uHAzIyoD/edit?html,js,console).

The `jQuery.data()` method we introduced previously can also be used to retrieve the stored data. To do that, you need to pass the DOM element on which you want to operate as the first parameter and the name of the data to retrieve as the second. In this case, too, the name of the data is optional. If you omit it, jQuery retrieves all the data stored for the given DOM element. The `jQuery.data()` utility function can be used to retrieve data stored using the `data()` method and vice versa.

jQuery 3: Bug fixed

jQuery 3 fixes a bug of the `data()` method that occurred when dealing with attributes with digits in the name—for example, if you have the following element:

```
<div id="name" data-foo-42="test"></div>
```

If you're using a version of jQuery prior to 3 and you write

```
console.log($('#name').data());
```

instead of an object containing the property `foo-42` with a value of `test`, you obtain an empty object.

jQuery 3: Feature changed

jQuery 3 changes the behavior of the `data()` method to align it to the Dataset API specifications (http://www.w3.org/TR/html5/dom.html#dom-dataset). In particular, the main change is that jQuery will transform all the properties' key to be camel case. To understand this change, consider the following element:

```
<div id="name"></div>
```

If you're using a version of jQuery prior to 3 and you write

```
$('#name').data({'my-property': 'hello'});
console.log($('#name').data());
```

you'll obtain the following result on the console:

```
{my-property: "hello"}
```

In jQuery 3, you'll obtain this result instead:

```
{myProperty: "hello"}
```

Note how in jQuery 3 the name of the property is in camel case while in jQuery 1.11 and 2.1 it contains a dash. In case you want to learn more about this change, you can refer to the related issue on GitHub: https://github.com/jquery/jquery/issues/2257.

jQuery not only has methods to set and get data, but in the interests of proper memory management, it also provides the `removeData()` method as a way to dump any data that may no longer be necessary. This method allows you to remove values that were previously set using `data()`, but it doesn't affect any HTML5 `data-*` attribute of the element (you have to use `removeAttr()` for this). The syntax of this method is shown here.

Method syntax: removeData

`removeData([name])`

Removes any previously stored data with the specified name on all elements of the jQuery object. The parameter can also be an array or a string of space-separated names to remove. If no arguments are given, all values are removed.

Parameters

name   (String|Array) A string containing the name or names separated by a space of the data to be removed. If an array is provided, its elements are used to search the names of the data to remove.

Returns

The jQuery collection.

Based on what you've just learned, to remove all the data stored for a given element you can write the following:

```
$('#legal-age').removeData();
```

If you want to remove the `foo` and `bar` data, you can write

```
$('#legal-age').removeData(['foo', 'bar']);
```

or

```
$('#legal-age').removeData('foo bar');
```

Note that it's not necessary to remove data manually when removing an element from the DOM with jQuery methods; the library will smartly handle that for you.

As for the `data()` method, jQuery provides a utility function equivalent to `removeData()`. This utility is called `jQuery.removeData()` (or `$.removeData()`, using the jQuery alias), and it accepts as its first parameter the DOM element on which you want to remove the data, and optionally the name of the data to remove as its second parameter. So, if you want to employ `$.removeData()` to remove all the data stored on the element having `legal-age` as its ID, you'll have to write

```
$.removeData(document.getElementById('legal-age'));
```

In addition to the `jQuery.removeData()` method, jQuery has another utility function that deals with data stored on an element. This method, called `jQuery.hasData()`, allows you to test whether a DOM element has any jQuery data associated with it. Its syntax is as follows.

Method syntax: jQuery.hasData

`jQuery.hasData(element)`

Determines whether an element has any associated data

Parameters

element   (Element) A DOM element to be checked for data

Returns

`true` if there's any data associated with the element; `false` otherwise

---

To see how you can use it, you'll test for the presence of any data on an element having `legal-age` as its ID, store some data, and then run the test again. Thus, you expect the first test to return `false` and the second to return `true`. Let's see `jQuery` `.hasData()` in action:

```
$.hasData(document.getElementById('legal-age'));
$.data(document.getElementById('legal-age'), 'count', 10);
$.hasData(document.getElementById('legal-age'));
```

Returns false; no data stored for this element

Stores a value (10) associated with the name count

Returns true because of the previous statement

We'll exploit the capability to tack data onto DOM elements to our advantage in many of the examples in the upcoming chapters. But if you've run into the usual headaches that global variables can cause, it's easy to see how storing data in context within the element hierarchy opens up a whole new world of possibilities. In essence, the DOM tree has become a complete "namespace" hierarchy for you to employ; you're no longer limited to a single global space.

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

4.5   *Summary*

In this chapter we've gone beyond the art of selecting elements and started manipulating their attributes and properties. In these pages you discovered the differences between attributes and properties and how you can operate on them using jQuery. In addition, you learned how to perform basic tasks—for example, forcing external links to be opened in a new page—just by adding or changing an attribute.

Another important role methods played in this chapter was to manage custom data. As you've seen, some methods have an interesting behavior that can be confusing. Hopefully we've cleared up any confusion by the examples we used.

Updating or removing properties, data, and attributes may be useful, but you need further exploration to see the real power of jQuery. You also need to understand concepts such as moving elements, creating new elements to wrap others, and handling events. In the next chapter we'll delve into some of these topics.

Bringing pages to life with jQuery

This chapter covers

- Manipulating element class names
- Setting the content of DOM elements
- Getting and setting form element values
- Cloning DOM elements
- Modifying the DOM tree by adding, moving, or replacing elements

Today web developers and designers know better than those of 10 years ago (or even themselves 10 years ago) and use the power given to them by DOM scripting to *enhance* a user's web experience, rather than showcase annoying tricks made of blinking texts and animated GIFs. 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, DOM manipulation has allowed many web developers to amaze (not annoy) their users.

On an almost daily basis, many of us come across web pages that do something that makes us say, "Hey! I didn't know you could do that!" And being the commensurate professionals that we are (not to mention being insatiably curious about

such things), we immediately start looking at the source code to find out how they did it. That's the beauty of the web, where you can see other developers' code at any time, isn't it?

But rather than having to code up all that script ourselves, we find that jQuery provides a robust set of tools to manipulate the DOM, making those types of "Wow!" pages possible with a surprisingly small amount of code. Whereas the previous chapter introduced you to working with properties, attributes, and data with jQuery, this chapter discusses how to perform operations on DOM elements to bring them to life and to bring that elusive "Wow!" factor to your pages.

5.1   Changing element styling

In the previous chapter, we mentioned that the `className` property is an example of a case where markup attribute names differ from property names. But, truth be told, class names are a bit special in other respects as well and are handled as such by jQuery. This section describes a better way to deal with class names than by directly accessing the `className` property or using the jQuery's `attr()` method.

When you want to change the styling of an element, there are two options used more often than others. The first is that you can add or remove a class, causing a restyle of the element based on its new or removed class. The other is that you can operate on the DOM element by applying styles directly.

Let's start by taking a look at how jQuery makes it simple to perform changes to an element's style via classes.

5.1.1   Adding and removing class names

The `class` attribute of HTML elements is crucially important to the creation of interactive interfaces. In HTML, the `class` attribute is used to supply these names as a space-separated string. You can have as many spaces as you want, but people usually use one. For example, you may have

```
<div class="some-class my-class    another-class"></div>
```

Unfortunately, rather than manifesting themselves as an array of names in the DOM element's corresponding `className` property, the class names appear as that same space-delimited string. How disappointing, and how cumbersome! This means that whenever you want to add class names to or remove class names from an element that already has class names, you need to parse the string to determine the individual names when reading it and be sure to restore it to a valid space-separated format when writing it.

Taking the cue from jQuery and other similar libraries, HTML5 has introduced a better way to manage this task through an API called `classList`. The latter has more or less the same methods exposed by jQuery, but unfortunately, unlike their jQuery counterparts, these native methods can work on only one element at a time. If you want to add a class to a set of elements, you have to iterate over them. In addition,

being a new introduction, it isn't supported by older browsers, most notably Internet Explorer 6–9. To better understand this difference, consider this code written in pure JavaScript that selects all elements having class `some-class` and adds the class `hidden`:

```
var elements = document.getElementsByClassName('some-class');
for(var i = 0; i < elements.length; i++) {
    elements[i].classList.add('hidden');
}
```

The previous snippet is compatible only with modern browsers, including Internet Explorer 10 and above. Now compare it with its jQuery equivalent:

```
$('.some-class').addClass('hidden');
```

The jQuery version is not only shorter but is also compatible starting with Internet Explorer 6 (depending on the version of jQuery you'll use, of course)!

> **NOTE** The list of class names is considered *unordered*; that is, the order of the names within the space-delimited list has no semantic meaning.

Although it's not a monumental activity to write code that handles the task of adding and removing class names from a set of elements, it's always a good idea to abstract such details behind an API that hides the mechanical details of such operations. Luckily, you don't have to develop your own code because jQuery has already done that for you.

Adding class names to all the elements of a set is an easy operation with the following `addClass()` method that was used in the previous snippet.

Method syntax: addClass

`addClass(names)`

Adds the specified class name(s) to all the elements in the set. If a function is provided, every element of the set is passed to it, one at a time, and the returned value is used as the class name(s).

Parameters

`names`	(String|Function) Specifies the class name, or a space-delimited string of names, to be added. If a function, the function is invoked for each element, with that element set as the function context (`this`). The function is passed two values: the element index and the element's current class value. The function's returned value is used as the new class name or names to add to the current value.

Returns

The jQuery collection.

Removing class names is just as straightforward with the following `removeClass()` method.

Method syntax: removeClass

`removeClass(names)`

Removes the specified class name(s) from each element in the jQuery collection. If a function is provided, every element of the set is passed to it, one at a time, and the returned value is used to remove the class name(s).

Parameters

names   (String|Function) Specifies the class name, or a space-delimited string of names, to be removed. If a function, the function is invoked for each element, setting that element as the function context (`this`). The function is passed two values: the element index and the class value prior to any removal. The function's returned value is used as the class name or names to be removed.

Returns

The jQuery collection.

To see when the `removeClass()` method is useful, let's say that you have the following element in a page:

```
<p id="text" class="hidden">A brief description</p>
```

You can remove the hidden class with this simple statement:

```
$('#text').removeClass('hidden');
```

Often, you may want to switch a set of styles that belong to a class name back and forth, perhaps to indicate a change between two states or for any other reasons that make sense with your interface. jQuery makes this easy with the `toggleClass()` method.

Method syntax: toggleClass

`toggleClass([names][, switch])`

Adds the specified class name(s) on elements that don't possess it, or removes the name(s) from elements that already possess the class name(s). Note that each element is tested individually, so some elements may have the class name added and others may have it removed.
If the `switch` parameter is provided, the class name(s) is always added to elements without it if `switch` is `true`, and removed from those that have it if it's `false`.
If the method is called without parameters, all the class names of each element in the set will be removed and eventually restored with a new call to this method.
If only the `switch` parameter is provided, all the class names of each element in the set will be kept or removed on that element based on the value of `switch`.
If a function is provided, the returned value is used as the class name or names, and the action taken is based on the `switch` value.

Parameters

names   (String|Function) Specifies the class name, or a space-delimited string of names, to be toggled. If a function, the function is invoked for each element, with that element set as the function context (`this`). The function is passed two values: the element index and the class value of that element. The function's returned value is used as the class name or names to toggle.

switch   (Boolean) A control expression whose value determines if the class will only be added to the elements (`true`) or only removed (`false`).

Returns

The jQuery collection.

As you can see, the `toggleClass()` method gives you many possibilities. Before moving forward with other methods, let's see some examples.

One situation where the `toggleClass()` method is most useful is when you want to switch visual renditions between elements quickly and easily, usually based on some other elements. Imagine that you want to develop a simple share widget where you have a button that, when clicked, shows a box containing the social media buttons to share the link of the page. If the button is clicked again, the box should be hidden.

Using jQuery—and jQuery's `click()` method that we'll cover in chapter 6—you can easily create this widget:

```
$('.share-widget').click(function() {
    $('.socials', this).toggleClass('hidden');
});
```

The complete code for this demo is available in the file chapter-5/share.widget.html. Before you're disappointed, we want to highlight that the demo doesn't provide any actual sharing functionality, only placeholder text. The resulting page in its two states (box hidden and displayed) is shown in figures 5.1a and figure 5.1b, respectively.

Figure 5.1a   The presence of the `hidden` class is toggled whenever the user clicks the button, causing the box to be shown or hidden. In its initial state the box is hidden.

Figure 5.1b   When the Share button is clicked, it toggles the `hidden` class. This figure shows the box when displayed.

Toggling a class based on whether the elements already possess the class or not is a common operation, but so is toggling the class based on some other arbitrary condition. Consider the following code:

```
if (aValue === 10) {
    $('p').addClass('hidden');
} else {
    $('p').removeClass('hidden');
}
```

For this common situation, jQuery provides the `switch` parameter we discussed in the description of the method. You can shorten the previous snippet of code as reported here:

```
$('p').toggleClass('hidden', aValue === 10);
```

In this case, the class `hidden` will be added to all the paragraphs selected if the variable `aValue` is strictly equal to 10; otherwise it'll be removed.

As our last example, imagine that you want to add a class on a per-element basis based on a given condition. You might want to add to all the odd-positioned elements in the set a class called `hidden` while keeping all the classes of the even-positioned elements unchanged. You can achieve this goal by passing a function as the first argument of `toggleClass()`:

```
$('p').toggleClass(function(index) {
    return (index % 2 === 0) ? 'hidden' : '' ;
});
```

Sometimes you need to determine whether an element has a particular class to perform some operations accordingly. With jQuery, you can do that by calling the `hasClass()` method:

```
$('p:first').hasClass('surprise-me');
```

This method will return `true` if any element in the set has the specified class, `false` otherwise. The syntax of this method is as follows.

Method syntax: hasClass

`hasClass(name)`
Determines if any element of the set possesses the passed class name
Parameters
  `names`   (String) The class name to be searched
Returns
  Returns `true` if any element in the set possesses the passed class name, `false` otherwise

---

Recalling the `is()` method from chapter 3, you could achieve the same goal with

```
$('p:first').is('.surprise-me');
```

But arguably, the `hasClass()` method makes for more readable code, and internally `hasClass()` is a lot more efficient.

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

5.1.2   *Getting and setting styles*

Modifying the class of an element allows you to choose which predetermined set of defined style sheet rules should be applied. But sometimes you only want to set the value of one or very few properties that are unknown in advance; thus a class name doesn't exist. Applying styles directly on the elements (via the `style` property available on all DOM elements) will automatically override the style defined in style sheets (some exceptions such as `!important` apply, but we aren't going to cover CSS specificity in

detail here), giving you more fine-grained control over individual elements and their styles.

The jQuery `css()` method allows you to manipulate these styles, working in a similar fashion to `attr()`. You can set an individual CSS style by specifying its name and value, or a series of styles by passing in an object.

Method syntax: css

`css(name, value)`
`css(properties)`

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

Parameters

name   (String) The name of the CSS property to be set. Both the CSS and DOM formatting of multiple-word properties (for example `background-color` versus `backgroundColor`) are supported. Most of the time you'll use the first format version.

value   (String|Number|Function) A string, number, or function containing the property value. If a number is passed, jQuery will convert it to a string and add "px" to the end of that string. If you need a different unit, convert the value to a string and add the appropriate unit before calling the method. If a function is passed as this parameter, it will be invoked for each element of the collection, setting the element as the function context (`this`). The function is passed two values: the element index and the current value. The returned value serves as the new value for the CSS property.

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

Returns

The jQuery collection.

The `value` argument can also be a function in a similar fashion to the `attr()` method. This means that you can, for instance, expand the width of all elements in the set by 20 pixels times the index of the element as follows:

```
$('.expandable').css('width', function(index, currentWidth) {
   return parseInt(currentWidth, 10) + 20 * index;
});
```

In this snippet you need to pass the current value to `parseInt()` because the width of an element is returned in pixels and as a string (for example, `"50px"`). Without a conversion, the sum will act as a concatenation of strings resulting in a value like `"50px20"` (if the value of `index` is 1).

In case you want to expand the width of all the elements by 20 pixels, jQuery offers you a nice shortcut. Instead of writing a function, you can write

```
$('.expandable').css('width', '+=20');
```

A similar shortcut is available if you want to subtract a given amount of pixels:

```
$('.expandable').css('width', '-=20');
```

One interesting side note—and yet another example of how jQuery makes your life easier—is that the normally problematic `opacity` property will work perfectly across browsers (even older ones) by passing in a value between `0.0` and `1.0`; no more messing with the old IE alpha filters!

Now let's see an example of use of the second signature of the `css()` method:

```
$('p').css({
   margin: '1em',
   color: '#FFFFFF',
   opacity: 0.8
});
```

This code will set the values specified to all the elements in the set. But what if you want to create a descending opacity effect with your elements?

As in the shortcut version of the `attr()` method, you can use functions as values to any CSS property in the property's parameter object, and they will be called on each element in the set to determine the value that should be applied. You can achieve this task by using a function as the value of `opacity` instead of a fixed number:

```
$('p').css({
   margin: '1em',
   color: '#1933FF',
   opacity: function (index, currentValue) {
      return 1 - ((index % 10) / 10);
   }
});
```

An example of a page using this code can be found in file chapter-5/descending.opacity .html and as a JS Bin (http://jsbin.com/cuhexe/edit?html,js,output).

Lastly, let's discuss how you can use `css()` with a name or an array of names passed in to retrieve the computed style of the property or properties associated with that name(s) of the first element in the jQuery object. When we say *computed* style, we mean the style after all linked, embedded, and inline CSS has been applied.

Method syntax: css

`css(name)`

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

Parameters

name	(String|Array) Specifies the name of a CSS property or array of CSS properties whose computed value is to be returned

Returns

The computed value as a string or an object of property-value pairs

---

This variant of the `css()` method always returns values as a string, so if you need a number or some other type, you'll need to parse the returned value using `parseInt()` or `parseFloat()` depending on the situation.

To understand how the getter version of css() works when you pass an array of names, let's see an example. The goal is to print on the console the property and its corresponding value of an element having special as its class for the following properties: font-size, color, and text-decoration. To complete the task, you have to write this:

```
var styles = $('.special').css([          Retrieves the object with
   'font-size', 'color', 'text-decoration'   the property-value pairs
]);

for(var property in styles) {        ◁————— Loops over the object
   console.log(property + ': ' + styles[property]);
}
```

This code can be found in the file chapter-5/css.and.array.html and as a JS Bin (http:// jsbin.com/mimixu/edit?html,css,js,console,output). Loading the page (or the JS Bin) in your browser, you can see how the values printed are the result of the combination of all the styles defined in the page. For example, the value printed for font-size isn't "20px" but "24px". This happens because the value defined for the special class (24px) has more specificity than the one defined for the div elements (20px).

The css() method is another example of how jQuery solves a lot of cross-browser incompatibilities for you. To achieve this goal using native methods, you should use getComputedStyle() in all the versions of Chrome, Firefox, Opera, Safari, and Internet Explorer starting from version 9 and use the currentStyle and runtimeStyle properties in Internet Explorer 8 and below.

Before moving on, we want to highlight two important facts. The first fact is that different browsers may return CSS color values that are logically but not textually equal. For example, if you have a declaration like color: black; some browsers may return #000, #000000, or rgb(0, 0, 0). The second is that the retrieval of shorthand CSS properties such as margin or border is not guaranteed by jQuery (although it works in some browsers).

For a small set of CSS values that are commonly accessed, jQuery provides convenience methods that access these values and convert them to the most commonly used types.

GETTING AND SETTING DIMENSIONS

When it comes to CSS styles that you want to set or get on your pages, is there a more common set of properties than the element's width or height? Probably not, so jQuery makes it easy for you to deal with the dimensions of the elements as numeric values rather than strings.

Specifically, you can get (or set) the width and height of an element as a number by using the convenient width() and height() methods. You can set the width or height as follows.

Method syntax: width and height

`width(value)`
`height(value)`
Sets the width or height of all elements in the matched set.

Parameters

`value`   (Number|String|Function) The value to be set. This can be a number of pixels or
                 a string specifying a value in units (such as `px`, `em`, or `%`). If no unit is specified,
                 `px` is the default.
                 If a function is provided, the function is invoked for each element in the set,
                 passing that element as the function context (`this`). The function is passed
                 two values: the element index and the element's current value. The function's
                 returned value is used as the new value.

Returns

The jQuery collection.

Keep in mind that these are shortcuts for the `css()` method, so

```
$('div').width(500);
```

is identical to

```
$('div').css('width', 500);
```

You can also retrieve the width or height as follows.

Method syntax: width and height

`width()`
`height()`
Retrieves the width or height of the first element of the jQuery object

Parameters

`none`

Returns

The computed width or height as a number in pixels; `null` if the jQuery object is empty

---

The getter version of these two methods is a bit different from its `css()` counterpart. `css()` returns a string containing the value and the unit measure (for example, `"40px"`), whereas `width()` and `height()` return a number, which is the value without the unit and converted into a `Number` data type. If your style defines the width or height using units different from pixels (`em`, `%`, and so on), jQuery will still return the value relative to the width or height of the element in pixels.

---

jQuery 3: Bug fixed

jQuery 3 fixes a bug of the `width()`, `height()`, and all the other related methods. These methods will no longer round to the nearest pixel, which made it hard to position elements in some situations. To understand the problem, let's say that you have three elements with a width of 33% inside of a container element that has a width of 100px:

```
<div class="wrapper">
    <div>Hello</div>
    <div>Hi</div>
    <div>Bye</div>
</div>
```

Prior to jQuery 3, if you tried to retrieve the width of one of the three children elements as follows

```
$('.wrapper div:first').width();
```

you'd obtain the value 33 as the result because jQuery rounds the value 33.33333. In jQuery 3 this bug has been fixed, so you'll obtain more accurate results.

The fact that the width and height values are returned from these functions as numbers isn't the only convenience that these methods 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` properties, you were confronted with the sad truth 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()` methods, on the other hand, compute and return the size of the element. Knowing the precise dimensions of an element in simple pages that let their elements lay out wherever they end up isn't usually necessary, but knowing such dimensions in highly interactive scripted pages is crucial to being 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 5.2 shows a sample page that was set up with two primary elements: a `div` serving as a test subject that contains a paragraph of text (with a border and background color for emphasis) and a second `div` in which to display the dimensions. To write the dimensions in the second `div`, we'll use the `html()` method that we'll cover shortly.

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 will cause both dimensions to change.

In our page, we define a function that will use the `width()` and `height()` methods to obtain the dimensions of the test subject `div` (identified as `test-subject`) and display the resulting values in the second `div` (identified as `display`):

```
function displayDimensions() {
   $('#display').html(
      $('#test-subject').width() + 'x' + $('#test-subject').height()
   );
}
```

Lorem ipsum dolor sit amet, consectetuer 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.

700x90

Lorem ipsum dolor sit amet, consectetuer 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.

Figure 5.2 The width and height of the test element aren't fixed and depend on the width of the browser window.

338x180

We call this function as the last statement of our script, resulting in the initial values being displayed, as shown in the upper portion of figure 5.2.

We also add a call to the same function in a resize handler on the window that updates the display whenever the browser window is resized (you'll learn how to do this in chapter 6), as shown in the lower portion of figure 5.2. The full code of this page is shown in the following listing and can be found in the file chapter-5/dimensions.html.

Listing 5.1 Dynamically tracking and displaying the dimensions of an element

```
<!DOCTYPE html>
<html>
    <head>
        <title>Dynamic Dimensions Example - jQuery in Action, 3rd edition</title>
        <link rel="stylesheet" href="../css/main.css"/>
        <style>
            #test-subject
            {
                background-color: #FFFFCC;
                border: 2px ridge maroon;
                padding: 0.5em;
            }
        </style>
    </head>
    <body>
        <div id="test-subject">                          ⟵  Declares test subject
            Lorem ipsum dolor sit amet, consectetuer adipiscing elit.       with dummy text
            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>                    ←——————  Displays dimensions in this area

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
        function displayDimensions() {            ←——┐  Defines a function that
          $('#display').html(                          │  displays width and
            $('#test-subject').width() + 'x' +         │  height of test subject
            $('#test-subject').height()                │
          );
        }
                                                   ┌── Establishes resize handler
        $(window).resize(displayDimensions);    ←──┘   that invokes display function
        displayDimensions();                    ←──┐
      </script>                                     │  Invokes the function to
   </body>                                          │  show the initial values
</html>
```

In addition to the convenient `width()` and `height()` methods, jQuery also provides similar methods for getting more particular dimension values, as described in table 5.1.

Table 5.1 Additional jQuery dimension-related methods

Method	Description
`innerHeight()`	Returns the inner height of the first matched element, which excludes the border but includes the padding. The value returned is of type `Number` unless the jQuery object is empty, in which case `null` is returned. If a number is returned, it refers to the value of the inner height in pixels.
`innerHeight(value)`	Sets the inner height of all the matched elements with the value specified by `value`. The type of value can be `String`, `Number`, or `Function`. The default unit used is `px`. If a function is provided, it's called for every element in the jQuery object. The function is passed two values: the index position of the element within the jQuery object and the current inner height value. Within the function, `this` refers to the current element within the jQuery object. The returned value of the function is set as the new value of the inner height of the current element.
`innerWidth()`	Same as `innerHeight()` except it returns the inner width of the first matched element, which excludes the border but includes the padding.
`innerWidth(value)`	Same as `innerHeight(value)` except the value is used to set the inner width of all the matched elements.
`outerHeight ([includeMargin])`	Same as `innerHeight()` except it returns the outer height of the first matched element, which includes the border and the padding. The `includeMargin` parameter causes the margin to be included if it's `true`.
`outerHeight(value)`	Same as `innerHeight(value)` except the value is used to set the outer height of all the matched elements.

Table 5.1   Additional jQuery dimension-related methods *(continued)*

Method	Description
`outerWidth` `([includeMargin])`	Same as `innerHeight()` except it returns the outer width of the first matched element, which includes the border and the padding. The `includeMargin` parameter causes the margin to be included if it's `true`.
`outerWidth(value)`	Same as `innerHeight(value)` except the value is used to set the outer width of all the matched elements.

You're not finished yet; jQuery also gives you easy support for positions and scrolling values.

POSITIONS AND SCROLLING

jQuery provides two methods for getting the position of an element. Both of these methods return a JavaScript object that contains two properties: `top` and `left`, which indicate the top and left values of the element.

The two methods use different origins from which their relative computed values are measured. One of these methods, `offset()`, returns the position relative to the document.

Method syntax: offset

`offset()`
Returns the current coordinates (in pixels) of the first element in the set, relative to the document.

Parameters

  `none`

Returns

An object with `left` and `top` properties as numbers depicting the position in pixels relative to the document.

This method can also be used to set the current coordinates of one or more elements.

Method syntax: offset

`offset(coordinates)`
Sets the current coordinates (in pixels) of all the elements in the set, relative to the document.

Parameters

  `coordinates`   (Object|Function) An object containing the properties `top` and `left`, which are numbers indicating the new top and left coordinates for the elements in the set. If a function is provided, it's invoked for each element in the set, passing that element as the function context (`this`) and passing two values: the element index and the object containing the current values of `top` and `left`. The function's returned object is used to set the new values.

Returns

The jQuery collection.

The other method, `position()`, returns values relative to an element's closest offset parent. The *offset parent* of an element is the nearest ancestor that has an explicit positioning rule of `relative`, `absolute`, or `fixed` defined. The syntax of `position()` is as follows.

Method syntax: position

`position()`
Returns the position (in pixels) of the first element in the set relative to the element's closest offset parent

Parameters

Returns

An object with `left` and `top` properties as numbers depicting the position in pixels relative to the closest offset parent

---

Both `offset()` and `position()` can only be used for visible elements.

In addition to element positioning, jQuery gives you the ability to get and set the scroll bar position of an element. Table 5.2 describes these methods that work with both visible and hidden elements.

Table 5.2   The jQuery scroll bar control methods

Method	Description
scrollLeft()	Returns the horizontal position of the scroll bar of the first matched element. The value returned is of type `Number` unless the jQuery object is empty, in which case `null` is returned. If a number is returned, it refers to the value of the position in pixels.
scrollLeft(value)	Sets the horizontal position of the scroll bar for all matched elements of `value` pixels. This method returns the jQuery set it has been called upon.
scrollTop()	Same as `scrollLeft()` except it returns the vertical position of the scroll bar of the first matched element.
scrollTop(value)	Same as `scrollLeft(value)` except the value is used to set the vertical position of the scroll bar for all the matched elements.

Now that you've learned how to get and set the horizontal and vertical position of the scroll bar of elements using jQuery, let's see an example.

Imagine that you have an element with an ID of `elem` shown in the middle of your page and that after one second you want to move it to the top-left corner of the document. The point we've described has as its coordinates [0, 0], which means that to move it there you have to set both `left` and `top` to `0`. To achieve the goal just described, all you need is these few lines of code:

```
setTimeout(function() {
   $('#elem').offset({
```

```
        left: 0,
        top: 0
    });
}, 1000);
```

Let's now discuss different ways of modifying an element's contents.

5.2   *Setting element content*

When it comes to modifying the contents of elements, there are a lot of different methods you can employ, depending on the type of the text you want to inject. If you're interested in setting a text whose content should not be parsed as markup, you can use properties like textContent or innerText, depending on the browser.

Once again jQuery saves you from these browser incompatibilities by giving you a number of methods that you can employ.

5.2.1   *Replacing HTML or text content*

First up is the simple html() method, which allows you to retrieve the HTML content of an element when used without parameters or, as you've seen with other jQuery methods, to set the content of all the elements in the set when used with a parameter.

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

Method 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.

And here's how to set the HTML content of all the matched elements.

Method syntax: html

html(content)
Sets the passed HTML fragment as the content of all matched elements.

Parameters

content   (String|Function) The HTML fragment to be set as the element's content. If a
             function, the function is invoked for each element in the set, setting that
             element as the function context (this). The function is passed two values: the
             element index and the existing content. The function's returned value is used
             as the new content.

Returns
The jQuery collection.

Let's say that in your page you have the following element:

```
<div id="message"></div>
```

You're running a function you've developed, and once it ends you need to show a message that contains some content to your user. You can perform this task with a statement like the following:

```
$('#message').html('<p>Your current balance is <b>1000$</b></p>');
```

This statement will cause your previous element to be updated as reported here:

```
<div id="message"><p>Your current balance is <b>1000$</b></p></div>
```

In this case, the tags passed to the method will be processed as HTML. The total balance, for instance, will be shown in bold.

In addition to setting the content as HTML, you can also set or get only the text contents of elements. The `text()` method, when used without parameters, returns a string that's the concatenation of all the texts in the matched set. For example, let's say you have the following HTML fragment:

```
<ul id="the-list">
  <li>One</li><li>Two</li><li>Three</li><li>Four</li>
</ul>
```

The statement

```
var text = $('#the-list').text();
```

results in the variable `text` being set to `OneTwoThreeFour`. Note that if there are white spaces or new lines in between elements (for example between a closing `</li>` and an opening `<li>`) they'll be included in the resulting string.

The syntax of this method is as follows.

Method syntax: text

`text()`
Retrieves the combined text contents of each element in the set of matched elements, including their descendants.

Parameters

Returns

A string of all the text contents.

You can also use the `text()` method to set the text content of the elements in the jQuery object. The syntax for this format is as follows.

Method syntax: text

`text(content)`

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

Parameters

content   (String|Number|BooleanFunction) The text content to be set into the elements in the set. When the value is of type `Number` or `Boolean`, it'll be converted to a `String` representation. Any angle bracket characters are escaped as HTML entities. If a function, it's invoked for each element in the set, setting that element as the function context (`this`). The function is passed two values: the element index and the existing text. The function's returned value is used as the new content.

Returns

The jQuery collection.

Changing the inner HTML or text of elements using these methods will replace the contents that were previously in the elements, so use these methods carefully. jQuery isn't limited to these methods only, so let's take a look at the others.

5.2.2   *Moving elements*

Manipulating the DOM of a page without the necessity of a page reload opens a world of possibilities for making your pages dynamic and interactive. You've already seen a glimpse of how jQuery lets you create DOM elements on the fly. These new elements can be attached to the DOM in a variety of ways, and you can also move (and copy and move) existing elements.

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

Method syntax: append

`append(content[, content, ..., content])`

Appends the passed argument(s) to the content of all matched elements. This method accepts an arbitrary number of arguments with a minimum of one.

Parameters

content   (String|Element|jQuery|Array|Function) A string, a DOM element, an array of DOM elements, or a jQuery object to be appended. If a function is provided, the function is invoked for each element in the set, setting that element as the function context (`this`). The function is passed two values: the element index and the existing contents of that element. The function's returned value is used as the content to append.

Returns

The jQuery collection.

Let's see an example of the use of this method. 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 method identifies existing elements of the DOM as the items to be appended. Consider the following:

```
$('p.append-to').append($('a.append'));
```

This statement moves all `a` elements with the class `append` to the end of the content of all `p` elements having class `append-to`. If there are multiple targets (the elements of the jQuery object the `append()` method is called upon) for the operation, the original element is cloned as many times as is necessary and then appended. In all cases, the original is removed from its initial location.

This operation is semantically a *move* if one target is identified; the original source element is removed from its initial location and appears at the end of the target's list of children.

Consider the following HTML code:

```
<a href="http://www.manning.com" class="append">Text</a>
<p class="append-to"></p>
```

By running the previous statement, you'll end up with the following markup:

```
<p class="append-to">
   <a href="http://www.manning.com" class="append">Text</a>
</p>
```

The operation can also be a copy-and-move operation if multiple targets are identified, creating enough copies of the original so that each target can have one appended to its children.

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

```
$('p.appendToMe').append(someElement);
```

Another example of use for this method is the following:

```
$('#message').append(
   '<p>This</p>',
   [
      '<p>is</p>',
      $('<p>').text('my')
   ],
   $('<p>text</p>')
);
```

In this code you can see how the `append()` method can manage multiple arguments and each argument can be of a different type (a string, an array, and a jQuery object). The result of running it is that you'll have an element having an ID of `message` with four paragraphs that compose the sentence "This is my text."

Although it's a common operation to add elements to the end of an element's content—you might be adding a list item to the end of a list, a row to the end of a table, or

adding a new element to the end of the document body—you might also need to add a new or existing element to the start of the target element's contents.

When such a need arises, the `prepend()` method will do the trick.

Method syntax: prepend

`prepend(content[, content, ..., content])`

Prepends the passed argument(s) to the content of all matched elements. This method accepts an arbitrary number of arguments with a minimum of one.

Parameters

content   Same as the `content` parameter of `append()` except the argument(s) are
            prepended to the content of each element in the set of matched elements.

Returns

The jQuery collection.

Sometimes you might wish to place elements somewhere other than at the beginning or end of an element's content. jQuery allows you to place new or existing elements anywhere in the DOM by identifying a target element that the source elements are to be placed before or after.

Not surprisingly, the methods are named `before()` and `after()`. Their syntax should seem familiar by now.

Method syntax: before

`before(content[, content, ..., content])`

Inserts the passed argument(s) into the DOM as siblings of the target elements, positioned before the targets. The target elements in the set must already be part of the DOM. This method accepts an arbitrary number of arguments with a minimum of one.

Parameters

content   Same as the `content` parameter of `append()` except the argument(s) is
            inserted before each element in the set of matched elements.

Returns

The jQuery collection.

Method syntax: after

`after(content[, content, ..., content])`

Inserts the passed argument(s) into the DOM as siblings of the target elements positioned after the targets. The target elements in the set must already be part of the DOM. This method accepts an arbitrary number of arguments with a minimum of one.

Parameters

content   Same as the `content` parameter of `append()` except the argument(s) is
            inserted after each element in the set of matched elements.

Returns

The jQuery collection.

These operations are crucial to manipulating the DOM effectively in your pages, so we've provided a Move and Copy Lab Page so that you can play around with these operations until you thoroughly understand them. This lab is available at chapter-5/ lab.move.and.copy.html, and its initial display is as shown in figure 5.3.

The left pane of this Lab contains three images that can serve as sources for your move/copy experiments. Select one or more of the images by checking their corresponding check boxes.

Targets for the move/copy operations are in the right pane and are also selected via check boxes. Controls at the bottom of the pane allow you to select one of the four operations to apply: append, prepend, before, or after. (Ignore "clone" for now; we'll attend to that later.)

Figure 5.3 The Move and Copy Lab Page will let you inspect the operation of the DOM manipulation methods.

The Execute button causes any source images you've selected to be applied to a set of the selected targets using the specified operation. When you want to put everything back into place so you can run another experiment, use the Restore button.

Let's run an append experiment. Select the dog image and then select Target 2. Leaving the append operation selected, click Execute. The result of this operation is shown in figure 5.4.

Figure 5.4   Cozmo has been added to the end of Target 2 as a result of the append operation.

Use the Move and Copy Lab Page to try various combinations of sources, targets, and the four operations until you have a good feel for how they operate.

Sometimes it might make the code more readable if you could reverse the order of the elements passed to these operations. If you want to move or copy an element from one place to another, a possible approach would be to wrap the source elements (rather than the target elements) and to specify the targets in the parameters of the method. Well, jQuery lets you do that by providing analogous operations to the four that we just examined, reversing the order in which sources and targets are specified. They are `appendTo()`, `prependTo()`, `insertBefore()`, and `insertAfter()`, and are described in table 5.3.

Table 5.3   Additional methods to move elements in the DOM

Method	Description
`appendTo(target)`	Inserts every element in the set of matched elements to the end of the content of the specified target(s). The argument provided (`target`) can be a string containing a selector, an HTML string, a DOM element, an array of DOM elements, or a jQuery object. The method returns the jQuery object it was called upon.
`prependTo(target)`	Same as `appendTo(target)` except the elements in the set of matched elements are inserted at the beginning of the content of the specified target(s).
`insertBefore(target)`	Same as `appendTo(target)` except the elements in the set of matched elements are inserted before the specified target(s).
`insertAfter(target)`	Same as `appendTo(target)` except the elements in the set of matched elements are inserted after the specified target(s).

Wow, that's a lot of stuff to learn all at once. To help you digest this bunch of new methods, we'll show you a couple of examples.

EXAMPLE #1 - MOVING ELEMENTS

Let's say you have the following HTML code in a page:

```
<div id="box">
   <p id="description">jQuery is so awesome!</p>
   <button id="first-btn">I'm a button</button>
   <p id="adv">jQuery in Action rocks!</p>
   <button id="second-btn">Click me</button>
</div>
```

This code will be rendered by Chrome as shown in figure 5.5a.

Your first goal will be to move all the buttons before the first paragraph, the one having `description` as its ID. To perform this task you can use the `insertBefore()` method:

```
$('button').insertBefore('#description');
```

Because you're a good and observant reader, you're thinking "Hey, I can do that using the `before()` method by just switching the selectors and wrapping the target one with the `$()` function!" Congratulations, you're right! The previous statement can be equivalently turned into this:

```
$('#description').before($('button'));
```

Once executed, regardless of which of the two previous statements you run, the page will be updated as shown in figure 5.5b.

You can see this code in action accessing the related JS Bin (http://jsbin.com/ARedIWU/edit?html,js,output) or the file
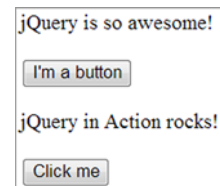
Figure 5.5a   The HTML code rendered by Chrome

Figure 5.5b   The HTML snippet rendered by Chrome after the execution of the statement

chapter-5/moving.buttons.html (as soon as the page is loaded, the buttons will be moved). Now let's see a slightly more complex example.

EXAMPLE #2 - COPYING AND MERGING CONTENT

Imagine you have the same markup as the previous example and you want to create a new paragraph having the content equal to the union of the two paragraphs inside the div and put it right after the div itself. The content of this newly created paragraph will be "jQuery is so awesome! jQuery in Action rocks!" You can do that by combining two of the methods we explained in this chapter: text() and after(). The code that implements this request is listed here:

```
var $newParagraph = $('<p></p>').text(
    $('#description').text() + ' ' + $('#adv').text()
);
$('#box').after($newParagraph);
```

These two examples should prove how the more methods you discover, the more power you have in your hands. And we're not finished yet! There's one more thing we need to address before we move on. Sometimes, rather than inserting elements *into* other elements, you want to do the opposite. Let's see what jQuery offers for that.

5.2.3 *Wrapping and unwrapping elements*

Another type of DOM manipulation that you'll often need to perform is to wrap an element (or a set of elements) in some markup. You might want to wrap all links of a certain class inside a <div>. You can accomplish such DOM modifications by using jQuery's wrap() method.

Method syntax: wrap

wrap(wrapper)
Wraps the elements in the jQuery object with the argument provided.

Parameters

wrapper   (String|Element|jQuery|Function) A string containing the opening tag (and optionally the closing tag) of the element with which to wrap each element of the matched set. The argument can also be an element, a jQuery object, or a selector specifying the elements to be cloned and serve as the wrapper. If a selector matching more than one element or a jQuery object is passed, only the first element is used as the wrapper. If a function is provided, the function is invoked for each element in the set, passing that element as the function context (this) and passing one parameter to the function that's the element index. The function's returned value, which can be an HTML fragment or a jQuery object, is used to wrap the current element of the set.

Returns

The jQuery collection.

To understand what this method does, let's say that you have the following markup:

```
<a class="surprise">A text</a>
<a>Hi there!</a>
```

```
<a class="surprise">Another text</a>
```

To wrap each link with class `surprise` with a `<div>` having class `hello`, you could write

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

The result of running such a statement would be the following:

```
<div class="hello"><a class="surprise">A text</a></div>
<a>Hi there!</a>
<div class="hello"><a class="surprise">Another text</a></div>
```

If you wanted to wrap the link in a clone of a hypothetical first `div` element on the page, you could write

```
$('a.surprise').wrap($('div:first'));
```

or alternatively

```
$('a.surprise').wrap('div:first');
```

Remember that in this case the content of the `div` will also be cloned and used to surround the `a` elements.

When multiple elements are collected in a jQuery object, the `wrap()` method operates on each one individually. If you'd rather wrap all the elements in the jQuery object as a unit, you could use the `wrapAll()` method instead.

Method syntax: wrapAll

`wrapAll(wrapper)`
Wraps the elements of the matched set, as a unit, with the argument provided

Parameters

  `wrapper`   Same as the `wrapper` parameter of `wrap()`

Returns
The jQuery collection

---

jQuery 3: Bug fixed

jQuery 3 fixes a bug of the `wrapAll()` method that occurred when passing a function to it. Prior to jQuery 3, when passing a function to `wrapAll()`, it wrapped the elements of the matched set individually instead of wrapping them as a unit. Its behavior was the same as passing a function to `wrap()`.

In addition to fixing this issue, because in jQuery 3 the function is called only once, it isn't passed the index of the element within the jQuery object. Finally, the function context (`this`) now refers to the first element in the matched set.

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

Method syntax: wrapInner

`wrapInner(wrapper)`

Wraps the contents, including text nodes, of the elements in the matched set with the argument provided

Parameters

wrapper   Same as the `wrapper` parameter of `wrap()`

Returns

The jQuery collection

The converse operation of `wrap()`, which is removing the parent of a child element, is possible with the `unwrap()` method.

Method syntax: unwrap

`unwrap()`

Removes the parent element of the elements in the set. The child element, along with any siblings, replaces the parent element in the DOM.

Parameters

Returns

The jQuery collection.

---

jQuery 3: Parameter added

jQuery 3 adds an optional `selector` parameter to `unwrap()`. You can pass a string containing a selector expression to match the parent element against. In case there is a match, the child elements are unwrapped; otherwise, the operation isn't performed.

Before moving on, let's see an example of the use of these methods.

HOW TO WRAP LABEL-INPUT AND LABEL-TEXTAREA PAIRS OF A FORM

Imagine that you have the following contact form:

```
<form id="contact" method="post">
    <label for="name">Name:</label>
    <input name="name" id="name" />
    <label for="email">Email:</label>
    <input name="email" id="email" />
    <label for="subject">Subject:</label>
    <input name="subject" id="subject" />
    <label for="message">Message:</label>
```

```
    <textarea name="message" id="message"></textarea>
    <input type="submit" value="Submit" />
</form>
```

You want to wrap every `label`-`input` or `label`-`textarea` pair in a `<div>` having the class `field`. You define this class as follows:

```
.field
{
    border: 1px solid black;
    margin: 5px 0;
}
```

Wrapping the pairs means that you don't want to wrap each element inside the form individually. To achieve this goal, you'll need to use some of the knowledge you've acquired throughout the first chapters of the book. But don't worry! This can be a good test to see if you've digested the concepts explained so far or if you need a quick refresher. One of the possible solutions is the following:

Selects all the form's `<input>`s and `<textarea>`s and processes them individually

```
$('input, textarea', '#contact').each(function(index, element) {
    var $this = $(this);
    $this
        .add($this.prev('label'))
        .wrapAll('<div class="field"></div>');
});
```

Caches the set containing the current element only

Wraps the pair inside the `<div>`

Adds to the set the preceding sibling element only if it's a `<label>`

To perform the exercise, you need to select all the `<input>`s and the `<textarea>`s inside the `<form>` (for the sake of brevity we're ignoring other tags such as `<select>`) and find a way to tie them with the related `<label>`. Then you have to process each pair individually. To do that, after selecting the elements you need, use the `each()` method we covered in chapter 3. Inside the anonymous function passed to it, wrap the current element using `$()` and then store it in a variable because you're going to use it twice. Then add to the element in the set the preceding sibling element only if it's a `label` element. At this point, you have a set with the two elements you need, so you wrap this set with the `<div>` as required. Yeah, mission accomplished!

Figure 5.6a shows the `<form>` before running the code and figure 5.6b shows it after.

Figure 5.6a   The form before executing the code

Figure 5.6b   The form after executing the code. Note how each `label-input` or `label-textarea` pair is surrounded by a black border, proving that they've been wrapped by the `<div>` as required.

In case you want to play further with this example, it's available as a JS Bin (http://jsbin.com/IrUhEfAg/edit?html,css,js,output) and in the file chapter-5/wrapping .form.elements.html.

So far you've learned how to perform many operations. It's now time to learn how to remove elements from the DOM.

5.2.4   Removing elements

Sometimes you might need to remove elements that are no longer needed. If you want to remove a set of elements and all their content, you can use the `remove()` method, whose syntax is as follows.

Method syntax: remove

`remove([selector])`

Removes all elements and their content in the set from the page, including event listeners attached and any data stored

Parameters

`selector`   (String) An optional selector that further filters which elements of the set are to be removed

Returns

The jQuery collection

Note that as with many other jQuery methods, the set is returned as the result of this method. The elements that were removed from the DOM are still referenced by this set (and hence are not yet eligible for garbage collection) and can be further operated upon using other jQuery methods including `appendTo()`, `prependTo()`, `insert-Before()`, `insertAfter()`. But any data stored or event listener added to the removed element is lost.

If you want to remove the elements from the DOM but retain any bound events and data (that you might have added using the `data()` method), you can use `detach()`.

Method syntax: detach

`detach([selector])`

Removes all elements and their content in the set from the page DOM, retaining any bound events and jQuery data

---

Method syntax: detach *(continued)*

Parameters

selector	(Selector) An optional selector string that further filters which elements of the set are to be detached

Returns

The jQuery collection.

The detach() method is the preferred means of removing an element that you'll want to put back into the DOM at a later time with its events and data intact. A typical situation pulls the element from the DOM, applies several changes to it, and then pushes it again in the DOM. Doing so will improve the performance of your code because modifying a detached element is faster than applying all the changes to one or more elements that are currently in the DOM.

To completely empty DOM elements of their contents but retain the elements themselves, you can use the empty() method. Its syntax is as follows.

Method syntax: empty

empty()
Removes the content of all DOM elements in the matched set.

Parameters

Returns

The jQuery collection.

This method is useful when you deal with injecting external content fetched using Ajax. Let's say that you fetched some new content and now you need to add it inside your page in a `<div>` having content as its ID. You can perform this task with the following code:

```
var newContent = '<p>Wow, this new content is awesome!</p>';    ⟵  Content ideally
$('#content')                                                       fetched from an
    .empty()                    ⟵──────  Removes its content        external resource
    .html(newContent);          ⟵┐
                                  │  Injects the fetched content
                                  │  inside it as HTML
```

Retrieves the element ──▷

Remember to pay attention when you inject external content inside your page using the html() method because you may be exposed to attacks such as XSS (cross-site scripting) and CSRF (cross-site request forgery).

Removing elements is nice, but sometimes you need to clone elements.

5.2.5   *Cloning elements*

One more way that you 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()` method.

Method syntax: clone

`clone([copyHandlersAndData[, copyChildrenHandlersAndData]])`

Creates a deep copy of the elements in the jQuery collection and returns a new jQuery collection that contains them. The elements and any children are copied. Event handlers and data are optionally copied depending on the setting of the `copyHandlersAndData` parameter.

Parameters

`copyHandlersAndData`	(Boolean) If `true`, event handlers and data are copied. If `false` or omitted, handlers and data aren't copied.
`copyChildrenHandlersAndData`	(Boolean) If `true`, copies the event handlers and the data for all the children of the cloned elements. If omitted, if the first parameter is provided, the same value is used; otherwise `false` is assumed. If `false`, the event handlers and the data aren't copied.

Returns

The newly created jQuery collection.

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

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

makes copies of all `img` elements and appends them to all `fieldset` elements with the class name `photo`.

A slightly more interesting example is as follows:

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

This method's chain performs a similar operation, but the targets of the cloning operation—all `ul` elements—are copied *including* their data and event handlers. In addition, because the `clone()` method clones children, too, and it's likely that any `ul` element will have a number of `li` children, you're sure that no information is lost. Because you omit the second argument but specify the first, the data and event handlers of all the children are also copied.

Before moving to another topic, let's discuss one last example. Imagine that you have a set of links with an image inside them. Both links and images have some data and events handlers attached. You want to copy all of them and place the copies after all the elements inside the first `div` of the page. In addition, you want to retain only the data and the handlers of the links, not those of the images inside. This task is performed with the following statement:

```
$('a').clone(true, false).appendTo('div:first');
```

This statement shows you the use of the optional parameters discussed in the description of the method.

In order to see the clone operation in action, return to the Move and Copy Lab Page. Just above the Execute button is a pair of radio buttons that allows you to specify a cloning operation as part of the main DOM manipulation operation. When the Yes radio button is selected, the sources are cloned before the `append()`, `prepend()`, `before()`, and `after()` methods are executed.

Repeat some of the experiments you conducted earlier with cloning enabled, and note how the original sources are unaffected by the operations.

You can insert, remove, and copy. Using these operations in combination, it would be easy to perform higher-level operations such as *replace*. But guess what? You don't need to!

5.2.6 *Replacing elements*

For those times when you want to replace existing elements with new ones, or to move an existing element to replace another, jQuery provides the `replaceWith()` method.

Method syntax: replaceWith

`replaceWith(content)`

Replaces each matched element with the specific content.

Parameters

content   (String|Element|Array|jQuery|Function) A string containing an HTML fragment to become the replaced content, or a DOM element, an array of DOM elements, or a jQuery object containing the elements to be moved to replace the existing elements. If a function, the function is invoked for each element in the set, setting that element as the function context (`this`) and passing no parameters. The function's returned value is used as the new content.

Returns

A jQuery collection containing the replaced elements.

To understand what this method does, let's discuss an example. Imagine that you have the following markup:

```
<img src="image1.jpg" alt="A ball" />
<img src="image2.jpg" alt="A blue bird" />
<img src="image3.jpg" />
```

You want to replace all the images that have an `alt` attribute, one at a time, with `span` elements. The latter will have as their text the value of the `alt` attribute of the image being replaced. Employing `each()` and `replaceWith()`, you could do it like this:

```
$('img[alt]').each(function(){
  $(this).replaceWith('<span>' + $(this).attr('alt') + '</span>');
});
```

The `each()` method lets you iterate over each matched element, and `replaceWith()` is used to replace the images with generated span elements. The resulting markup is shown here:

```
<span>A ball</span>
<span>A blue bird</span>
<img src="image3.jpg" />
```

This example shows once again how easy it is to work with jQuery to manipulate the DOM.

The `replaceWith()` method returns a jQuery set containing the elements that were removed from the DOM, in case you want to do something other than just discard them. As an exercise, consider how you'd augment the example code to reattach these elements elsewhere in the DOM after their removal.

When an existing element is passed as the argument to `replaceWith()`, it's detached from its original location in the DOM and reattached to replace the target elements. If there are multiple such targets, the original element is cloned as many times as needed.

At times, it may be convenient to reverse the order of the elements as specified by `replaceWith()` so that the replacing element can be specified using the matching selector. You've already seen such complementary methods, such as `append()` and `appendTo()`, that let you specify the elements in the order that makes the most sense for your code.

Similarly, the `replaceAll()` method mirrors `replaceWith()`, allowing you to perform a similar operation. But in this case the elements to be replaced are defined by the selector passed as arguments and thus are not those the method is called upon.

<div style="text-align:center">**Method syntax: replaceAll**</div>

replaceAll(target)
Replaces each element matched by the passed `target` with the set of matched elements to which this method is applied

Parameters

`target`	(String|Element|Array|jQuery) A selector string expression, a DOM element, an array of DOM elements, or a jQuery collection that specifies the elements to be replaced

Returns
A jQuery collection containing the inserted elements

Like `replaceWith()`, `replaceAll()` returns a jQuery collection. But it doesn't contain the replaced elements but rather the *replacing* elements. The replaced elements are lost and can't be further operated upon. Keep this in mind when deciding which replace method to employ.

Based on the description of the `replaceAll()` method, you can achieve the same goal of the previous example by writing the following:

```
$('img[alt]').each(function(){
  $('<span>' + $(this).attr('alt') + '</span>').replaceAll(this);
});
```

Note how you invert the argument passed to `$()` and `replaceAll()`.

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

5.3   *Dealing with form element values*

Because form elements have special properties, jQuery's core contains a number of convenience functions for activities such as these:

- Getting and setting their values
- Serializing them
- Selecting elements based on form-specific properties

What's a form element?

When we use the term *form element*, we're referring to the elements that can appear within a form, possess `name` and `value` attributes, and whose values are sent to the server as HTTP request parameters when the form is submitted.

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

Method syntax: val

`val()`

Returns the current value of the first element in the jQuery collection. If the first element is a `<select>` and no option is selected, the method returns `null`. If the element is a multiselect element (a `<select>` having the `multiple` attribute specified) and at least one option is selected, the returned value is an array of all selected options.

Parameters

Returns

The fetched value or values.

This method, although quite useful, has a number of limitations of which you need to be wary. If the first element in the set isn't a form element, an empty string is returned, which some of you may find misleading. This method 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're 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 `radio-group` and the following expression:

```
$('input[type="radio"][name="radio-group"]:checked').val();
```

This expression returns the value of the single checked radio button (or `undefined` if none are checked). That's a lot easier than looping through the buttons looking for the checked element, isn't it?

Because `val()` considers only the first element in a set, it's not as useful for check box groups where more than one control might be checked. But jQuery rarely leaves you without recourse. Consider the following:

```
var checkboxValues =
   $('input[type="checkbox"][name="checkboxgroup"]:checked').map(function() {
      return $(this).val();
   })
   .toArray();
```

Although the `val()` method is great for obtaining the value of any single form control element, if you want to obtain the complete set of values that would be submitted through a form submission, you'll be much better off using the `serialize()` or `serializeArray()` methods (which you'll see in chapter 10).

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

Method syntax: val

`val(value)`

Sets the passed value as the `value` of all matched elements. If an array of values is provided, it causes any check boxes, radio buttons, or options of `select` elements in the set to become checked or selected if their `value` properties match any of the values passed in the `values` array.

Parameters

value   (String|Number|Array|Function) Specifies the value that is to be set as the `value` property of each element in the set. An array of values will be used to determine which elements are to be checked or selected. If a function, the function is invoked for each element in the set, with that element passed as the function context (`this`), and two values: the element index and the current value of the element. The value returned from the function is taken as the new value to be set.

Returns

The jQuery collection.

As the description of the method specifies, the `val()` method can be used to cause check box or radio elements to become checked or to select options within a `<select>` element. Consider the following statement:

```
$('input[type="checkbox"], select').val(['one', 'two', 'three']);
```

It'll search all the `checkboxes` and `selects` on the page for values that match any of the input strings: "one", "two", or "three". Any element found that matches will become checked or selected. In case of a `select` element without the `multiple` attribute defined, only the first value to match is selected. In our previous code only an option having a value of `one` is selected because in the array passed to `val()` the string "one" comes before the strings "two" and "three". This makes `val()` useful for much more than the text-based form elements.

5.4 Summary

With the techniques learned in this chapter, you're able to copy elements, move them, replace them, or even remove them. You can also append, prepend, or wrap any element or set of elements on the page. In addition, we discussed how to manage the values of form elements, all leading to powerful yet succinct logic.

With that behind you, you're ready to look at some more advanced concepts, starting with the typically messy job of handling events in your pages.

Events are where it happens!

Like many other GUI management systems, the interfaces presented by HTML web pages are *asynchronous* and *event-driven*, even if the protocol used to deliver them to the browser, HTTP, is wholly synchronous in nature. Whether a GUI is implemented as a desktop program using Java Swing, X11, or the .NET Framework, or as a page in a web application using HTML and JavaScript, the program steps are pretty much the same:

1   Set up the user interface.
2   Wait for something interesting to happen.
3   React accordingly.
4   Go to step 2.

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 style (CSS) that you send to it. The script you include in the page defines the behavior of the interface, although it can change the user interface (UI) as well.

This script takes the form of *event listeners*, also referred as *event handlers* (although there's a subtle technical difference), 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 often the result of some user activity (such as moving the mouse, clicking a button of the mouse, entering text via the keyboard, or even touch gestures). Without the ability to react to these events, the World Wide Web's greatest use might be limited to showing pictures.

Although HTML itself does define a small set of built-in semantic actions that require no scripting on your 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 you wish your pages to exhibit require you to handle the various events that occur as your users interact with those pages.

In this chapter, we'll examine the various ways that browsers expose these events and how they allow you to establish handlers to control what happens when these events occur. In addition, we'll look at the challenges that you face due to the multitude of differences between the browser event models. Then you'll see how jQuery cuts through the browser-induced fog to relieve you of these burdens.

Let's start off by examining how browsers expose their event models.

JavaScript you need to know!

One of the benefits that jQuery brings to web pages is the ability to implement a great deal of scripting-enabled behavior without having to write a lot of script yourself. jQuery handles the nuts-and-bolts details so that you can concentrate on the job of making your applications do what they need to do!

Up to this point, the ride has been pretty painless. You needed only 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 important 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 may have been able to get pretty far without a firm grasp of them—the very 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.

(continued)

If you're already comfortable with the workings of the JavaScript `Object` and `Function` 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 the appendix to help you get up to speed on these necessary concepts and then come back.

6.1   *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. 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 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 Internet Explorer 9 and above, Firefox, Chrome, Safari, and Opera. Internet Explorer 8 and below have their own way and support a subset of the functionality in the DOM Level 2 Event Model, albeit using a proprietary interface.

Before we show you how jQuery makes that irritating fact a non-issue, you need to spend some time getting to know how the various event models operate.

6.1.1   *The DOM Level 0 Event Model*

The DOM Level 0 Event Model is the event model that most amateurs and beginning web developers employ on their pages because it's somewhat browser-independent and 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 also allow you to specify the body of event handler functions as attribute values embedded within the HTML markup of the DOM elements, providing a shortcut for creating event handlers. An example of defining handlers in both ways is shown in the following listing, which can be found in the downloadable code for this book in the file chapter-6/dom.0.events.html.

Listing 6.1   Declaring DOM Level 0 event handlers

```html
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Level 0 Events - jQuery in Action, 3rd edition</title>
    <link rel="stylesheet" href="../css/main.css"/>
    <style>
      img
      {
        display: block;
        margin: auto;
      }
    </style>
  </head>
  <body>
    <img id="example" src="images/example.jpg" alt="A bolt of lightning"
      onclick="console.log('At ' + formatDate(new Date()) + ' BOOM!');" />
    <script>
      function formatDate(date) {
        return (date.getHours() < 10 ? '0' : '') + date.getHours() +
          ':' + (date.getMinutes() < 10 ? '0' : '') + date.getMinutes() +
          ':' + (date.getSeconds() < 10 ? '0' : '') + date.getSeconds() +
          '.' + (date.getMilliseconds() < 10 ?
              '00' : (date.getMilliseconds() < 100 ? '0' : '')) +
          date.getMilliseconds();
      }
      document.getElementById('example').onmouseover = function(event) {
        console.log('At ' + formatDate(new Date()) + ' Crackle!');
      };
    </script>
  </body>
</html>
```

❶ Instruments img element

❷ Defines the mouseover handler

❸ Emits text to the console

In this example, you employ both styles of event handler declaration: declaring in a markup attribute ❶ and declaring under script control ❷. In the body of the page, you define an `img` element, having `example` as its ID, upon which you're defining the event handler for the `click` event using the `onclick` attribute ❶.

Inside the `<script>` tag, you declare a function called `formatDate()` used to format and return a string representing the time of the given `Date` object. Then, using the JavaScript's `getElementById()` method, you retrieve a reference to the image, setting its `onmouseover` property to an inline function ❷. 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, which we'll discuss in a few moments. Within this function, you print on the console a formatted date using once again the previously declared `formatDate()` function ❸.

> **NOTE** We've thrown the concept of unobtrusive JavaScript out the window for this example. Long before you reach the end of this chapter, you'll see why you won't need to embed event behavior in the DOM markup anymore!
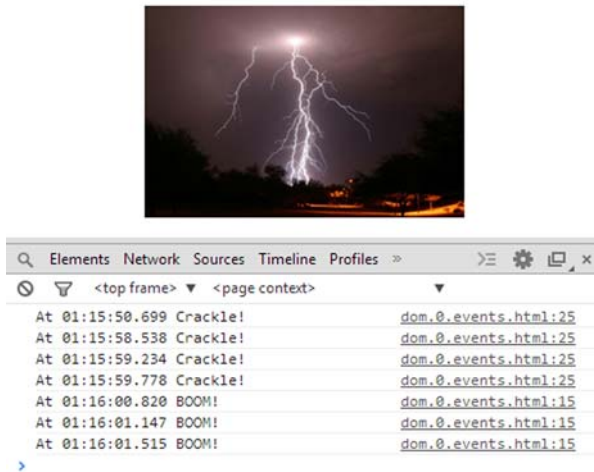
Figure 6.1   Waving the mouse over the image and clicking it results in the event handlers firing and emitting their messages to the console.

Loading this page (found in the file chapter-6/dom.0.events.html) into Chrome, waving the mouse pointer over the image a few times, and then clicking the image results in a display similar to that shown in figure 6.1.

You declared the `click` event handler in the `img` element markup using the following attribute:

```
onclick="console.log('At ' + formatDate(new Date()) + ' BOOM!');"
```

This might lead you to believe that the statement becomes the `click` event handler for the element, but that's not the case. When handlers are declared via HTML markup attributes, an anonymous function is automatically created using the value of the attribute as the function **body**. Assuming that `imageElement` is a reference to the image element, the construct created as a result of the attribute declaration is equivalent to the following:

```
imageElement.onclick = function(event) {
  console.log('At ' + formatDate(new Date()) + ' BOOM!');
};
```

Note how the value of the attribute is used as the body of the generated function and that the function is created so that the `event` parameter is available within the generated function. Finally, within this function you can refer to the element itself using the `this` keyword. We haven't used it in our simple example, but it's a detail we wanted to highlight.

Once again, we want to remind you that using the attribute mechanism of declaring DOM Level 0 event handlers violates the precepts of unobtrusive JavaScript. You'll shortly see that JavaScript and jQuery, which also deal with browser incompatibilities, provide a much better way to declare event handlers than either of the means described. But first, let's examine what that `event` parameter is all about.

THE EVENT INSTANCE

When an event handler is fired, an instance of an object named `Event` is passed to the handler as its first parameter in all standards-compliant browsers. Once again, the latter are Internet Explorer 9 and above, Firefox, Chrome, Safari, and Opera. Internet Explorer 8 and below do things in their own proprietary way by tacking the `Event` instance onto a global property (in other words, a property of `window`) named `event`. It's worth noting that in order to preserve backward compatibility with older scripts, new versions of Internet Explorer still have a reference to the event in the `window` object. Not only that, but Microsoft has also kept its proprietary properties of the object, merging them with the standard ones. Notably, although Chrome was released long after the publication of the DOM Level 2, it adds a reference to the event in the `window` object and supports Internet Explorer's properties as well.

In order to deal with this discrepancy, you'll often see the following used as the first statement in a non-jQuery event handler:

```
if (!event) {
    event = window.event;
}
```

Those of you who are more experienced with JavaScript will know that you can reduce the previous code to a one-line statement:

```
event = event || window.event;
```

This levels the playing field by using feature detection (a concept we'll explore in chapter 9) to check if the `event` parameter is passed and assigning the value of the window's `event` property to it otherwise. After this statement, you can reference the `event` parameter 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. If you're dealing with older versions of Internet Explorer, not only do they use a proprietary means to get the `Event` instance to the handler, but they also use a proprietary definition of the `Event` object in place of the W3C-defined standard—you're not out of the object-detection woods yet. With all these exceptions in place, you can understand how happy developers have been that new versions of Internet Explorer embraced W3C standards. Microsoft is not as evil as many think.

To give you an idea of these inconsistencies among browsers, let's look at an example. To get a reference of the target element—the element on which the event was triggered—you must access the `target` property in standards-compliant browsers but the `srcElement` property in older versions of Internet Explorer. You deal with this inconsistency by employing feature detection with a statement such as the following:

```
var target = event.target || event.srcElement;
```

This statement tests whether `event.target` is defined and, if so, assigns its value to the local `target` variable; otherwise, it assigns the value of `event.srcElement`. You'll be required to take similar steps for other `Event` properties of interest.

Up until this point, we've acted as if event handlers are pertinent only to the elements that serve as the trigger to an event, such as the image element of listing 6.1, 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 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 a handler for the event type, and if so, it's also invoked. At this point 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 you view the DOM tree with its root at the top), this process is termed *event bubbling*.

Let's modify the example from listing 6.1 so that you can see this process in action. Consider the code in the next listing, which can be found in the file chapter-6/dom.0.propagation.html.

Listing 6.2   Events propagate from the point of origin to the top of the DOM

```
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Level 0 Bubbling - jQuery in Action, 3rd edition</title>
    <link rel="stylesheet" href="../css/main.css"/>
    <style>
      img
      {
        display: block;
        margin: auto;
      }
    </style>
  </head>
  <body>
    <div id="greatgrandpa">
      <div id="grandpa">
        <div id="pops">
          <img id="example" src="images/example.jpg"
               alt=" A bolt of lightning" />
        </div>
      </div>
    </div>
    <script>
      function formatDate(date) {
        return (date.getHours() < 10 ? '0' : '') + date.getHours() +
            ':' + (date.getMinutes() < 10 ? '0' : '') + date.getMinutes() +
            ':' + (date.getSeconds() < 10 ? '0' : '') + date.getSeconds() +
```

```
                                   '.' + (date.getMilliseconds() < 10 ?
                                       '00' : (date.getMilliseconds() < 100 ? '0' : '')) +
                                   date.getMilliseconds();
                           }

           Loops over      var elements = document.getElementsByTagName('*');
          the selected     for(var i = 0; i < elements.length; i++) {
             elements         (function(current) {
                               current.onclick = function(event) {
                                 event = event || window.event;
                                 var target = event.target || event.srcElement;
                                 console.log(
                                     'At ' + formatDate(new Date()) +
                                     ' For ' + current.tagName + '#'+ current.id +
                                     ' target is ' + target.tagName + '#' + target.id
                                 );
                               };
                             })(elements[i]);
                           }
                       </script>
                     </body>
                   </html>
```

2 Loops over the selected elements
1 Selects every element on the page
Defines the onclick handler for **3** every element

You do a lot of interesting things in the changes to this example. First you remove the previous handling of the `mouseover` event so that you can concentrate on the `click` event. You also wrap the image element that will serve as the target for your event experiment into three nested `div` elements, merely to place the image element artificially deeper within the DOM hierarchy. You also give almost every element in the page a specific ID.

Inside the `<script>` tag, you use JavaScript's `getElementsByTagName()` method and the Universal selector to retrieve all the elements on the page **1**. Then, using a `for` loop, you iterate over each of them **2** and attach a handler to react to a `click` event **3**. For each matched element, you create a *closure* (please read the section on closures in the appendix if closures are a subject that gives you heartburn) to record its instance in the local variable `current`. Inside the handler you can't refer to `elements[i]` because at the time the handler will be executed, the value of the index `i` is outside the range of the array(-like) object.

> **NOTE** The method `getElementsByTagName()` doesn't return an actual array but an `HTMLCollection`. Objects like this are called array-like because they allow you to reference their elements using an index, and they also have a `length` property but don't implement array methods like `push()` and `join()`.

The handler employs the browser-dependent tricks that we discussed in the previous section to locate the `Event` instance and identify the event target, and then it prints a message on the console. This message is another interesting part of this example. It displays the tag name and the ID of the current element (if any), putting *closures* to work, 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.
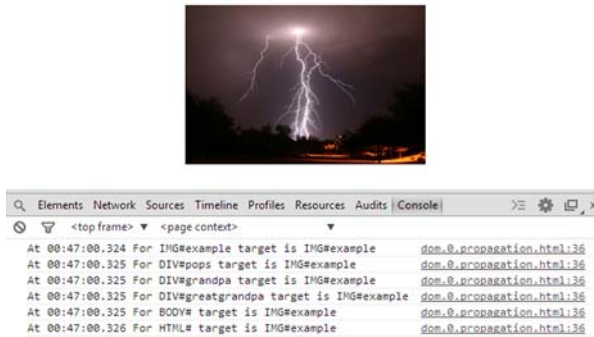
Figure 6.2   The console messages clearly show the propagation of the event as it bubbles up the DOM tree from the target element to the tree root.

Loading the page (located in the file chapter-6/dom.0.propagation.html) and clicking the image results in the display shown in figure 6.2.

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 root `html` element.

This is a powerful ability because it allows you to establish handlers on elements at any level to manage events occurring on their descendants. Consider as an example a handler on a `form` element that reacts to any change event on its child elements to effect dynamic changes to the display based on the elements' new values.

But what if you don't want the event to propagate? Can you stop it?

AFFECTING EVENT PROPAGATION AND SEMANTIC ACTIONS

There may be occasions when you want to prevent an event from bubbling any further up the DOM tree. This might be because you're fastidious and you know that you've already accomplished any processing necessary to handle the event, or you may want to forestall unwanted handling that might occur higher up in the chain.

Regardless of the reason, you can prevent an event from propagating any higher via mechanisms provided on the `Event` instance. In modern browsers, you can call the `stopPropagation()` method of the `Event` instance to halt the propagation of the event farther up the ancestor hierarchy. In older versions of Internet Explorer, you have to 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. For example, 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 you wish to cancel these semantic actions—usually referred as the *default actions*—of the event, in modern browsers you can call the `preventDefault()` method of the `Event` instance. In older versions of Internet Explorer, the method doesn't exist, so you have to set a property called `returnValue` to `false`. Alternatively, within the event handler you can achieve the same result by returning `false`. Sometimes you may also need to stop the propagation and the default action.

A frequent use for such an action is in the realm of form validation. In the handler for the form's `submit` event, you can make validation checks on the form's controls and return `false` if any problems with the data entry are detected.

You may also have seen the following on `form` elements:

```
<form name="myForm" onsubmit="return false;" ...
```

This effectively prevents the form from being submitted in any circumstances except under script control (via `form.submit()`, which doesn't trigger a `submit` event).

Under the DOM Level 0 Event Model, almost every step you 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 you consider the more advanced event model.

6.1.2 *The DOM Level 2 Event Model*

One severe shortcoming of the DOM Level 0 Event Model is that only one event handler per element can be registered for any specific event type at a time. This happens because a property is used to store a reference to a function that's to serve as an event handler. If you have two things that you want to do when an element is clicked, the following statements won't let that happen:

```
someElement.onclick = function doFirstThing() {};
someElement.onclick = function doSecondThing() {};
```

Because the second assignment replaces the previous value of the `onclick` property, only `doSecondThing()` is invoked when the event is triggered. Sure, you could wrap the two functions in another function that calls both, but as pages get more complicated it becomes increasingly difficult to keep track of such things. Moreover, if you use multiple reusable components or libraries in a page, they may have no idea of the event-handling needs of the other components. The code that somebody else has written might be trying to set the `someElement.onclick` property as well, and either this click handler or yours is going to be overwritten. You could employ other solutions, but all of these add complexity to pages that are likely to already be complex enough.

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 ones, are established on DOM elements under this more advanced model.

ESTABLISHING EVENT HANDLERS

Rather than assigning a function reference to an element property, DOM Level 2 event handlers are established via an element *method.* Each DOM element defines a method named `addEventListener()` that's used to attach event 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. These string values are, generally, the same event names you 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 an inline, usually anonymous, 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, you'll leave it set to `false`.

Now that we've discussed the parameters of `addEventListener()`, guess what? Older versions of Internet Explorer have their own way of attaching handlers! We'll delve into the details of the Internet Explorer way in section 6.1.3, "The Internet Explorer Event Model."

To see the method in action, you'll once again change the example from listing 6.1 to use the more advanced event model. You'll concentrate only on the `click` event type. This time you'll establish three `click` event handlers on the `img` element. The new example code can be found in the file chapter-6/dom.2.events.html and is shown in the following listing.

Listing 6.3   Establishing event handlers with the DOM Level 2 Event Model

```
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Level 2 Events - jQuery in Action, 3rd edition</title>
    <link rel="stylesheet" href="../css/main.css"/>
    <style>
      img
      {
        display: block;
        margin: auto;
      }
    </style>
  </head>
  <body>
    <img id="example" src="images/example.jpg"
        alt=" A bolt of lightning " />
    <script>
      function formatDate(date) {
        return (date.getHours() < 10 ? '0' : '') + date.getHours() +
            ':' + (date.getMinutes() < 10 ? '0' : '') + date.getMinutes() +
            ':' + (date.getSeconds() < 10 ? '0' : '') + date.getSeconds() +
            '.' + (date.getMilliseconds() < 10 ?
                '00' : (date.getMilliseconds() < 100 ? '0' : '')) +
            date.getMilliseconds();
      }

      var element = document.getElementById('example');
```

```
    element.addEventListener('click', function(event) {
      console.log('At ' + formatDate(new Date()) + ' BOOM once!');
    }, false);
    element.addEventListener('click', function(event) {
      console.log('At ' + formatDate(new Date()) + ' BOOM twice!');
    }, false);
    element.addEventListener('click', function(event) {
      console.log('At ' + formatDate(new Date()) + ' BOOM thrice!');
    }, false);
  </script>
 </body>
</html>
```

Establishes three event handlers on the img element ①

This code is simple, but it clearly shows how you can establish multiple event handlers on the same element for the same event type—something you weren't able to do easily with the Basic Event Model. Inside the `<script>` tag, you grab a reference to the image element and then establish *three* event handlers for the `click` event ①.

Loading this page into a standards-compliant browser (not Internet Explorer 8 and below) and clicking the image results in the display shown in figure 6.3.
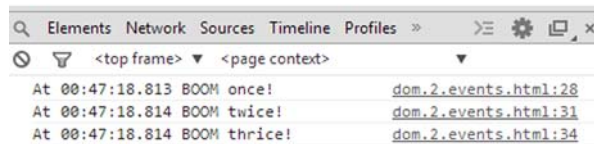

Figure 6.3   Clicking the image once demonstrates that all three handlers established for the `click` event are triggered.

Now that we've shown this very useful feature, you'll find out what's up with that `useCapture` parameter.

EVENT PROPAGATION

You saw earlier that with the DOM Level 0 Event Model, once an event was triggered on an element, the event propagated from the target element upward in the DOM tree to all the target's ancestors. The advanced Level 2 Event Model also provides this bubbling phase but ups the ante with an additional 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 the *capture phase*, and the latter (target to root) is called the *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 the capture phase, or as a bubble handler, to be triggered during the 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. This parameter has become optional, defaulting to false in newer versions of all major browsers (for example, in Firefox starting from version 6).

Think back a moment to the example of listing 6.2 where you explored the propagation of the Basic Model events through a DOM hierarchy. In that example, you embedded an image element within three layers of div elements. Within such a hierarchy, under DOM Level 2, the propagation of a click event with the img element as its target would move through the DOM tree as shown in figure 6.4.
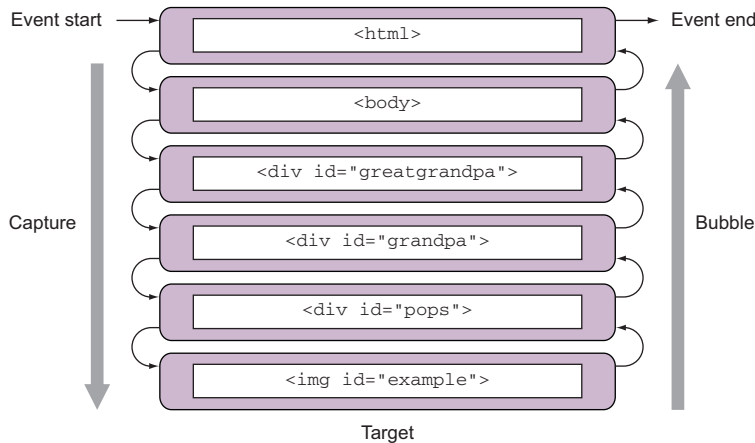

Figure 6.4   Propagation in the DOM Level 2 Event Model traverses the DOM hierarchy twice: once from top to target during the capture phase and once from target to top during the bubble phase.

Let's put that to the test, shall we? The next listing shows the code for a page containing the element hierarchy of figure 6.4 (chapter-6/dom.2.propagation.html).

Listing 6.4   Tracking event propagation with bubble and capture handlers

```
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Level 2 Propagation - jQuery in Action, 3rd edition</title>
    <link rel="stylesheet" href="../css/main.css"/>
    <style>
      img
      {
        display: block;
        margin: auto;
```

```
          }
        </style>
      </head>
      <body>
        <div id="greatgrandpa">
          <div id="grandpa">
            <div id="pops">
              <img id="example" src="images/example.jpg"
                    alt="A bolt of lightning" />
            </div>
          </div>
        </div>
        <script>
          function formatDate(date) {
            return (date.getHours() < 10 ? '0' : '') + date.getHours() +
                ':' + (date.getMinutes() < 10 ? '0' : '') + date.getMinutes() +
                ':' + (date.getSeconds() < 10 ? '0' : '') + date.getSeconds() +
                '.' + (date.getMilliseconds() < 10 ?
                    '00' : (date.getMilliseconds() < 100 ? '0' : '')) +
                date.getMilliseconds();
          }
          var elements = document.getElementsByTagName('*');
          for(var i = 0; i < elements.length; i++) {
            (function(current) {
              current.addEventListener('click', function(event) {
                console.log(
                    'At ' + formatDate(new Date()) +
                    ' Capture for ' + current.tagName + '#'+ current.id +
                    ' target is ' + event.target.tagName + '#' +
                    event.target.id
                );
              }, true);
              current.addEventListener('click', function(event) {
                console.log(
                    'At ' + formatDate(new Date()) +
                    ' Bubble for ' + current.tagName + '#'+ current.id +
                    ' target is ' + event.target.tagName + '#' +
                    event.target.id
                );
              }, false);
            })(elements[i]);
          }
        </script>
      </body>
    </html>
```

Establishes listener for the capture phase on the current element ❶

Establishes listener for the bubble phase on the current element ❷

This code changes the example of listing 6.2 to use the DOM Level 2 Event Model to establish the event handlers. In the <script> tag, you use the getElementsByTag-Name() method and the Universal selector to retrieve all the elements on the page. On each of them, you establish two handlers: one capture handler ❶ and one bubble handler ❷. Each handler prints a message on the console identifying the type of handler, the current element, and the ID of the target element.
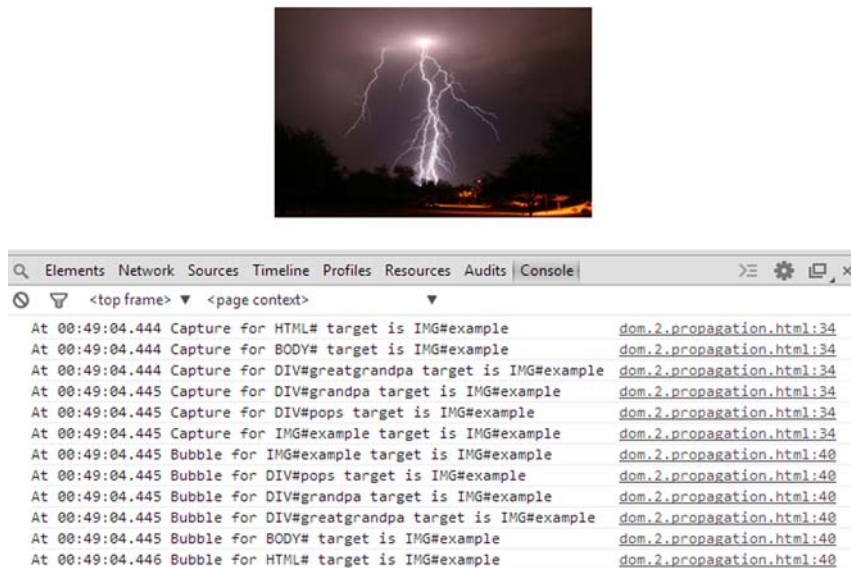
Figure 6.5   Clicking the image results in each handler emitting a console message that identifies the path of the event during both the capture and bubble phases.

With the page loaded into Chrome, clicking the image results in the display in figure 6.5 showing the progression of the event through the handling phases and the DOM tree. Note that because you defined both capture and bubble handlers for the target, two handlers were executed for the target and all its ancestor nodes.

Now that we've put you through all the trouble to understand these two types of handlers, you should know that capture handlers are hardly ever used in web pages. One of the historical reasons is that old versions of Internet Explorer don't support this type of event propagation.

Before we look at how jQuery can help sort out this mess, let's briefly examine the Internet Explorer Model.

6.1.3   The Internet Explorer Model

Versions of Internet Explorer prior to 9 don't provide support for the DOM Level 2 Event Model. These versions of Microsoft's browser provide a proprietary interface that closely resembles the bubble phase of the standard model. Rather than `add-EventListener()`, the Internet Explorer Model defines a method named `attach-Event()` for each DOM element. This method 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 but uses the name of the corresponding element property from the DOM Level 0 Model such as `onclick`, `onmouseover`, `onkeydown`, and so on.

The second parameter is the function to be established as the handler, and the `Event` instance must be fetched from the `window.event` property. In addition, this method doesn't support any semblance of a capture phase.

Even when using the relatively browser-independent DOM Level 0 Model, you'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, you even have to fork your code when establishing the handlers in the first place if you want to support a wider audience.

jQuery is going to simplify your life by hiding these browser disparities for you. Let's see how!

6.2   *The jQuery Event Model*

Although it's true that the creation of highly interactive 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 would be enough to daunt even the most intrepid of page authors.

You could hide the differences behind an API that abstracts the differences away from your page code, but why bother when jQuery has already done it for you?

jQuery's event model 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`
- Passes the `Event` instance as the first argument of 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 DOM Level 2 Event Model while supporting both standards-compliant browsers and older versions of Internet Explorer with a single API. The omission of the capture phase shouldn't 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 older IE. But is it really that simple? Let's find out.

6.2.1   *Attaching event handlers with jQuery*

Using the jQuery Event Model, you can attach event handlers to DOM elements with the `on()` method. So far, you've seen how to attach one or more handlers to a single element, but one of the advantages of using jQuery is that you can use its powerful `jQuery()` method to retrieve a set of elements and then attach the same handler to all of them in one statement. Consider the following simple example:

```
$('img').on('click', function(event) {
   alert('Hi there!');
});
```

This statement binds the supplied inline anonymous function as the `click` event handler for every image on a page. The full syntax of the `on()` method is as follows.

<div align="center">

Method syntax: on

</div>

```
on(eventType[, selector][, data], handler)
on(eventsHash[, selector][, data])
```
Attaches an event handler function for one or more events to the selected elements.

Parameters

`eventType`	(String) Specifies the name of the event type or types (a complete list can be found in table 6.1) for which the handler is to be established. Multiple event types can be specified by separating them with a space. These event types can be namespaced using a string as a suffix of the event name preceded by a period character (for example, `click.myapp`). Multiple namespaces are allowed (for example, `click.myapp.mymodule`).
`selector`	(String) An optional selector string used for *event delegation* to filter the descendants of the selected elements that trigger the event. If the selector is omitted, the event is always triggered when it reaches the selected element.
`data`	(Any) Data to be passed to the `Event` instance as a property named `data` and made available to the handler function.
`handler`	(Function) The function that's to be established as the event handler. When invoked, it will be passed the `Event` instance as its first argument, and its function context (`this`) is set to the current element of the bubble phase. The value `false` is also allowed as a shorthand for a function that does `return false;`. The function can also receive additional parameters through the use of `trigger()` or `triggerHandler()` (discussed later in this chapter).
`eventsHash`	(Object) A JavaScript object that allows handlers for multiple event types to be established in a single call. The property names identify the event type (same as would be used for the `eventType` parameter), and the property value provides the handler.

Returns

The jQuery collection.

---

jQuery 3: Signature improved

jQuery 3 allows the `handler` parameter to be an object. At the time of this writing there aren't many details about this improvement, but don't worry: this isn't a common use case. If you want to learn more about this feature, you can take a look at the related issue on GitHub: https://github.com/jquery/jquery/issues/1735.

Before delving into a more detailed discussion of this crucial jQuery method, let's put a basic example into action. Taking the code of listing 6.3 and converting it from the DOM Level 2 Event Model to the jQuery Event Model, you end up with the code shown in the next listing, which you can find in the file chapter-6/jquery.events.html.

Listing 6.5   Establishing advanced event handlers without browser-specific code

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Events in jQuery Example - jQuery in Action, 3rd edition</title>
    <link rel="stylesheet" href="../css/main.css"/>
    <style>
      img
      {
        display: block;
        margin: auto;
      }
    </style>
  </head>
  <body>
    <img id="example" src="images/example.jpg"
        alt=" A bolt of lightning" />

    <script src="../js/jquery-1.11.3.min.js"></script>
    <script>
      function formatDate(date) {
        return (date.getHours() < 10 ? '0' : '') + date.getHours() +
            ':' + (date.getMinutes() < 10 ? '0' : '') + date.getMinutes() +
            ':' + (date.getSeconds() < 10 ? '0' : '') + date.getSeconds() +
            '.' + (date.getMilliseconds() < 10 ?
                '00' : (date.getMilliseconds() < 100 ? '0' : '')) +
            date.getMilliseconds();
      }

      $('#example')
          .on('click', function (event) {
            console.log('At ' + formatDate(new Date()) + ' BOOM once!');
          })
          .on('click', function (event) {
            console.log('At ' + formatDate(new Date()) + ' BOOM twice!');
          })
          .on('click', function (event) {
            console.log('At ' + formatDate(new Date()) + ' BOOM thrice!');
          });
    </script>
  </body>
</html>
```

Binds event handlers to image using jQuery ❶

The changes to this code, limited to how you attached the event handlers ❶, are simple but important. You create a set consisting of the target img element and invoke the on() method three times on it—remember, jQuery chaining lets you apply multiple methods in a single statement—each of which establishes a click event handler on the element.

Loading this page into any browser supported by jQuery (you can finally forget the standards-compliant browser story!) and clicking the image results in the display shown in figure 6.6, which, not surprisingly, is the exact same result you saw in figure 6.3.
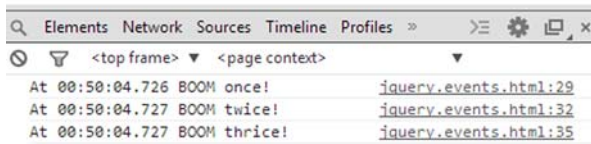
Figure 6.6   Using the jQuery Event Model allows you to specify multiple event handlers just like the DOM Level 2 Event Model.

This code also works in older versions of Internet Explorer (the specific versions depend on the jQuery version used), which wasn't possible using the code from listing 6.3 without adding browser-specific testing and branching code to use the correct event model for the current browser.

It's worth noting that, unlike other jQuery methods, the `selector` parameter isn't used at the time the `on()` method is called to further filter the objects in the jQuery collection that will receive the event handler. It's used at the time the event happens to determine whether or not the event handler is called. This concept will be clearer in a few pages where you'll see a concrete example, so please be patient.

jQuery 3: Methods deprecated

The `on()` method provides a unified interface to replace jQuery's `bind()`, `delegate()`, and `live()` methods. While `live()` was deprecated in version 1.7 and then removed in version 1.9, `bind()` and `delegate()` were still around but their use was strongly discouraged. jQuery 3 deprecates the `bind()` and `delegate()` methods, so we suggest you to stick with `on()` (which explains why we haven't covered these older methods in this book).

The `on()` method has an important difference compared to the native methods to attach event handlers. Returning `false` from an event handler added using this jQuery method is the same as calling `preventDefault()` and `stopPropagation()`, whereas returning `false` in an event handler added using a native method is equivalent to only invoking `preventDefault()`.

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?

Listing 6.5 shows how flexible and clever the `on()` method is. If you only want to pass a handler without specifying `data` or `selector`, you can do so without the need to

pass `null` for arguments. The `on()` method allows you to pass the handler as the second parameter. Instead of writing

```
$('img').on('click', null, null, function() { ... });
```

you can write

```
$('img').on('click', function() { ... });
```

Explaining how this result is achieved is outside the scope of this book, but we strongly encourage you to read the source of the method to learn it.

In the description of the `on()` method you learned that the first parameter can be a list of space-separated events. An example of code that uses this variant is the following:

```
$('button')
   .on('click', function(event) {
      console.log('Button clicked!');
   })
   .on('mouseenter mouseleave', myFunctionHandler);
```

In this simple snippet, you retrieve all the `<button>`s in the page and attach to them three event handlers. The first one, an anonymous function, is executed when the `click` event is triggered. The second handler, `myFunctionHandler`, is a hypothetical function declared elsewhere that's executed when either the event `mouseenter` or `mouseleave` is fired.

In the description of the signature of `on()`, we also highlighted how the first parameter can be a JavaScript object that we named `eventsHash`, where the property names identify the event type and the property value provides the handler. Using this approach, the previous code can be rewritten as follows:

```
$('button').on({
   click: function(event) {
      console.log('Button clicked!');
   },
   mouseenter: myFunctionHandler,
   mouseleave: myFunctionHandler
});
```

Which version to use is up to you, but we suggest you choose a style and stick with it.

Sometimes you may need to pass some data to the event handler. Although you can store the data in a variable and rely on closure, there are cases so simple that you can avoid this technique and use the `data` parameter. Let's say that you have a handler attached to the `click` event of a `button`, and this handler prints on the console the full name of a person. You can pass the name using the `data` parameter as shown here:

```
$('#my-button').on('click', {
   name: 'John Resig'
}, function (event) {
    console.log('The name is: ' + event.data.name);
});
```

As demonstrated by this snippet, you access the `name` property of the object via the `data` property of the `Event` instance (event). In case you want to see a live demo, this example is available in the file chapter-6/on.data.parameter.html and also as a JS Bin (http://jsbin.com/IVONuWol/edit?html,js,console,output).

So far, you've attached event handlers on DOM elements that are already in the HTML markup of the page. But what about elements that don't exist yet but *will*?

As we've discussed, jQuery allows you to dynamically manipulate the DOM by adding, modifying, or deleting elements. When you throw Ajax into the mix, it's likely that DOM elements will come into and out of existence frequently during the lifetime of the page. This is particularly true if you're dealing with single-page applications.

The solution to this issue is called *event delegation*. Event delegation is an important technique that suggests that you attach the handler to a parent of the element(s) instead of the element(s) itself. Event delegation can be employed in raw JavaScript as well, but this is one of the cases where jQuery shows its power in all its greatness. Imagine that you have an empty unordered list that has been filled with five list items using an Ajax call. At the time the scripts of the page were executed, the list was still empty:

```
<ul id="my-list"></ul>
```

After the Ajax call, the list resembles the following:

```
<ul id="my-list">
   <li>Item 1</li>
   <li>Item 2</li>
   <li>Item 3</li>
   <li>Item 4</li>
   <li>Item 5</li>
</ul>
```

You want to print on the console the index of a list item as soon as the mouse hovers over it. Now let's compare how you can implement event delegation in raw JavaScript and use jQuery to achieve your goal.

Using raw JavaScript, the code to perform this task looks like the following:

```
document.getElementById('my-list').addEventListener('mouseover',
   function(event) {
      if (event.target.nodeName === 'LI') {
         console.log('List item: ' +
            (Array.prototype.indexOf.call(
               document.getElementById('my-list').children,
               event.target
            ) + 1)
         );
      }
   },
   false
);
```

Using jQuery, you can reduce it all to three lines of code (using the same code style to be fair):

```
$('#my-list').on('mouseover', 'li', function(event) {
    console.log('List item: ' + ($(this).index() + 1));
});
```

That's clean code! Do you think all you've gained is fewer lines of code? No way! Did you consider that the raw JavaScript version doesn't take into account older versions of Internet Explorer whereas jQuery does? Of course, behind the scenes jQuery performs more or less the same operations, but why worry about compatibility issues when jQuery can do it for you? At this point you should start to understand why jQuery has been so widely adopted for developing websites. In case you want to play with these snippets, you can find the raw JavaScript code in the file chapter-6/javascript.event.delegation.html and as a JS Bin (http://jsbin.com/fihixa/edit?html,js ,console,output). The code showing how to implement event delegation with jQuery can be found in the file chapter-6/jquery.event.delegation.html and as a JS Bin (http://jsbin.com/bobaza/edit?html,js,console,output).

The advantages of event delegation aren't limited to executing handlers for elements that don't exist yet, though. It also allows you to save memory and time. Imagine if the Ajax call filled the list with hundreds of list items instead of just five as in our example. To attach the handler directly to the list items, you should have looped over each item (you could have used the jQuery() method, but behind the scenes it would have done nothing but iterate over each element). This task would have required a given amount of time, maybe even a few hundred milliseconds. While your loop is executing, the user interface of the browser is blocked and the user perceives poor performance. In addition, because JavaScript is a single-threaded language, no other operations can be performed while your loop is executing. It's pretty obvious that attaching a handler to just one (parent) element requires less time. We also said that event delegation enables you to save memory. In this example, you went from having to keep in memory hundreds of handlers to just one. A big savings! When implementing event delegation, you'll sometimes see code attached to document. Modifying our example, the jQuery code would resemble this:

```
$(document).on('mouseover', '#my-list li', function(event) {
    console.log('List item: ' + ($(this).index() + 1));
});
```

Note how you update the selector parameter to assure that you attach the handler on the same set of elements. Because of this example you may think that it's a good idea to attach all the handlers to document, but attaching many delegated handlers to an element near the root of the DOM or to document can degrade the performance. In such cases, every time an event is fired, jQuery has to compare the provided selector parameter with the element that fired the event for every element from the event target up to the top (because of event bubbling). To avoid this issue, a good practice is to attach delegated events to an element as close as possible to those you're targeting.

Up to this point, we've talked a lot about events and we've even used some of them. But how many are there, and what are these events? Table 6.1 lists all those that you can listen for.

Table 6.1   Events available for listening

Events			
blur	focusin	mousedown	mouseup
change	focusout	mouseenter	ready
click	keydown	mouseleave	resize
dblclick	keypress	mousemove	scroll
error	keyup	mouseout	select
focus	load	mouseover	submit
			unload

Now that you've learned all about how to attach event handlers, we can discuss a variation of `on()` that allows you to attach a handler that has to be fired only once and then removed.

LISTENING FOR AN EVENT ONCE

jQuery provides a specialized version of the `on()` method, named `one()`, that establishes an event handler as a one-shot deal. Once the event handler executes, it's automatically removed. Its syntax is identical to the `on()` method, so we'll omit the explanation of the parameters (if you need a refresher, you can refer to the syntax of `on()`).

Method syntax: one

one(eventType[, selector][, data], handler)
one(eventsHash[, selector][, data])

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.

Returns

The jQuery collection.

Up to this point, you've seen how to bind an event handler to a set of elements, but you may eventually need to remove it. Let's see how.

6.2.2   *Removing event handlers*

Typically, once an event handler is established using `on()`, it remains in effect for the remainder of the life of the page. But 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 to save memory. You've seen that the `one()` method can automatically remove a handler after it has completed its first (and only) execution, but for the more general cases where you'd like to remove event handlers under your own control, jQuery provides an `on()` counterpart called `off()`. The syntax of `off()`, shown next, has parameters with the same meaning described for `on()` and `one()`, so we won't repeat them.

Method syntax: off

```
off(eventType[, selector][, handler])
off(eventsHash, [, selector])
off()
```

Removes event handlers from all elements of the jQuery object as specified by the optional parameters given. If no parameters are provided, all listeners are removed from the elements.

Returns

The jQuery collection.

You can use this method to remove event handlers from the elements of the jQuery object at various levels of granularity. All listeners can be removed by omitting all the parameters. Listeners of a specific type can be removed by providing just that event type. If the name of one or more events is provided, all the handlers are removed, delegated or not. Finally, specific handlers can be removed by providing a reference to the function originally established as the listener. To do this, 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 `off()`.

---

jQuery 3: Methods deprecated

The `off()` method provides a unified interface to replace `unbind()`, `undelegated()`, and `die()` (what a scary name!). Like their respective counterparts, `die()` (a counterpart of `live()`) was deprecated in version 1.7 and then removed in version 1.9, whereas `unbind()` (a counterpart of `bind()`) and `undelegate()` (a counterpart of `delegate()`) were still in the core but their use was discouraged. jQuery 3 deprecates the `unbind()` and `undelegate()` methods, so we suggest you stick with `off()`.

In the case of anonymous inline functions, using namespaced events can come in quite handy, because you can unbind all events in a particular namespace without having to retain individual references to the listeners. For example,

```
$('*').off('.fred');
```

will remove all event listeners in namespace fred (remember that in this case the period in front of the name doesn't indicate a class selector). This use of namespaces is particularly useful when attaching handlers from a jQuery plugin, as we'll discuss in chapter 12.

Before moving on, let's see an example of using on() and off(). Consider the following markup where you have three buttons:

```
<button id="btn">Does nothing</button>
<button id="btn-attach">Attach handler</button>
<button id="btn-remove">Remove handler</button>
```

Moreover, you have the following code:

```
var $btn = $('#btn');
var counter = 1;

function logHandler() {
    console.log('click ' + counter);
    counter++;
};

$('#btn-attach').on('click', function() {
    $btn
        .on('click', logHandler)
        .text('Log');
});

$('#btn-remove').on('click', function() {
    $btn
        .off('click', logHandler)
        .text('Does nothing');
});
```

1 Defines a function that will print on the console the number of times it's executed

2 Attaches a handler to execute when the click event is fired on the btn-attach button

3 Attaches a handler to execute when the click event is fired on the btn-remove button

The idea is to have the first button, which does nothing by its own, and the other two buttons, which attach a handler to the first button and remove the handler, respectively. Inside the handler, you print on the console the number of times the button was clicked while the handler was attached **1**.

To perform this task, in the first part of the code you declare a variable, $btn, containing a set made of one element, the first button (the one having btn as its ID). Then you declare the counter (counter) and a function that you'll use later as the handler on the button. In the second part of the snippet, you attach an inline handler to the second button (the one having btn-attach as its ID), using the on() method **2**. Its aim is to attach the function logHandler as a handler of the first button when the click event is fired. In the same way, you attach an inline handler to the third button (the one having btn-remove as its ID) whose aim is to remove the handler from the first button **3**. You can see this example in action in the file chapter-6/adding .removing.handlers.html and also as a JS Bin (http://jsbin.com/iquRujug/ edit?html,js,console,output). Note that multiple clicks of the Attach handler button will add a number of identical event handlers, so the counter will jump accordingly at every click of the Log button.

So far, you've seen that the jQuery Event Model makes it easy to establish (as well as remove) event handlers without worrying about browser differences, but what about writing the event handlers themselves?

6.2.3   *Inspecting the Event instance*

When an event handler established with the `on()` method or the other ones we've mentioned is invoked, an `Event` instance is passed to it as the first parameter to the function regardless of the browser, eliminating the need to worry about the `window.event` property under older versions of Internet Explorer. But that still leaves you dealing with the divergent properties of the `Event` instance, doesn't it?

Thankfully, no, because jQuery doesn't really pass the `Event` instance to the handlers. *Screech!* (sound of a needle being dragged across a record).

Yes, we've been glossing over this little detail because, up until now, it hasn't mattered. But now that we've advanced to the point where we're going to examine the instance within handlers, the truth must be told!

In reality, jQuery defines an object of type `jQuery.Event` that it passes to the handlers. But you can forgive us for our simplification because jQuery copies most of the original `Event` properties to this object. Therefore, if you look for only the properties that you expected to find on `Event`, the object is almost indistinguishable from the original `Event` instance.

But that's not the important aspect of this object; what's really valuable, and the reason that this object exists, is that it holds a set of normalized values and methods that you can use independently of the containing browser, ignoring the differences in the `Event` instance.

Table 6.2 lists the `jQuery.Event` properties that are safe to access in a platform-independent manner. Note that some properties may have the value `undefined` depending on the event triggered.

Table 6.2   **Browser-independent `jQuery.Event` properties**

Properties			
altKey	currentTarget	offsetY	screenX
bubbles	data	originalTarget	screenY
button	detail	originalEvent	shiftKey
cancelable	delegateTarget	pageX*	target*
charCode	eventPhase	pageY*	timeStamp
clientX	metaKey*	prevValue	type
clientY	namespace	relatedTarget*	view
ctrlKey	offsetX	result	which*

As you can see, some of the properties have been marked with an asterisk. The reason is that jQuery normalizes them for cross-browser consistency. What this means is that they're named differently in some browsers (yes, older versions of Internet Explorer). To avoid the pain of remembering all these differences, jQuery provides one property

name and takes care of filling the holes. One example is the `target` property, which in older versions of Internet Explorer is called `srcElement`. In addition, some events may have specific properties that you can access through the `originalEvent` property.

The `jQuery.Event` object also has several methods, described in table 6.3.

Table 6.3   Browser-independent `jQuery.Event` methods

Methods	
`preventDefault()`	Prevents any default semantic action (such as form submission, link redirection, check box state change, and so on) from occurring.
`stopPropagation()`	Stops any further propagation of the event up the DOM tree. Additional events on the current target aren't affected. Works with browser-defined events as well as custom events.
`stopImmediatePropagation()`	Stops all further event propagation including additional events on the current target.
`isDefaultPrevented()`	Returns `true` if the `preventDefault()` method has been called on this instance.
`isPropagationStopped()`	Returns `true` if the `stopPropagation()` method has been called on this instance.
`isImmediatePropagationStopped()`	Returns `true` if the `stopImmediatePropagation()` method has been called on this instance.

In addition to allowing you to manage event handling and the `Event` object in a browser-independent manner, jQuery provides a set of methods that gives you the ability to trigger events or run event handlers under script control. Let's look at those.

6.2.4   *Triggering event handlers*

Event handlers are designed to be invoked when browser or user activity triggers the propagation of their associated events through the DOM hierarchy. But there may be times when you want to trigger the execution of a handler under script control. You could define such event handlers as top-level functions so that you can invoke them by name, but as you've seen, defining event handlers as inline anonymous functions is much more common and so darned convenient! Moreover, calling an event handler as a function doesn't cause semantic actions or bubbling to occur.

For this need, jQuery has provided methods that will automatically trigger event handlers on your behalf under script control. The most general of these methods is `trigger()`, whose syntax is as follows.

<table>
<tr><td colspan="2" align="center">**Method syntax: trigger**</td></tr>
</table>

`trigger(eventType[, data])`

Invokes any event handlers and behaviors established for the passed event type for all matched elements.

Parameters

event	(String|jQuery.Event) Specifies the name of the event type for which handlers are to be invoked, including namespaced events. Alternatively a `jQuery.Event` can be passed.
data	(Any) Data to be passed to the handlers. If an array is provided, the elements are passed to the handler as different parameters.

Returns

The jQuery collection.

The `trigger()` method, as well as the convenience methods that we'll introduce in a moment, do their best to simulate the event to be triggered, including propagation through the DOM hierarchy and the execution of semantic actions.

Each handler called is passed a populated instance of `jQuery.Event`. Because there's no real event, properties that report event-specific values, such as the location of a mouse event or the key of a keyboard event, aren't passed (the properties don't exist at all, which is different from having their value as `undefined`). The `target` property is set to reference the element of the matched set to which the handler was bound.

Just as with actual events, triggered event propagation can be halted via a call to the `jQuery.Event` instance's `stopPropagation()` method, or a `false` value can be returned from any of the invoked handlers.

The `data` parameter passed to the `trigger()` method isn't the same as the one passed when a handler is established. The latter is placed into the `jQuery.Event` instance as the `data` property; the value passed to `trigger()` (and, as you're about to see, `triggerHandler()`) is passed as a parameter to the listeners. This allows both data values to be used without conflicting with each other.

Before moving to `triggerHandler()`, let's discuss in detail the `data` parameter of the `trigger()` method and how passing an array differs from passing any other data type. For the sake of the discussion, consider the following snippet, which adds an event handler on a hypothetical element having `foo` as its ID that's executed when the `click` event is fired:

```
$('#foo').on('click', function(event, par1, par2, par3){
   console.log(par1, par2, par3);
});
```

As you can see, you define the `event` parameter as usual but also three new parameters: `par1`, `par2`, and `par3`. Inside the handler you print on the console the value of these three new parameters. Now imagine that you employ jQuery's `trigger()`

method to execute this handler as shown here, passing three numbers (1, 2, and 3) in addition to the event to trigger:

```
$('#foo').trigger('click', 1, 2, 3);
```

As a result, you'll see the following logged on the console:

```
1 undefined undefined
```

This happens because the trigger method accepts only one argument to pass data to the handler, so all the others (2 and 3 in this case) are ignored. If you want to provide more than one argument, you can employ an array as shown here:

```
$('#foo').trigger('click', [1, 2, 3]);
```

Once the handler attached to the element is executed, the following line will be printed on the console, proving that all the elements of the array have been passed as separated parameters (par2 and par3 aren't undefined anymore):

```
1 2 3
```

In case you want to play further with the data parameter, we've set up a demo for you that you can find in the file chapter-6/trigger.data.parameter.html and also as a JS Bin (http://jsbin.com/mefohu/edit?html,js,console,output).

   For cases where you want to trigger a handler but not cause propagation of the event and execution of semantic actions, jQuery provides the triggerHandler() method, which looks and acts just like trigger() except that no bubbling or semantic actions will occur.

Method syntax: triggerHandler

triggerHandler(eventType[, data])
Invokes any event handlers established for the passed event type for all matched elements without bubbling, semantic actions, or live events. This method returns whatever value was returned by the last handler it caused to be executed, or undefined.

Parameters

eventType	(String) Specifies the name of the event type for which handlers are to be invoked.
data	(Any) Data to be passed to the handlers. If an array is provided, the elements are passed to the handler as different parameters.

Returns

Any. If no handler is triggered or the handler doesn't return a value, undefined is returned.

The triggerHandler() method has the data parameter in common with trigger(), so the previous demo works with triggerHandler() as well, but they also have some important differences that are worth discussing. The first one is that trigger() acts on all the elements in the jQuery collection, whereas triggerHandler() operates on only the first one. In addition, triggerHandler() doesn't allow for chaining because it returns as its value the same value returned by the handler executed (if no handler is triggered or the handler doesn't return a value, it returns undefined), whereas

trigger() returns the set of matched elements. Finally, events fired with trigger-
Handler() don't bubble up the DOM hierarchy.

No words can replace a good example, so before moving to the next section, it's
time to see both trigger() and triggerHandler() in action (this book is still called
jQuery in Action, isn't it?). The code of the example is shown in the following listing
and can be found in the downloadable code in the file chapter-6/jquery.triggering
.events.html and as a JS Bin (http://jsbin.com/AqaqAGO/edit?html,css,js,console
,output).

Listing 6.6 Triggering events in jQuery

```html
<!DOCTYPE html>
<html>
   <head>
      <title>Triggering Events - jQuery in Action, 3rd edition</title>
      <link rel="stylesheet" href="../css/main.css"/>
      <style>
         #wrapper
         {
            border: 1px solid #3A5895;
            padding: 10px;
         }

         #address:focus
         {
            border: 3px solid #000000;
         }
      </style>
   </head>
   <body>
      <div id="wrapper">
         <button id="btn">Click me!</button>
         <input type="text" id="address" />
      </div>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         $('#wrapper')
            .on('focus', function() {
               console.log('Div focused');
            })
            .on('click', function() {
               console.log('Div clicked');
            });

         $('#address')
            .on('focus', function() {
               console.log('Input focused');
            })
            .triggerHandler('focus');

         $('#btn')
            .on('click', function() {
               console.log('Button clicked');
```

1 Attach two handlers to div element for focus and click events.

2 Attach handler to input element, executed when focus event is triggered.

3 Trigger focus event on input element using triggerHandler().

4 Attach handler to button element, executed when click event is triggered.

```
          })
          .trigger('click');
      </script>
  </body>
</html>
```

Trigger click event on button
⑤ element using trigger().

The code in this listing allows you to see the behavior of trigger() and trigger-Handler() discussed previously. Opening the page, you should see a layout, as shown by figure 6.7.
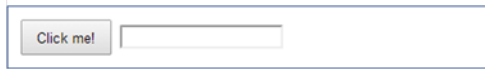

Figure 6.7   Layout of the page jquery.triggering.events.html

In listing 6.6 you create a <div> acting as a wrapper of the other two elements of the page: a button and an input box. In the usual script element, you first select the wrapper element, attaching two listeners to it using the on() method: one for the focus event and one for the click event ①. Inside them, you print a message on the console to explain what element and event have been called. Then you attach a handler for the focus event on the input element that has address as its ID ②. Thanks to chaining, before closing the statement with the semicolon, you also call the triggerHandler() method, passing the string focus as its argument ③. By calling triggerHandler(), you cause the execution of the function that you've just attached in ②. As we said, triggerHandler() doesn't bubble up, so the handler attached to the element, having wrapper as its ID for the same event, won't be executed.

Finally, you select the button element, attaching to it a handler for the click event ④. As for the input element, before closing the statement, you trigger an event but this time using trigger() ⑤. Therefore, after printing on the console the message set in its handler, the event bubbles up so that the handler attached to the click event of the div is executed as well.

Based on the discussion of this example, as soon as you load the page, the output of the console should be as shown in figure 6.8.
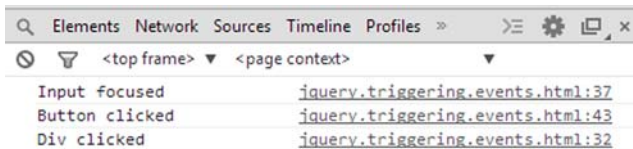

Figure 6.8   Output of the page jquery.triggering.events.html

Methods like on() and trigger() are frequently used, so writing their full syntax each time rapidly becomes annoying. The jQuery team knows it, so they introduced a set of shortcut methods.

6.2.5 *Shortcut methods*

jQuery provides a handful of shortcut methods to establish specific event handlers as well as trigger events. Because the syntax of each of these methods is identical except for the name of the method, we'll save some space and present them all in the following single syntax descriptor.

Method syntax: specific event binding

eventName`([data,] handler)`

Establishes the specified function as the event handler for the event named by the method's name. The supported methods are as follows:

blur	focusout	mouseleave	resize
change	keydown	mousemove	scroll
click	keypress	mouseout	select
dblclick	keyup	mouseover	submit
focus	mousedown	mouseup	
focusin	mouseenter	ready	

Parameters

data	(Any) Data to be passed to the `Event` instance as a property named `data` and made available to the handler function.
handler	(Function) The function that's to be established as the event handler.

Returns

The jQuery collection.

In addition to the ability to attach a handler, these methods can act as the `trigger()` method. The syntax for all these methods is exactly the same except for the method name, and that syntax is as follows.

Method syntax: Specific event triggering

eventName`()`

Invokes any event handler and behavior established for the named event type for all matched elements. The supported methods are as follows:

blur	focusout	mouseleave	scroll
change	keydown	mousemove	select
click	keypress	mouseout	submit
dblclick	keyup	mouseover	
focus	mousedown	mouseup	
focusin	mouseenter	resize	

Parameters

none	

Returns

The jQuery collection.

> ### jQuery 3: Methods removed
>
> jQuery 3 gets rid of the already deprecated `load()`, `unload()`, and `error()` shortcut methods. These methods weren't listed in the previous descriptions because they were deprecated a long time ago (since jQuery 1.8). If you're still using them in your projects or you're employing a plugin that relies on one or more of them, upgrading to jQuery 3 will break your code.

To give you an idea of what a shortcut looks like, let's change the last statement of listing 6.6 so that it employs them. For ease of reading we report the statement here:

```
$('#btn')
    .on('click', function() {
        console.log('Button clicked');
    })
    .trigger('click');
```

Using the shortcuts just discussed, you can change it as follows:

```
$('#btn')
    .click(function() {
        console.log('Button clicked');
    })
    .click();
```

In addition to these shortcuts, there's another one in jQuery that's a bit different.

HOVERING OVER ELEMENTS

A common multi-event scenario that's frequently employed in interactive applications involves mousing into and out of elements. Events that inform you 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 your pages. Among these element types, cascading menus used as navigation systems are a common example.

A vexing behavior of the `mouseover` and `mouseout` event types often hinders the easy creation of such elements: a `mouseout` event fires as the mouse is moved over an area that is covered by its children. Consider the display in figure 6.9 that shows the layout of the file chapter-6/hover.html, which is also available as a JS Bin (http://jsbin.com/nobuti/edit?html,js,console,output).

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.
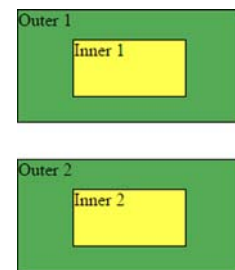

Figure 6.9 This page helps demonstrate when mouse events fire as the mouse pointer is moved over an area and its children.

For the top set of rectangles on the page, the following statements establish handlers for the `mouseover` and `mouseout` events:

```
$('#outer1').on('mouseover mouseout', report);
$('#inner1').on('mouseover mouseout', report);
```

These statements establish a function named `report` as the event handler for both the `mouseover` and `mouseout` events defined as follows:

```
function report(event) {
   event.stopPropagation();
   console.log(event.type + ' on ' + event.target.id);
}
```

This listener first stops the event from bubbling up and then prints some text on the console containing the name of the event and the ID of the element it was fired.

Now move the mouse pointer into the area labeled "Outer 1" (being careful not to enter "Inner 1"). You'll see on the console that a `mouseover` event has fired. Move the pointer back out of the area and, as expected, you'll see that a `mouseout` event has fired.

Now move the mouse pointer into "Outer 1" but this time continue inward until the pointer enters "Inner 1". As the mouse enters "Inner 1", a `mouseover` event is fired for it and a `mouseout` event fires for "Outer 1". If you wave your pointer back and forth over the boundary between "Outer 1" and "Inner 1", you'll see a flurry of `mouseout` and `mouseover` events. This is the defined behavior, even if it's rather unintuitive. 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 to be leaving the outer area.

Expected or not, you don't always want that behavior. Often you 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.

Luckily, major browsers support a pair of mouse events, `mouseenter` and `mouseleave`, first introduced by Microsoft in Internet Explorer. This event pair acts slightly more intuitively, not firing a `mouseleave` event when moving from an element to a descendant of that element.

Using jQuery you could establish handlers for this set of events using the following code:

```
$(element).mouseenter(function1).mouseleave(function2);
```

But jQuery also provides a single method that makes it even easier: `hover()`. The syntax of this method is as follows.

Method syntax: hover

```
hover(enterHandler, leaveHandler)
hover(handler)
```
Establishes handlers for the `mouseenter` and `mouseleave` events for matched elements. These handlers fire only when the area covered by the elements is entered and exited, ignoring transitions to child elements.

Parameters

`enterHandler`	(Function) The function to become the `mouseenter` handler.
`leaveHandler`	(Function) The function to become the `mouseleave` handler.
`handler`	(Function) A single handler to be called for both `mouseenter` and `mouseleave` events.

Returns

The jQuery collection.

Use the following script to establish mouse event handlers for the second set of areas ("Outer 2" and its "Inner 2" child) of the example page:

```
$('#outer2').hover(report);
$('#inner2').hover(report);
```

As with the first set of areas, the `report()` function is established as both the `mouseenter` and `mouseleave` handlers for "Outer 2" and "Inner 2". But unlike the first set of areas, when you pass the mouse pointer over the boundary between "Outer 2" and "Inner 2", neither of these handlers (for "Outer 2") is invoked. This is useful for those situations where you have no need for parent handlers to react when the mouse pointer passes over child elements.

Let's now see how it's possible to create *custom events* in jQuery.

6.2.6 *How to create custom events*

Creating custom events in jQuery is straightforward and requires the use of the methods we've discussed so far. Custom events are a convenient way to execute one or more statements based on a given condition that may happen in different parts of your code. Let's say that you have a set of statements to execute that are logically related. You can group them to create a handler and then trigger your custom event when needed. To be honest, the event doesn't need to be created in any formal way but only listened to and triggered. This means that you can attach a handler for a custom event using the `on()` method, passing as its first argument the name of the new event. Then you can fire it using `trigger()` and passing the same name.

A basic example of creating and using a custom event is shown here:

```
$('#btn').on('customEvent', function(){
    alert('customEvent');
});
$('#anotherBtn').click(function() {
    $('#btn').trigger('customEvent');
});
```

In this code you attach a handler for a custom event called `customEvent` to the element having ID of `btn`. Then you attach a handler to the element having ID of `anotherBtn` that, once clicked, triggers the `customEvent` event. Because you fired the `customEvent` event on the element having ID of `btn`, the alert will be shown.

Keep in mind that jQuery won't create at runtime a shortcut with the same name of a custom event, so writing

```
$('#btn').customEvent();
```

will throw an error.

In addition to custom events, jQuery allows you to namespace events. We introduced this feature in the previous sections without covering it in detail. It's time to fill in the gap.

6.2.7 *Namespacing events*

Another nifty little event-handling extra that jQuery provides 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. If you'd like, you can use multiple suffixes to place the event into multiple namespaces, as we cited in the description of the `on()` method. By grouping event bindings in this way, you 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 elements of the page, but these listeners aren't appropriate for display mode and need to be removed when the page transitions out of edit mode. You could namespace the edit mode events with code such as this:

```
$('.my-class').on('click.editMode', myFunction);
```

By grouping all these bindings into a namespace called `editMode`, you can later operate upon them as a whole. For example, you can remove all the events namespaced under `editMode` from all the elements of the page with the following statement:

```
$('*').off('.editMode');
```

As we said, jQuery also allows you to use multiple namespaces for a given event. In the following example you can see this feature:

```
$('.elements').on('click.myApp.myName', myFunction);
```

Namespaces are case sensitive, so if you have the previous statement and execute the following

```
$('.elements').trigger('click.myapp');
```

the handler attached to the event won't be executed (note the lowercase `a`).

There's another important concept to highlight. Imagine you have the following code:

```
$('.elements').on('click.myApp.myName', myFunction);
$('.other-elements').on('click.myApp', myOtherFunction);
```

You want to execute all the event handlers attached to a `click` event that has the namespace `myApp`. This means that you want to execute both `myFunction()` and `myOtherFunction()`. To do so you don't have to execute two different statements like these:

```
$('.elements').trigger('click.myApp');
$('.other-elements').trigger('click.myApp.myName');
```

You can select all the elements of both the sets and fire the `click` event, specifying only the `myApp` namespace:

```
$('.elements, .other-elements').trigger('click.myApp');
```

If a comparison will help you, you can consider multiple namespaces acting as the OR logical operator. If a namespaced event is fired, all the handlers attached to events having as one of their namespaces the one specified will be executed.

With all these event-handling tools under your belt, you'll employ what you've learned so far in the next chapter and look at an example page that makes use of them, as well as some of the other jQuery techniques that you've learned from previous chapters.

6.3   Summary

Building upon the jQuery knowledge that you've gained so far, this chapter introduced you to the world of event handling.

You learned that there are vexing challenges to implementing event handling in web pages, but such handling is essential for creating pages in interactive web 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 popular 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 in order to deal with differences in the `Event` instance. Although simple, this model limits you to only one listener for any event type on a particular DOM element.

You 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's supported only by standards-compliant browsers such as Chrome, Firefox, Internet Explorer 9 and above, Safari, and Opera.

For Internet Explorer 8 and below, an API-based proprietary event model that provides a subset of the functionality of the DOM Level 2 Event Model is available.

Coding all event handling in a series of `if` statements—one clause for the standard browsers and one for older versions of Internet Explorer—is a good way to drive yourself to early dementia. Luckily, jQuery comes to the rescue and saves you from that fate.

The library provides a general `on()` method to establish event listeners of any type on any element, as well as event-specific convenience methods 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 by exposing the `off()` method, or cause them to be triggered under script control. As if all that wasn't enough, jQuery provides the possibility of using the `on()` method to assign handlers proactively to elements that may not even exist yet.

Finally, we discussed how to create custom events and namespacing events, explaining the advantages and the use cases for these features.

In this chapter, we looked at a few examples of using events in your pages, and we explored a comprehensive example that demonstrated many of the concepts that you've learned up to this point. In the next chapter, we'll look at how to put together the concepts explained so far to create a nice application built on top of jQuery.

Demo: DVD discs locator

We're not even at the halfway point of the book and hopefully you've learned a lot of new topics, methods, and techniques. Throughout the previous chapters, we've covered jQuery selectors, DOM traversal and manipulation, and event handling. For each of these topics we've shown you several examples. They were great for letting you focus on a single aspect and helping you fix the concept, but they were limited.

In this chapter, we'll cover all the previously mentioned topics and also try to fill the gap by providing a demonstration of what you can do with the knowledge you've acquired. In the next few sections you'll develop a basic yet fully functional application to manage a collection of DVDs. Let's see what you can do!

7.1 Putting events (and more) to work

Let's pretend that you're a videophile whose collection of DVDs, numbering in the thousands, has become a huge problem. Not only has organization become an issue, making it hard to find a DVD quickly, but all those DVDs in their cases have become a storage problem. They've taken over way too much space and will get you thrown out of the house if the problem isn't solved.

We'll posit that you solved the storage side of the problem by buying DVD binders that hold one hundred DVDs each in much less space than the comparable number of DVDs in their cases. But although that saved you from having to sleep on a park bench, organizing the DVD discs is still an issue. How will you find a DVD that you're looking for without having to manually flip through each binder until you find the one you're seeking?

You can't do something like sort the DVDs in alphabetical order to help quickly locate a specific disc. That would mean that every time you bought a new DVD, you'd need to shift all the discs in perhaps dozens of binders to keep the collection sorted. Imagine the job ahead of you if you bought *Armageddon!*

Well, you have a computer, you have the know-how to write web applications, and you have jQuery! You'll solve the problem by writing a DVD database program to help keep track of what DVDs you have and where they are. The code for this example can be found in the file chapter-7/dvds.html.

> **NOTE** This demo has been heavily changed compared to the one presented in the second edition of this book. If you bought it (thank you!), we really encourage you not to skip this chapter. You'll find a lot of new stuff, and this time the filters really work (in the old version you had always the same, static results)!

The project you're going to build uses a few Ajax calls to perform some tasks. We haven't covered this topic yet, but we assure you that this lack won't make it harder for you to see all the other concepts in action. Due to the security restrictions of some browsers, the demo can raise an error. We already advised you about this issue in chapter 3, but for your convenience we're repeating the same note here.

> **NOTE** Due to security restrictions of some browsers, you may fail in playing with this demo. To avoid this issue, you can either execute the page under a web server like Apache, Tomcat, or IIS or search for a specific solution for your browser. For example, in WebKit-based browsers, you can run it through the command-line interface (CLI) using the flag `--allow-file-access-from-files`. It's important that the command creates a new process, so it must open not a new tab but a new window.

Let's get to work!

7.1.1   Filtering large data sets

Our DVD database program is faced with the same problem of many other applications, web-delivered or otherwise. How do you allow your users (in this case yourself) to quickly find the information they seek?

You could be all low-tech about it and display a sorted list of all the titles, but that would still be painful to scroll through if there's anything more than a handful of entries. Besides, you want to learn how to do it right so that you can apply what you learn to real, customer-facing applications. So no shortcuts!

Obviously, designing a complex application would be well beyond the scope of this chapter. Therefore, we'll concentrate on developing a control panel that allows you to specify filters with which you can tune the list of titles returned when you perform a database search.

You'll want the ability to filter on the DVD title, of course. But you'll also add the ability to filter the search based on the year that the movie was released, the binder in which you placed the disc, and even whether you've viewed the movie yet or not. (This will help answer the commonly asked question, "What should I watch tonight?")

Your initial reaction may be to wonder what the big deal is. After all, you can put up a number of fields and be done with it, right? Well, not so fast.

A single field for something like the title is fine if, for example, you want to find all movies with the word *creature* in their title. But what if you want to search for *creature* only if the movie was released between 1987 and 1999?

In order to provide a robust interface for specifying filters, you'll need to specify multiple filters for different properties of the DVD. But in this project you won't allow specifying the same filter multiple times. In addition, rather than trying to guess how many filters will be needed, you'll create them on demand.

Each filter is identified by a drop-down (single-selection `select` element) that specifies the field that's to be filtered. Based on the type of that field (string, date, number, and even Boolean), the appropriate controls are displayed on the line to capture information about the filter. The users are given the ability to add as many of these filters as they like but, once again, one of the same type, or to remove previously specified filters.

A picture being worth a thousand words, study the time-progression display of figures 7.1a through 7.1c. They show the filter panel that you'll build (a) when initially displayed, (b) after a filter has been specified, and (c) after a number of filters have been specified.

As you can see by inspecting the interactions shown in figures 7.1a through 7.1c, there's going to be a lot of element creation on the fly. Let's take a few moments to discuss how you're going to go about that.

Figure 7.1a   The display initially shows a single, unconfigured filter.

After a field is selected, the qualifiers are added.

Figure 7.1b   After a filter type is selected, its qualifier controls are added.

The user can add multiple filters.

Figure 7.1c   The user can add multiple filters.

7.1.2   *Element creation by template replication*

You can readily see that to implement this filtering control panel, you're going to need to create a fair number of elements in response to various events. For example, you'll need to create a new filter entry whenever the user clicks the Add Filter button and new controls specific for that filter whenever a specific field is selected.

No problem! As you've learned, jQuery allows you to dynamically create elements using the `$()` function. Although you'll do some of that in this example, you're also going to explore some higher-level alternatives.

When you're dynamically creating lots of elements, all the code necessary to create those elements and stitch together their relationships can get a bit unwieldy and difficult to maintain, even with jQuery's assistance. (Without jQuery's help, it can be a complete nightmare!) It would be great if you could create a "blueprint" of the complex markup using HTML and then replicate it whenever you needed an instance of the blueprint.

Yearn no more! The jQuery `clone()` method gives you just that ability.

The approach that you're going to take is to create sets of template markup that represent the HTML fragments you'd like to replicate and use the `clone()` method whenever you need to create an instance of that template. You don't want these templates to be visible to the end user, so you'll wrap them in a `div` element that's hidden from view using CSS.

As an example, consider the combination of the X button and drop-down that identifies the filterable fields. You'll need to create an instance of this combination every time the user clicks the Add Filter button. The jQuery code to create such a button and the `select` element, along with its child `option` elements, could be considered a tad long, although it wouldn't be too onerous to write or maintain. But it's easy to envision that anything more complex would get unwieldy quickly.

Using our template technique, and placing the template markup for that button and drop-down in a parent `<div>` used to hide all the templates, create the markup as follows:

```
<div class="templates">                          ❶ Encloses and hides all templates
  <div class="template filter-chooser">
    <input type="button" class="filter-remover" value="X" />

    <select name="filter" class="filter-type">
      <option value="" data-template-type="" selected="selected">
        Choose a filter
      </option>
      <option value="title" data-template-type="template-title">
        DVD Title
      </option>
      <option value="binder" data-template-type="template-binder">
        Binder
      </option>
      <option value="year" data-template-type="template-year">
        Release Date
      </option>
      <option value="viewed" data-template-type="template-viewed">
```

Defines the filter-chooser template ❷

```
      Viewed?
    </option>
  </select>
</div>
<!—- more templates go here -->
</div>
```

The outer `<div>` with `class` of `templates` serves as a container for all your templates and will be given a CSS declaration `display: none;` to prevent it from being displayed **❶**. Within this container, you define another `<div>` that you give the classes `template` and `filter-chooser` **❷**. You'll use the `template` class to identify (single) templates in general and the `filter-chooser` class to identify this particular template type. You'll see how these classes are used as JavaScript hooks shortly.

Also note that each `<option>` in the `<select>` has been given a custom attribute: `data-template-type`. You'll use this value to determine what type of filter controls need to be used for the selected filter field.

Based on which filter type is identified, you'll populate the remainder of the filter entry line with controls that are appropriate for the filter type. For example, if the template type is `template-title`, you'll want to display a text field into which the user can type a title (or part of it) to search and a drop-down giving them options for how that term is to be applied (contains, equal to, and so on).

You'll set up the template for this set of controls as follows:

```
<div class="template template-title">
   <select name="title-condition">
      <option value="contains">contains</option>
      <option value="starts-with">starts with</option>
      <option value="ends-with">ends with</option>
      <option value="equal">is exactly</option>
   </select>
   <input type="text" name="title" />
</div>
```

Again, you use the `template` class to identify the element as a template, and flag the specific template with the class `template-title`. We've purposely made it such that this class matches the `data-template-type` value on the field chooser drop-down.

Replicating these templates whenever and wherever you want is easy using the jQuery knowledge you have under your belt. Let's say that you want to append a template instance to the end of an element that you have a reference to in a variable named `whatever`. You could write

```
$('div.template.template-title')
   .clone()
   .appendTo(whatever);
```

In this statement, you select the template container to be replicated (in this case, the one having class `template-title`) using those convenient classes you placed on the template markup. Then you clone the element using the `clone()` method, and finally you attach the template to the end of the contents of the element identified by `whatever`. See why we keep emphasizing the power of jQuery method chains?

Inspecting the options of the `filter-chooser` drop-down, you see that you have a number of other template types defined: `template-binder`, `template-year`, and `template-viewed`. You'll define controls templates for those filter types as well with this code:

```
<div class="template template-binder">
   <input type="text" name="binder-min" class="numeric" />
   <span>through</span>
   <input type="text" name="binder-max" class="numeric" />
</div>
<div class="template template-year">
   <input type="text" name="year-min" class="date" />
   <span>through</span>
   <input type="text" name="year-max" class="date" />
</div>
<div class="template template-viewed">
   <label><input type="radio" name="viewed" value="true" checked="checked" />
     Yes</label>
   <label><input type="radio" name="viewed" value="false" /> No</label>
</div>
```

Binder filter template ⟵

Year filter template ⟵

Viewed filter template ⟵

Okay. Now that you have your replication strategy defined, let's take a look at the primary markup.

7.1.3   *Setting up the mainline markup*

If you refer back to figure 7.1a, you can see that the initial display of your DVD search page is pretty simple: a header, a first filter instance, a few buttons, and a preset table where you'll display the results. Take a look at the HTML markup that achieves that:

```
<h1>Disc Locator</h1>

<form id="form-filters" action="#">
   <fieldset>
      <legend>Filters</legend>
      <div id="filters">
      </div>
      <div class="buttons-wrapper">
         <input type="button" id="filter-add" value="Add Filter" />
         <input type="submit" id="filter-apply" value="Apply Filters"/>
      </div>
   </fieldset>
</form>

<div id="panel-results">
   <table id="results">
      <tr>
         <th>Title</th>
         <th>Year</th>
         <th>Binder</th>
         <th>Page</th>
         <th>Slot</th>
         <th>Viewed</th>
      </tr>
```

❶ Container for filter instances

❷ Container for search results

```
      </table>
</div>
```

There's nothing too surprising in that markup—or is there? Where, for example, is the markup for the initial filter drop-down? You've set up a container in which the filters will be placed ❶, but it's initially empty. Why?

Well, you're going to need to be able to populate new filters dynamically—which we'll get to in just a moment—so why do the work in two places? As you'll see, you'll be able to use the dynamic code to initially populate the first filter, so you don't need to explicitly create it in the static markup. One other thing that we should point out is that you've set aside a table to receive the results ❷.

You have your simple, mainline HTML laid out, and you have a handful of hidden templates that you can use to quickly generate new elements via replication. Finally you can start writing the code that will apply the behavior to your page!

7.1.4 *Adding new filters*

Upon a click of the Add Filter button, you need to add a new filter to the `<div>` that you've set up to receive it, which you've identified with the ID of `filters`. If you recall how easy it is to establish event handlers using jQuery, it should be an easy matter to add a click handler to the Add Filter button. But there's something else to consider!

You've already seen how you're going to replicate form controls when the user adds filters, and you have a good strategy for easily creating multiple instances of these controls. But eventually you're going to have to submit these values to the server so it can look up the filtered results in the database. Writing the backend is outside the scope of this chapter, but this doesn't mean that your application won't work. You'll simulate a database with a file called movies.json, which contains a JSON array. The movies.json file is bundled with the source of this book, and you'll find it in the folder called chapter-7. Each element of the array is an object having the following properties: `title`, `year`, `binder`, `page`, `slot`, and `viewed`.

To avoid loading the JSON object every time you interact with the application, you'll load it once when the page is loaded and store it in a global variable called `movies`. To perform this task, you need to employ a jQuery utility function called `getJSON()`. We haven't covered it yet (we'll discuss it in section 10.3.2), but what it does is access a resource (URL or file) containing a JSON object (an array or any other type of valid JSON format is allowed) and execute a handler once it has been retrieved. This method passes the JSON that was retrieved, converted into a JavaScript type, to the handler. Inside the latter, you'll do nothing but assign this JavaScript object to your global `movies` variable and fire a custom event, called `moviesLoaded`, to inform your application that you're ready to work.

"Global variable? I thought those were evil," I hear you say. Global variables can be a problem when used incorrectly. In this case, this is truly a global value that represents a page-wide concept, and it will never cause any conflicts because all aspects of the page will want to access this single value in a consistent fashion. Nonetheless, ideally an application should have only one global variable for its application-specific

concerns in a similar fashion to jQuery, where you can find all the methods, utility functions, and properties of the library under the jQuery property. Rather than assigning the data to a global variable called movies, you can create one global variable—for example, dvdApp—retaining the movie data and the application's methods.

In this demo, you'll ignore the risks of using a global variable in order to make everything as simple as possible. The final code that implements what we've described so far is shown here:

```
var movies;
$.getJSON('movies.json', function(data) {
   movies = data;
   $(document).trigger('moviesLoaded');
});

$(document).on('moviesLoaded', function() {
   // Business logic here
});
```

Inside the handler for your custom event, moviesLoaded, you're ready to establish another handler that will be executed when the Add Filter button is clicked. Before writing the necessary code, you need to write the following two statements at the beginning of your code:

```
var $filters = $('#filters');
var templatesAvailable = $('.template', '.templates')
  .not('.filter-chooser')
  .length;
```

The first is needed because you'll use the element having filters as its ID several times. The second is needed because you need to verify that not all the templates defined in the page are already in use. Once you've done this, you're ready to write the body of the callback:

❶ Establishes click handler   **Verifies if the button was pressed before selecting a filter ❷**

```
$('#filter-add')
        .click(function() {
           if ($filters.find('.template:last .filter-type').val() === '') {   ◁
              return;
           }
```

Finds filters already in use ❸
```
           var filterInUse = $filters
                   .children()
                   .map(function() {
                      return $(this)
                              .children('.template')
                              .attr('class')
                              .match(/\b(template-.+?)\b/g)[0];
                   })
                   .get();
```

❹ Tests if all the filters available are in use
```
           if (filterInUse.length === templatesAvailable) {   ◁
              return;
           }
```

Creates filter entry block ❺
```
           var $filterChooser = $('div.template.filter-chooser')
```

```
            .clone()
            .removeClass('filter-chooser')
            .addClass('filter');
    $filterChooser
            .find('option[data-template-type]')
            .filter(function() {
                return filterInUse
                  .indexOf($(this)
                  .data('template-type')) >= 0;
            })
            .remove();
    $filterChooser.appendTo($filters);
})
.click();
```

6 Removes filters already in use

7 Appends the filters drop-down template

8 Triggers the click event

Although this snippet may look complicated at first glance, it accomplishes a great deal without a lot of code. Let's break it down one step at a time.

The first thing that you do in this code is to establish a click handler on the Add Filter button ❶ by using the jQuery `click()` method. It's within the function passed to this method, which will get invoked when the button is clicked, that all the interesting stuff happens.

Before performing any action, you need to verify if the Add Filter button was pressed before the user selected a filter, in which case you need to terminate the function prematurely ❷. If a filter was selected because a click of the Add Filter button is going to, well, add a filter, you need to create a new container for the filter to reside within. As we said, you don't allow multiple filters of the same type, so you have to retrieve all the types of filters in use ❸ to exclude them. To retrieve the filters in use you rely on the class name you used for each template (`template-title`, `template-year`, and so on) and two methods we introduced in the previous chapters of this book: `map()` and `get()`. When finished, you test if all the filters available are in use; if so, prematurely terminate the handler ❹. If not, continue the process.

You clone the template, using the jQuery `clone()` method, that you set up containing the filter drop-down using the replication approach that we discussed in the previous section. Then give it the class `filter` not only for CSS styling but also to be able to locate these elements later in the code ❺. After the element is created, it's time to exclude the filters already in use by using the `filter()` method ❻. Then append the cloned template to the master filter container that you created with the ID value of `filters` ❼.

The previous operation was the last one defined inside the handler. After you attach the latter, but before closing the statement, trigger the click event on the same element using the `click()` method ❽. We discussed this version of the shortcuts in the section "Shortcut methods" of chapter 6. This is a well-known and often used technique to execute a handler you've just attached.

Why do this? Do you remember when you guessed where the markup is for the initial filter drop-down? Yes, you got it! You execute the handler so that as soon as the

page is loaded, the first select element is appended in the panel to allow you to perform the first choice.

Load this page into your browser and test the action of the Add Filter button. Note how every time you click the Add Filter button, a new filter is added to the page. If you inspect the DOM with a JavaScript debugger (Firebug in Firefox and the Chrome Developer Tool are great for this), you'll see how the template has been copied in the container.

In one function (the handler) you've used a lot of the knowledge you've gained so far. This should prove to you, once again, how jQuery allows you to perform complex operations in a few lines of code.

But your job isn't over yet. The drop-downs don't yet specify which field is to be filtered. When the user makes a selection, you need to populate the filter container, adding the appropriate controls for that filter type.

7.1.5 *Adding the controls templates*

Whenever a selection is made from a filter drop-down, you need to populate the filter with the controls that are appropriate for that filter. You've made it easy for yourself by creating markup templates to copy when you determine which one is appropriate. But there are also a few other housekeeping tasks that you need to do whenever the value of the drop-down is changed.

Take a look at what you'll do when establishing the change handler for the drop-down. Remember that the following code, like that of the previous section, is written inside the handler for the custom event called moviesLoaded:

```
$('#filters').on('change', '.filter-type', function() {          Establishes
    var $this = $(this);                                       ❶ change handler
    var $filter = $this.closest('.filter');
    var filterType = $this.find(':selected').data('template-type');

    $('.qualifier', $filter).remove();
    $('div.template.' + filterType)
            .clone()                              Replicates
            .addClass('qualifier')             ❸ appropriate template
            .appendTo($filter);
    $this.find('option[value=""]').remove();      Removes the "Choose
})                                              ❹ a filter" option
```

Removes ❷ any old controls

You take advantage of jQuery's on() method to establish a handler up front that will automatically be established at the appropriate points without further action on your part. This time, you proactively establish a change handler, employing event delegation, for any filter drop-down that comes into being ❶. This is needed because at the time you attach the handler, the filters don't exist inside the div having the ID of filters.

When the change handler fires, cache the jQuery object that contains the current element (this) because you'll use it several times. Then, collect a couple pieces of information: the parent filter container and the filter type recorded in the custom data-template-type attribute.

When you have those values in hand, you need to remove any filter controls that might already be in the container ❷. After all, the user can change the value of the selected field many times, and you don't want to keep adding more and more controls as you go along! You'll add the qualifier class to all the appropriate elements as they're created (in the next statement) so it's easy to select and remove them.

Once you're sure you have a clean slate, replicate the template for the correct set of qualifiers ❸ by using the value you obtained from the data-template-type attribute. The qualifier class name is added to each created element for easy selection (as you saw in the previous statement) and the element is appended to the parent filter container.

Finally, remove the "Choose a filter" <option> from the filter drop-down ❹, because when the user has selected a specific field, it doesn't make any sense to choose that entry again. You *could* just ignore the change event that triggers when the user selects this option, but the best way to prevent a user from doing something that doesn't make sense is to not let them do it in the first place!

Refer again to the example page in your browser. Try adding multiple filters and change their selections. Note how the qualifiers always match the field selection.

Now for those remove buttons...

7.1.6 *Removing unwanted filters and other tasks*

You've given the user the ability to change the field that any filter will be applied to, but you've also given them a remove button (labeled X) that they can use to remove a filter completely.

By this time, you should already have realized that this task will be almost trivial with the tools at your disposal. When the button is clicked, all you need to do is find the closest parent filter container and blow it away! Note that in the source the handler is attached by chaining this code with the previous segment. But for the sake of clarity we'll repeat the selection in the following code:

```
$('#filters').on('click', '.filter-remover', function() {
  $(this).closest('.filter').remove();
});
```

Here you employ event delegation again because at the time the page is loaded there aren't elements having class filter-remover inside the main panel.

Now that all the handlers for the filters have been set, there's only one thing left: applying the filters and showing the results.

7.1.7 *Showing the results*

In the previous section, we said that writing the backend of the application was outside the scope of this chapter, but we didn't want to leave you without a working application. As pointed out previously, you'll emulate a database using a JSON array stored in a file called movies.json bundled with the source. In this section you'll discover the

logic of the function that you need to attach as a handler for the submit event of the form. Let's see what this handler is all about:

```
$('#form-filters').submit(function(event) {
  event.preventDefault();

  var titleCondition = $filters.find('select[name="title-condition"]').val();
  var title = $filters.find('input[name="title"]').val();
  var binderMin = parseInt($filters.find('input[name="binder-min"]').val(), 10);
  var binderMax = parseInt($filters.find('input[name="binder-max"]').val(), 10);
  var yearMin = parseInt($filters.find('input[name="year-min"]').val(), 10);
  var yearMax = parseInt($filters.find('input[name="year-max"]').val(), 10);
  var viewed = $filters.find('input[name="viewed"]:checked').val();

  $('tr:has(td)', '#results').remove();
  results = $.grep(movies, function(element, index) {
    return (
      (
        (
          titleCondition === undefined &&
          title === undefined
        ) ||
        (
          titleCondition === 'contains' &&
          element.title.indexOf(title) >= 0
        ) ||
        (
          titleCondition === 'stars-with' &&
          element.title.indexOf(title) === 0
        ) ||
        (
           titleCondition === 'ends-with' &&
           element.title.indexOf(title) === element.title.length - title.length
        ) ||
        (
          titleCondition === 'equals' &&
          element.title === title
        )
      ) &&
      (isNaN(binderMin) || element.binder >= binderMin) &&
      (isNaN(binderMax) || element.binder <= binderMax) &&
      (isNaN(yearMin) || element.year >= yearMin) &&
      (isNaN(yearMax) || element.year <= yearMax) &&
      (viewed === undefined || element.viewed === (viewed === 'true'))
    );
  });

  var row;
  for(var i = 0; i < results.length; i++) {
    row = '<td>' + results[i].title + '</td>';
    row += '<td>' + results[i].year + '</td>';
    row += '<td>' + results[i].binder + '</td>';
    row += '<td>' + results[i].page + '</td>';
    row += '<td>' + results[i].slot + '</td>';
    row += '<td>' + (results[i].viewed ? 'X' : '') + '</td>';
```

1 Attaches a listener for the submit event

2 Prevents the default behavior

3 Retrieves the value of the filters used

4 Clears previous results but not headers

5 Filters the movies based on the filters

6 Loops over the filtered movies

7 Formats the properties of the movie as cells of a row of a table

```
      $('#results').append(
        $('<tr>').html(row)
      );
    }
});
```

> **Appends the cells to a row created on the fly and then appends the row to the results table** ❽

To start you attach a function as a handler for the `submit` event of the form ❶. Because you don't want your browser to perform an HTTP request (remember, you don't have a backend) inside the handler, you prevent the default behavior ❷. Then execute a set of assignments to retrieve the values of the filters filled by the user ❸. Even if the user won't use all of them, try to retrieve all the possible values. For those that haven't been added, the value will result in an `undefined` value that you'll deal with later.

In the next statement you clear the table from any previous result, keeping in place its headers ❹. Then, using `$.grep()`, create a new array containing only the movies that match the filters filled ❺. We haven't yet covered this method (we'll discuss it in section 9.3.3), but it's similar to the `filter()` method.

As soon as you have the results of the query, you must show them. To achieve this goal, loop over the filtered movies ❻. Inside the loop, format the properties of each movie as cells of a row of the table ❼. The actual row element (`tr`) is created on the fly using the ability of the `jQuery()` method to create elements based on a string. Once it's created, append the row to the table ❽.

Now that you've analyzed the handler, go ahead and click the Apply Filters button. There you go! The set of movies that match the filters has been displayed on the screen. An example of the results shown by this project is illustrated in figure 7.2.

With this last step you've completed the page, at least as far as we wanted to take it for the purposes of this chapter, but as you know…

Figure 7.2 An example of the results returned by the DVD Disc Locator

7.1.8   *There's always room for improvement*

For your filter form to be considered production-quality, there's still lots of room for improvement.

The following list describes some additional functionality either that this form requires before being deemed complete or that would be just plain nice to have. Can you implement these additional features with the knowledge you've gained up to this point?

- Data validation is poor in your form. For example, inside the handler of the form's Submit button you convert strings into numbers where it makes sense, but better controls may be of help, especially for the date fields.

  Ideally, with a backend on the server, you could punt and let the server-side code handle it—after all, it has to validate the data regardless. But that makes for a less-than-pleasant user experience, and as we've already pointed out, the best way to deal with errors is to prevent them from happening in the first place.

  Because the solution involves inspecting the Event instance—something that wasn't included in the example up to this point—we're going to give you the code to disallow the entry of any characters but digits into the numeric fields. The operation of the code should be evident to you with the knowledge you've gained in this chapter, but if not, now would be a good time to go back and review the key points:

  ```
  $('input.numeric').on('keypress', function(event) {
      // Character with code 48 is "0". Character with code 57 is "9".
      if (event.which < 48 || event.which > 57) return false;
  });
  ```

  For browsers that support HTML5, a simpler solution exists. You could force the user to use only numbers by using the new <input> type called number.

- Date fields aren't well validated. How would you go about ensuring that only valid date ranges are entered? What if the user fills a start date greater than the end date? You can't do it on a character-by-character basis as you did with the numeric fields.

- When qualifying fields are added to a filter, the user must click in one of the fields to give it focus. Not all that friendly! Add code to the example to give focus to the new controls as they're added.

- One of the requirements was to not allow a user to have the same filter more than once. In its current version, there's a way to have the same filter twice, which also uncovers a defect of the demo. Can you find out how and fix this behavior?

- Your form allows the user to specify more than one filter but only one per type. How would you change the form to allow the user to specify multiple filters of the same type?

- The types of the filters could be updated when deleting one of the filters in use. If you add all four types of filters (DVD Title, Binder, Release Date, and Viewed?)

and then delete the first (DVD Title), you won't be able to change one of the remaining filters in order to have DVD Title again. This happens because the list is created at the time you click the Add Filter button and is never updated. How can you update the demo to address this issue? (Hint: Listen to the `click` event of the buttons having class `filter-remover`.)

▪ What other improvements would you make, either to the robustness of the code or the usability of the interface? How does jQuery help?

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/derosa, which contains a link to the discussion forum. You're encouraged to post your solutions for all to see and discuss!

7.2 Summary

Using the jQuery knowledge that you've gained so far, in this chapter you developed a fully working web application to manage a DVD collection. One of the lessons that you may have learned from working on it is that tasks that at first glance may seem complex are nothing but a combined set of simple statements. We hope that you had fun developing this small application, which was aimed at reinforcing the concepts we covered up to this point.

In the next chapter, we'll look at how jQuery builds on these capabilities to put animation and animated effects to work for you.

Energizing pages with
animations and effects

This chapter covers

- Showing and hiding elements without animation
- Showing and hiding elements using core animation effects
- Extending the core easing functions
- Writing custom animations
- Controlling animation and function queuing

In the early days of the web, the capabilities afforded to page authors were severely limited, not only by the minimal APIs but also by the sluggishness of scripting engines and low-powered systems. The idea of using these limited abilities for animation and effects was laughable, and for years the only animation was through the use of animated GIF images (which were generally used poorly, making pages more annoying than usable).

Today's browser scripting engines are lightning fast, running on hardware that was unimaginable 10 years ago, and offer a rich variety of capabilities to us as page authors. Even more important, modern browsers have implemented several CSS3

modules with standardized properties that allow us to create amazing animations and effects. Some examples of these properties are `transition`, `transform`, `filter`, `blur`, and `mask`. Unfortunately, there are some issues to keep in mind. The first is that the actual modules implemented depend on the browser, so not all browsers support the same modules. In addition, browsers supporting a given module have implemented it at different times and the span can be considerable. The second point is that older browsers and several mobile browsers don't support these modules and some of them (most notably older ones) will never do so. Therefore, if we want to create an animation that works on all browsers, we have no other choice but to use JavaScript.

But even though JavaScript can help us in achieving this task, it's not easy to create animations using native functions. Fortunately, jQuery comes to our rescue, providing a trivially simple interface for creating 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 movie that's all special effects and no plot, a page that overuses effects can elicit a very different, and negative, reaction than what we intend. Be mindful that effects should be used to enhance the usability of a page, not hinder it by just showing off. Also remember that too many animations can slow down the performance of a website, especially when accessed from a mobile device. With those cautions in mind, let's see what jQuery has to offer.

8.1 Showing and hiding elements

Perhaps the most common type of dynamic effect you'll want to perform on one or more elements is the simple act of showing or hiding them. We'll get to more fancy animations in a bit, but sometimes you just want to keep it simple and pop elements into existence or make them instantly vanish!

The methods for showing and hiding elements are pretty much what you'd expect: `show()`, to show the elements in a jQuery object, 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 methods with no arguments.

As simple as these methods may seem, you should keep a few things in mind. First, jQuery hides elements by changing their `style.display` property to `none`. If an element in the set of matched elements is already hidden, it will remain hidden but will still be returned for chaining. For example, suppose you have the following HTML fragment:

```
<div style="display: none;">This will start hidden</div>
<div>This will start shown</div>
```

If you write

```
$('div').hide().addClass('fun');
```

you'll end up with the following result:

```
<div style="display: none;" class="fun">This will start hidden</div>
<div style="display: none;" class="fun">This will start shown</div>
```

Note that even though the first element was already hidden, it remains part of the matched set and takes part in the remainder of the method chain. This is confirmed by the fact that both elements possess the class `fun` after the statement is executed.

The second point to keep in mind is that jQuery shows objects by changing the `display` property from `none` to either `block` or `inline`. Which of these values is chosen is based on whether a previously specified explicit value was set for the element or not. If the value was explicit, it's remembered and reverted. Otherwise it's based on the default state of the `display` property for the target element type. For example, `div` elements will have their `display` property set to `block`, whereas a `span` element's `display` property will be set to `inline`.

> **NOTE**   Until versions 1.11.0 and 2.1.0 of jQuery, this mechanism had a bug. After the first call to `hide()`, jQuery stored the old value of the `display` property internally in a variable. Then, when the `show()` method was called, jQuery restored this value. Therefore, if after the first call to `hide()`, the `display` property was set to something else, when the `show()` method was executed jQuery restored the old value and not the changed one. You can find more on this issue here: http://bugs.jquery.com/ticket/14750.

Now that you know how these methods behave, let's see how you can work with them.

8.1.1   *Implementing a collapsible "module"*

You're no doubt familiar with websites that present various pieces of information in configurable modules (sometimes referred to as *tiles*) or some sort of dashboard page. This kind of website lets you configure much about how the page is presented, including moving the modules around, expanding them to full-page size, and even removing them completely. Many of them also provide a nice feature: the ability to roll up a module into its caption bar so that it takes up less space, without having to remove it from the page. This seems a perfect example of a functionality that you can replicate by using the knowledge acquired in the previous section. Thus, what you're going to do is create dashboard modules that allow users to roll up a module into its caption bar.

Before delving into writing code, let's see what the module will look like in its normal and rolled-up states. These states are shown in figures 8.1a and 8.1b.

In figure 8.1a, we show a module with two major sections: a caption bar and a body. The body contains the data of the module—in this case, random "Lorem ipsum" text (you can learn more about what this text is here: http://en.wikipedia.org/wiki/Lorem_ipsum). The more interesting caption bar contains a caption for the module and a small button (the minus sign on the top right) that you'll instrument to invoke the roll-up (and roll-down) functionality.

Once the button is clicked, the body of the module will disappear as if it had been rolled up into the caption bar. A subsequent click will roll down the body, restoring its original appearance.

The code to implement this functionality can be found in file chapter-8/collapsible .module.take.1.html and is shown in listing 8.1. In case you surmise that the "take.1" part of this filename indicates that we'll be revisiting this example, you're right!
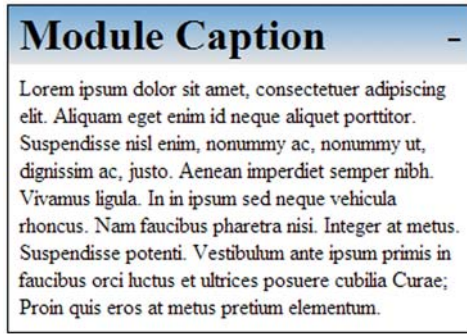
Figure 8.1a You'll create your own dashboard modules, which consist of two parts: a bar with a caption and roll-up button (the minus sign on the top right), and a body in which data can be displayed.

Figure 8.1b When the roll-up button (the minus sign on the top right) is clicked, the module body disappears as if it had been rolled up into the caption bar.

Listing 8.1 The first implementation of our collapsible module

```
<!DOCTYPE html>
<html>
  <head>
    <title>Collapsible Module - Take 1</title>
    <link rel="stylesheet" href="../css/main.css" />
    <link rel="stylesheet" href="../css/module.css" />
  </head>
  <body>
    <div class="module">                                    ❶ The block
      <div class="caption clearfix">                           representing
        <h1>Module Caption</h1>                                the module
        <span class="icon-roll">-</span>                    ❷ The caption of
      </div>                                                    the module
      <div class="body">                                    ❸ The body of
        Lorem ipsum dolor sit amet, consectetuer adipiscing elit.   the module
        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>

    <script src="../js/jquery-1.11.1.min.js"></script>     ❹ Attaches a
    <script>                                                  handler to the
      $('.icon-roll').click(function() {                      <span>
        var $body = $(this).closest('.module').find('.body'); containing the
        if ($body.is(':hidden')) {                            minus "icon"
          $body.show();                                     ❻ Tests the current
        } else {                                               status of the body
                                                               (shown or hidden)
```
❺ Finds the body of the module

❼ Shows the body if it was hidden

```
                $body.hide();                        Hides the body if
            }                                     8  it was shown
        });
    </script>
  </body>
</html>
```

The markup that you'll use to create the structure of your module is fairly straightforward. We've applied to it some class names that serve both for identification as well as for CSS styling. The entire construct is enclosed in a `<div>` having `module` as a class ❶, and it contains the caption ❷ and the body ❸ that are `<div>`s with the classes `caption` and `body` applied, respectively. The caption element has also a class called `clearfix` that's used only for styling purpose, so we're not going to discuss it.

In order to give this module the roll-up behavior, you put inside the caption a `<span>` having class `icon-roll`, containing an "icon" (actually a text with the minus sign). To this element you attach a handler for the click event ❹.

Within the click handler, you first locate the body associated with the module. You need to find the specific instance of the module body because, remember, you may have many modules on your dashboard page, so you can't just select all elements that have the `body` class. You quickly locate the correct body element by finding the closest `module` container, and then starting from it you search for a descendant having class `body` ❺. If how the expression finds the correct element isn't clear to you, now would be a good time to review the information in the early chapters of the book regarding finding and selecting elements.

Once the body is located, you test whether the body is hidden or visible using jQuery's `is()` method ❻. If the body is hidden, you show it using `show()` ❼; otherwise, you hide it using `hide()` ❽. That wasn't difficult at all, was it? But as it turns out, it can be even easier!

8.1.2   *Toggling the display state of elements*

Toggling the display state of elements between revealed and hidden—as you did for the collapsible module example—is such a common occurrence that jQuery defines a method named `toggle()` that makes it even easier.

Let's apply this method to the collapsible module and see how it helps to simplify the code of listing 8.1. The next listing shows only the click handler for the refactored page (no other changes are necessary) with the changes highlighted in bold. The complete page code can be found in the file chapter-8/collapsible.module.take.2 .html.

Listing 8.2   The collapsible module code, simplified with `toggle()`

```
$('.icon-roll').click(function() {
    $(this)
        .closest('.module')
        .find('.body')
        .toggle();
});
```

Note that you no longer need the conditional statement to determine whether to hide or show the module body; `toggle()` takes care of swapping the displayed state on your behalf. This allows you to simplify the code quite a bit and avoid the need to store the body reference in a variable.

The `toggle()` method isn't useful only as a shortcut for alternating calls to `show()` and `hide()`, and in the next few pages you'll discover how this method allows you to do more than that.

Instantaneously making elements appear and disappear is handy, but sometimes you want the transition to be less abrupt. Let's see what's available for that.

8.2 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. If you blink at the wrong moment, you could miss the transition, leaving you to wonder, "What just happened?"

Gradual transitions of a short duration help you know what's changing and how you got from one state to the other—and that's where the jQuery core effects come in. There are three sets of effect types:

- Show and hide (There's a bit more to these methods than what we discussed in section 8.1.)
- Fade in and fade out
- Slide down and slide up

Let's look more closely at each of these effect sets.

8.2.1 Showing and hiding elements gradually

The `show()`, `hide()`, and `toggle()` methods are more flexible than we led you to believe in the previous section. When called with no arguments, these methods effect a simple manipulation of the display state of the DOM elements, causing them to instantaneously be revealed or hidden. But when arguments are passed to them, these effects can be animated so that the changes in display status of the affected elements take place over a period of time.

With that, you're now ready to look at the full syntaxes of these methods.

Method syntax: hide

```
hide(duration[, easing][, callback])
hide(options)
hide()
```

Causes the elements selected 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 `duration` parameter is provided, the elements are hidden over a period of time by adjusting their width, height, and opacity downward to zero, at which time their `display` style property value is set to `none` to remove them from the display.

Method syntax: hide *(continued)*

An optional easing function name can be passed to specify how the transition between the states is performed.
An optional callback can be specified, and is invoked when the animation is complete.
In its second version, you can provide an object containing some options to pass to the method (more on the properties available later).

Parameters

`duration`	(Number|String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined value strings: `"slow"` (same as passing 600), `"normal"` (same as passing 400), or `"fast"` (same as passing 200). If this parameter is omitted and a callback function is specified as the first parameter, the value of `"normal"` is assumed.
`easing`	(String) Specifies an easing function name to use when performing the transition from the visible state to the hidden state. The easing functions specify the pace of the animation at different points while in execution. If an animation takes place but this parameter isn't specified, it defaults to `"swing"`. More about these functions in section 8.3.
`callback`	(Function) A 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. The callback is fired for each element that undergoes animation.
`options`	(Object) An optional set of options to pass to the method. The options available are shown in table 8.1.

Returns

The jQuery collection.

The `hide()` method gives us the opportunity to discuss several points. The first point is the `easing` parameter, which allows you to specify an *easing* function. 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. But what functions are available? The jQuery core supports only two functions: `linear`, which progresses at a constant pace throughout the animation, and `swing`, which progresses slightly slower at the beginning and end of the animation than it does in the middle. "Why only two functions?" you may ask. The reason isn't that jQuery wants to kill your creativity. Rather, jQuery wants to be as lean as possible by externalizing any additional feature to third-party libraries. You'll learn how to add more easing functions (sometimes referred to as *easings*) in section 8.3.

> **NOTE** The value `"normal"` of the `duration` parameter is just a convention. You can use whatever string you like (except `"slow"` and `"fast"`, of course) to specify a transition that has to last 400 milliseconds. For example, you may use `"jQuery"`, `"jQuery in Action"`, or `"wow"`, obtaining exactly the same result. If you want to see it with your own eyes, search for the property `jQuery.fx.speeds` within the jQuery source. That said, we strongly advise you to stick with the usual `"normal"` value to avoid driving your colleagues crazy.

The other parameter worth a discussion is `options`. Using this parameter you can heavily customize how the `hide()` method acts. The properties and the values allowed are shown in table 8.1.

Table 8.1   The properties and the values, in alphabetic order, allowed in the `options` parameter

Property	Value
always	(Function) A function called when the animation completes or stops without completing. The `Promise` object passed to it is either resolved or rejected (we'll discuss this concept in chapter 13).
complete	(Function) A function called when the animation is completed.
done	(Function) A function called when the animation completes, which means when its `Promise` object is resolved.
duration	(String|Number) The duration of the effect as a number of milliseconds or as one of the predefined strings. Same as explained before.
easing	(String) The function name to use when performing the transition from the visible state to the hidden state. Same as explained before.
fail	(Function) A function invoked when the animation fails to complete. The `Promise` object passed to it is rejected.
progress	(Function) A function executed after each step of the animation. The function is called only once per animated element regardless of the number of animated properties.
queue	(Boolean|String) A Boolean specifying whether the animation has to be placed in the effects queue (more on this in a later section). If the value passed is `false`, the animation will begin immediately. The default value is `true`. In case a string is passed, the animation is added to the queue represented by that string. When a custom queue name is used, the animation does not automatically start.
specialEasing	(Object) A map of one or more CSS properties whose values are easing functions.
start	(Function) A function invoked when the animation begins.
step	(Function) A function executed for each animated property of each animated element.

All these properties enable you to create amazing effects. We'll cover this topic shortly, developing three different custom effects. We know there are a lot of unclear concepts and you might be a bit confused. But don't worry; everything will be explained in detail shortly, so please bear with us.

Now that you've studied in depth the `hide()` method and its parameters, you can learn more about `show()`. Because the meaning of the parameters is the same for `show()` as for `hide()`, we won't repeat their description.

<div style="background:#f5f0e0;">

Method syntax: show

</div>

```
show(duration[, easing][, callback])
show(options)
show()
```

Causes any hidden elements in the set of matched elements to be revealed. If called with no parameters, the operation takes place instantaneously by setting the `display` style property value of the elements to an appropriate setting (`block` or `inline`).

If a `duration` parameter is provided, the elements are revealed over a specified duration by adjusting their width, height, and opacity upward to full size and opacity.

An optional easing function name can be specified to define how the transition between the states is performed.

An optional callback can be specified that's invoked when the animation is complete.

In its second version, you can provide an object containing some options to pass to the method, as described in table 8.1.

Returns

The jQuery collection.

As you saw in our second example, jQuery provides a shortcut called `toggle()` to toggle the state of one or more elements. Its syntax is shown here, and in this case too we'll omit the description of the parameters already described.

<div style="background:#f5f0e0;">

Method syntax: toggle

</div>

```
toggle(duration[, easing][, callback])
toggle(options)
toggle(condition)
toggle()
```

Performs `show()` on any hidden element and `hide()` on any non-hidden element. See the syntax description of those methods for their semantics.

In its third form, `toggle()` shows or hides the selected elements based on the evaluation of the passed condition. If `true`, the elements are shown; otherwise, they're hidden.

Parameters

condition	(Boolean) Determines whether elements must be shown (if `true`) or hidden (if `false`).

Returns

The jQuery collection.

Let's do a third take on the collapsible module, animating the opening and closing of the sections. Given the previous information about the `toggle()` method, you'd think that the only change you'd need to make to the code in listing 8.2 would be to change the call to `toggle()` to `toggle('slow')`. And you'd be right. But not so fast! Because that was just too easy, let's take the opportunity to add an additional feature to the module.

Let's say that to give the user an unmistakable visual clue, you want the module's caption to display a different icon when it's in its rolled-up state. You could make the change before firing off the animation, but it'd be much cooler to wait until the animation is finished.

jQuery 3: Feature changed

jQuery 3 changes the behavior of `hide()`, `show()`, `toggle()`, `fadeIn()`, and all the other related methods that we'll cover in this chapter. All these methods will no longer override the CSS cascade. What this means is that if an element is hidden—because in your style sheet you have a declaration of `display: none;`—invoking `show()` (or similar methods like `fadeIn()` and `slideDown()`) on that element will no longer show it. To understand this change, consider the following element:

```
<div class="hidden">Hello!</div>
```

Now, let's assume that in your style sheet you have the following code:

```
.hidden { display: none; }
```

If you're using a version of jQuery prior to 3, if you write

```
$('div').show('slow');
```

you'll see a nice animation that ultimately will show the element.

In jQuery 3, executing the same statement will have no effect as the library doesn't override the CSS declaration. If you want to obtain the same result in jQuery 3, you should use a statement like the following instead:

```
$('div')
    .removeClass('hidden')
    .hide()
    .show('slow');
```

Alternatively, you could use this statement:

```
$('div')
    .removeClass('hidden')
    .css('display', 'none')
    .show('slow');
```

This change is controversial and will probably break the code of many websites. At the time of writing, the final version of jQuery 3 has not been published, so this new behavior could be modified or completely reverted. Please take the time to check the official documentation to learn more.

You can't just make the call right after the animation method call because animations don't block. The statements following the animated method call would execute immediately, probably even before the animation has had a chance to commence. This is a great occasion to use the callback parameter of `toggle()`.

The approach you'll take is that after the animation is complete, you'll replace the text of the `<span>` containing the icon. You'll use the minus sign if the body is visible (to indicate that it can be collapsed) and the plus sign if the body is hidden (to indicate that it can be expanded). You could also have done this by working with CSS and the `content` property, by adding a class name to the module to indicate that it's rolled up and removing the class name otherwise. But because this isn't a book on CSS, we'll skip this solution. The next listing shows the change that you need to make to your code to make it happen.

Listing 8.3   Animated version of the module, with a change of the caption icon

```
$('.icon-roll').click(function() {
    var $icon = $(this);
    $icon
        .closest('.module')                                    Changes the text of
        .find('.body')                                         the icon based on
        .toggle('slow', function() {                           the state of the
            $icon.text($(this).is(':hidden') ? '+' : '-');  ◁— body of the module
        });
});
```

You can find the page with these changes in file chapter-8/collapsible.module.take .3.html.

Knowing how much people love to tinker, we've set up a handy tool that we'll use to further examine the operation of these and the remaining effect methods.

8.2.2   *Introducing the jQuery Effects Lab Page*

Back in chapter 2, we introduced the concept of lab pages to help you experiment with using jQuery selectors. For this chapter, we've set up a jQuery Effects Lab Page for exploring the operation of the jQuery effects in the file chapter-8/lab.effects.html. Loading this page into your browser results in the display shown in figure 8.2.
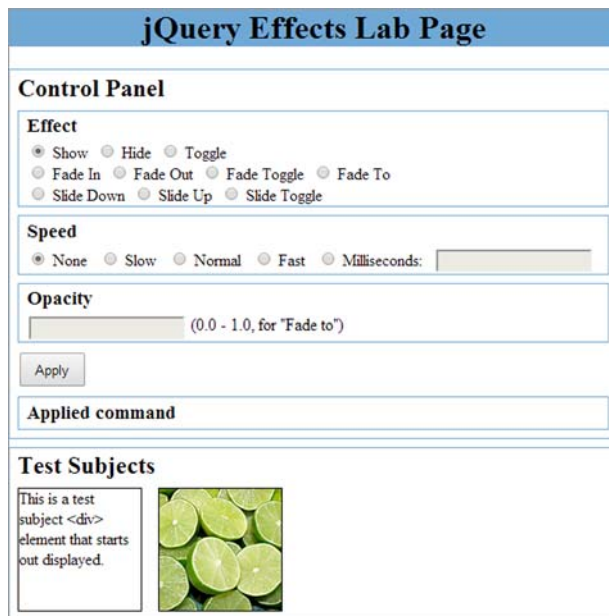
Figure 8.2   The initial state of the jQuery Effects Lab Page, which will help you examine the operation of the jQuery effects methods

This lab consists of two main panels: a control panel in which you'll specify which effect will be applied and another panel 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."

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 methods we've discussed to this point. Load the page in your browser, and follow along with the ensuing experiments:

- *Experiment 1*—With the controls left as-is after the initial page load, click the Apply button. This will execute a show() method with no parameters. The expression that was applied is displayed below the Apply button for your information. Note that the two initially hidden test subject elements appear instantly. If you're wondering why the belt image on the far right appears a bit faded, it's because its opacity has been purposefully set to 50% (the actual value in CSS is 0.5).

- *Experiment 2*—Select the Hide radio button and click Apply to execute the hide() method without arguments passed to it. All of the test subjects immediately vanish. Take special notice that the pane in which they resided has tightened up. This indicates that the elements have been completely removed from the display rather than merely 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 by setting its CSS display style property to none. It doesn't 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.

- *Experiment 3*—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.

- *Experiment 4*—Reload the page to reset everything to the initial conditions (in Firefox set focus to the address bar and press the Enter key—simply clicking the reload button won't reset the form elements). Select Toggle and click Apply. Note that the two initially visible subjects vanish and the two that were hidden appear. This demonstrates that the toggle() method applies individually to each element, revealing the ones that are hidden and hiding those that aren't.

- *Experiment 5*—In this experiment, we'll move into the realm of animation. Reload the page, leave Show selected, and select Slow for the Speed setting. Click Apply, and carefully watch the test subjects. The two hidden elements, rather than popping into existence, gradually grow from their top-left corner. If you want to really see what's going on, reload the page again, select Milliseconds for the Speed setting and enter 5000 for the value. This will extend the duration of the effect to five (excruciating) seconds and give you plenty of time to observe the behavior of the effect.

- *Experiment 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.

8.2.3   *Fading elements into and out of existence*

If you watched the operation of the show() and hide() effects carefully, you will have 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(), affect only the opacity of the elements; once they reach 0 (totally transparent) or 1 (totally visible) depending on the method called, they set the display property to either none or whatever it was (we covered this mechanism in section 8.1).

Other than the lack of scaling, these methods work in a fashion similar to the animated forms of show() and hide(). The syntaxes of these methods are similar to show() and hide() and the meaning of the parameters is the same, so we won't repeat the parameters. The only difference between fadeIn(), fadeOut(), and all the other animation-related methods and the previously described methods (show(), hide(), and toggle()) is that because the former when called without parameters perform a transition, the change from one state to another doesn't happen immediately.

With this in mind, their syntaxes are as follows.

Method syntax: fadeIn

fadeIn(duration[, easing][, callback])
fadeIn(options)
fadeIn()

Causes the 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 1 (totally visible). The duration of the change in opacity is determined by the duration parameter. If the duration parameter is omitted, the default is 400 milliseconds ("normal"). Only hidden elements are affected.

Returns

The jQuery collection.

Method syntax: fadeOut

fadeOut(duration[, easing][, callback])
fadeOut(options)
fadeOut()

Causes the matched elements that aren't hidden to be removed from the display by gradually changing their opacity to 0. The duration of the change in opacity is determined by the duration parameter. If the duration parameter is omitted, the default is 400 milliseconds ("normal"). Only displayed elements are affected. Once opacity has been reduced to 0, the element is removed from the display.

Returns

The jQuery collection.

In the same way a convenient method named `toggle()` is used to `hide()` and `show()` elements based on their current state, `fadeIn()` and `fadeOut()` have `fadeToggle()`. The syntax of this method is the following.

Method syntax: fadeToggle

```
fadeToggle(duration[, easing][, callback])
fadeToggle(options)
fadeToggle()
```
Performs `fadeOut()` on any non-hidden elements and `fadeIn()` on any hidden elements. See the syntax description of those methods for their semantics.

Returns

The jQuery collection.

Let's have some more fun with the jQuery Effects Lab Page. Load the lab page, and run through a set of experiments similar to those in the previous section but using the Fade In, Fade Out, and Fade Toggle 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()`, `toggle()`, `fadeIn()`, `fadeOut()`, and `fadeToggle()` effects remember the original opacity of an element and honor its value. In the lab, we purposely set the initial opacity of the belt image at the far right to 50% before hiding it. Throughout all the opacity changes that take place when applying the jQuery effects, this original value is never stomped on.

Another effect that jQuery provides is via the `fadeTo()` method. This effect adjusts the opacity of the elements like the previously examined fade effects, but it never removes the elements from the display. Before you start playing with `fadeTo()` in the lab, here's its syntax (we'll describe only the new parameters).

Method syntax: fadeTo

```
fadeTo(duration, opacity[, easing][, callback])
```
Gradually adjusts the opacity of the elements in the jQuery object from their current settings to the new setting specified by `opacity`

Parameters

`opacity`   (Number) The target opacity to which the elements will be adjusted, specified as a value from `0` to `1`

Returns

The jQuery collection

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 experiments:

- *Experiment 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 Opacity field (which expects a value between 0 and 1) to 0.1 and click Apply. The test subjects will fade to 0.1 opacity over the course of 4 seconds.
- *Experiment 2*—Set the opacity to 1 and click Apply. All elements, including the initially semitransparent belt image, are adjusted to full opaqueness.
- *Experiment 3*—Set the opacity to 0 and click Apply. All elements fade away to invisibility, but note that once they've vanished, the enclosing module doesn't tighten up. Unlike the fadeOut() effect, fadeTo() never removes the elements from the display, even when they're fully invisible.

Continue experimenting with the Fade To effect until you've mastered its behavior. Then you'll be ready to move on to the next set of effects.

8.2.4 *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, and without parameters they are animated.

Like the previous set of effects, the slide effects have a related method that will toggle the elements between hidden and revealed: slideToggle(). The by now familiar syntaxes for these methods follow.

Method syntax: slideDown

```
slideDown(duration[, easing][, callback])
slideDown(options)
slideDown()
```
Causes any matched elements that are hidden to be shown by gradually increasing their height. Only hidden elements are affected.

Returns

The jQuery collection.

Method syntax: slideUp

```
slideUp(duration[, easing][, callback])
slideUp(options)
slideUp()
```
Causes any matched elements that are displayed to be removed from the display by gradually decreasing their height.

Returns

The jQuery collection.

Method syntax: slideToggle

`slideToggle(duration[, easing][, callback])`
`slideToggle(options)`
`slideToggle()`

Performs `slideDown()` on any hidden elements and `slideUp()` on any displayed elements. See the syntax description of those methods for their semantics.

Returns

The jQuery collection.

Except for the manner in which the elements are revealed and hidden, these effects act similarly to the other show and hide effects. Convince yourself of this by displaying the jQuery Effects Lab Page and running through experiments like those you applied using the other effects.

8.2.5　*Stopping animations*

You may have a reason now and then to stop an animation once it has started. This could be because a user event dictates that something else should occur or because you want to start a completely new animation. The `stop()` method will achieve this for you.

Method syntax: stop

`stop([queue][, clearQueue[, goToEnd]])`
Stops any animation that's currently in progress for the elements in the jQuery object.

Parameters

queue	(String) The name of the queue in which to stop animations (we'll get to that shortly).
clearQueue	(Boolean) If specified and set to `true`, stops not only the current animation but any other animations waiting in the animation queue. The default value is `false`.
goToEnd	(Boolean) If specified and set to `true`, completes the current animation immediately (as opposed to merely stopping it). The default value is `false`.

Returns

The jQuery collection.

When using the `stop()` method, keep in mind that any change that has already taken place for any animated elements will remain in effect. In addition, if you call `stop()` on a set when jQuery is performing an animation like `slideUp()` and the animation isn't completed, a portion of the elements will still be visible on the page. If you want to restore the elements to their original states, it's your responsibility to change the CSS values back to their starting values using jQuery's `css()` method or a similar method.

You can avoid having elements with only part of the animation completed by passing `true` as the value of the `goToEnd` parameter. If you want to remove all the animations in the queue and set the CSS properties as if the *current* animation was completed, you have to call `stop(true, true)`. By specifying the current animation, we mean that if you've chained three animations and while the first is still running you call `stop(true, true)`, the style of the elements will be set as if the first animation was completed and the other two were never run (they're removed from the queue before being performed).

In some cases, when stopping an animation you also want to set the CSS properties as if all the animations were completed. In such situations you can use `finish()`.

<div style="text-align:center">**Method syntax: finish**</div>

`finish([queue])`

Stops the animation that's currently in progress for the elements in the jQuery object, removes all the animations in the queue (if any), and immediately sets the CSS properties to their target values.

Parameters

queue   (String) The name of the queue in which to stop animations. If not specified, the
                      `fx` queue is assumed, which is the default queue used by jQuery.

Returns

The jQuery collection.

To help you visualize the main difference between `stop()` and `finish()`, we've created a demo that you can find in the file chapter-8/stop.vs.finish.html and that's also available as a JS Bin (http://jsbin.com/taseg/edit?html,js,output).

In addition to these two methods, there's also a global flag called `jQuery.fx.off` that you can use to completely disable all animations. Setting this flag to `true` will cause all effects to take place immediately without animation. Another jQuery flag that deals with animations is `jQuery.fx.interval`. We'll cover these flags and why you'd want to use them formally in chapter 9 when we also discuss the other jQuery flags provided by jQuery.

Now that you've seen the effects built into the core of jQuery, let's investigate writing your own!

8.3   *Adding more easing functions to jQuery*

In the previous section you learned about the `easing` parameter and the easing functions available in jQuery: `linear` and `swing`. The number of core effects supplied with jQuery is purposely kept small, in order to keep jQuery's core footprint to a minimum. But this doesn't mean you can't employ third-party libraries to gain access to more easings (once again you need to remember that easing functions are often referred as easings). The jQuery Easing plugin (https://github.com/gdsmith/jquery.easing) and the jQuery UI library (http://jqueryui.com) provide additional transitions and effects. The term *plugin* should be familiar to you, but if it isn't, a

plugin is a component that adds a specific feature to an existing software, framework, or library. We'll cover plugins and how to create your own jQuery plugin extensively in chapter 12.

> **NOTE** The jQuery UI is a curated set of user interface interactions, effects, widgets, and themes built on top of jQuery. It's an amazing library and you should really consider taking a look at it (after reading this book, of course). Apart from the official website, an excellent resource is *jQuery UI in Action* by T.J. VanToll (Manning, 2014). A good read!

When there's more than one alternative, people usually start asking what the best solution is. As often happens, there's no best solution in general, but the choice should be based on the specific use case. To help you choose what library to adopt, we can say that you should use jQuery UI to add the effects only if you're already using the library in your project for other reasons (for example, the widgets it provides); otherwise, you should stick with the plugin. The reason is that both offer the same easing functions, but the jQuery Easing plugin is more lightweight (only 3.7 KB in its compressed version) because it's focused on one feature, whereas the jQuery UI has more than just easings. For the sake of completeness, we should also mention that the download page of the jQuery UI offers the possibility of customizing the build of the library, including only the modules you need, which leads to a reduction of the weight.

Both the jQuery Easing plugin and the jQuery UI library add 30 (yes, 30; you read it right) new easing functions, listed in table 8.2.

Table 8.2   The easing functions added by both the jQuery Easing plugin and the jQuery UIEasing functions

easeInQuad	easeOutQuint	easeInOutCirc
easeOutQuad	easeInOutQuint	easeInElastic
easeInOutQuad	easeInExpo	easeOutElastic
easeInCubic	easeOutExpo	easeInOutElastic
easeOutCubic	easeInOutExpo	easeInBack
easeInOutCubic	easeInSine	easeOutBack
easeInQuart	easeOutSine	easeInOutBack
easeOutQuart	easeInOutSine	easeInBounce
easeInOutQuart	easeInCirc	easeOutBounce
easeInQuint	easeOutCirc	easeInOutBounce

The jQuery Easing plugin has a peculiarity worth discussing. It stores jQuery's core `swing` easing, which is the default, under the name `jswing`. In addition, it redefines `swing` to behave in the same way as the `easeOutQuad` easing. Because of this change, the latter becomes the default easing.

To use the jQuery Easing plugin in your pages., you have two possibilities. The first and easier one is to include it using a CDN. Most of you will remember that we introduced CDNs in chapter 1 of this book when discussing those to include jQuery.

The second method is to download the plugin from the repository and save it in a folder that your pages can access. For example, if you save the library in a folder called js, you can include it by writing

```
<script src="js/jquery.easing.min.js"></script>
```

To add the jQuery UI to your pages, you can employ the same methods: CDN or hosting it locally. To include the library via the jQuery CDN, assuming you want to use version 1.11.4, you have to write

```
<script src="http://code.jquery.com/ui/1.11.4/jquery-ui.min.js"></script>
```

If you host it locally instead, assuming that you saved the library in a folder called js, you have to write

```
<script src="js/jquery-ui-1.11.4.min.js"></script>
```

Now that you know how to include the jQuery Easing plugin and jQuery UI in your pages, you can employ the additional easing functions provided. Using them is simple because all you have to do is pass the name of the transition you want to use as the `easing` parameter.

To get a quick glimpse of the evolution in the time of the easing functions listed in table 8.2, take a look at http://api.jqueryui.com/easings/. But taking a quick look at the easing functions isn't enough in our opinion; you deserve more. For this reason, we created a new lab page to allow you to see these effects applied to the methods covered in this chapter.

This new lab page, shown in figure 8.3, is called jQuery Advanced Effects Lab Page and can be found in the file chapter-8/lab.advanced.effects.html of the source code provided with the book.

Play with this new lab page until you have a clear idea of how different easing functions change the way the elements of a page can be animated at different paces.

Up to this point you've seen only the precreated animations at your disposal, but sometimes you'll want to create your own animations. Let's see how you can do this.

8.4 *Creating custom animations*

In the previous section you saw how to easily integrate new easing functions by using third-party libraries. Creating your own animations is also a surprisingly simple matter.

jQuery exposes the `animate()` method, which allows you to apply your own custom animated effects to a set of elements. Let's take a look at its syntax, including the parameters' descriptions. It's been a few pages since we reminded you of the meaning of the parameters, so even if they have the same meaning, we'll repeat their description here. This way you don't have to turn back a lot of pages.

Figure 8.3 The initial state of the jQuery Advanced Effects Lab Page

Select which easing function to use.

Method syntax: animate

`animate(properties[[, duration][, easing]][, callback])`
`animate(properties[, options])`

Animate the properties specified by `properties` of all the elements in the jQuery collection. An optional duration, easing function, and callback function can be specified. The callback function is invoked when the animation is complete. An alternative format specifies a set of `options` in addition to the `properties`.

Parameters

`properties` (Object) An object of CSS properties and values that the animation will move toward. 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. When specifying multiword properties you can write them either using camel case notation (i.e., `backgroundColor`) or quoting the name without the camel case notation (`'background-color'`).

`duration` (Number|String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined value strings: `"slow"` (same as passing 600), `"normal"` (same as passing 400), or `"fast"` (same as passing 200). If this parameter is omitted and a callback function is specified as the first parameter, the speed `"normal"` is assumed.

`easing`	(String) Specifies an optional easing function name to use when performing the transition. These functions specify the pace of the animation at different points while in execution. If an animation takes place but this parameter isn't specified, it defaults to `"swing"`. More about these functions in section 8.3.
`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. The callback is fired for each element that undergoes animation.
`options`	(Object) An optional set of options to pass to the method. The options available are shown in table 8.1.

Returns

The jQuery collection.

You can create custom animations by supplying a set of CSS style properties and target values that those properties will converge toward 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 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.

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's a logical progression from a start value to a target value. This numeric restriction is completely understandable—how would you envision the logical progress from a source value to an end value for a non-numeric property such as `background-image`? For values that represent dimensions, jQuery assumes the default unit of pixels, but you can also specify em units or percentages by including the `em` or `%` suffixes.

> **NOTE**   In CSS it's possible to animate the `color` property of an element, but you can't achieve this effect using jQuery's `animate()` method unless you employ the jQuery.Color plugin (https://github.com/jquery/jquery-color).

Frequently animated style properties include `top`, `left`, `width`, `height`, and `opacity`. But if it makes sense for the effect you want to achieve, you can also animate numeric style properties such as `font-size`, `margin`, `padding`, and `border`.

In addition to specific values for the target properties, you 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()` methods), and it should be noted that `"toggle"` remembers the initial state so that it can be restored on a subsequent `"toggle"`.

Before moving ahead, we thought we should give you a better idea of how the easing functions can change an animation. For this reason, we created a page that you can find in the file chapter-8/easings.html that shows how the `animate()` method can be used with the jQuery Easing plugin.

Figure 8.4   The initial state of the easings.html page

With this page you can move an image from the left to the right according to a given easing function. The initial state of the page is shown in figure 8.4.

Play with this page until you are sure you understand how the easing functions work. In case you want to test more easings, you don't have to reload the page. Once an animation is completed, clicking the Apply button will reset the position of the image and start the animation selected.

Now try your hand at writing a few more custom animations.

8.4.1   A custom scale animation

Consider a simple scale animation in which you want to adjust the size of the elements to twice their original dimensions. Such an animation is shown in listing 8.4.

> **NOTE**   This specific animation can be easily performed via CSS in modern browsers using `transform: scale(2)`. As pointed out in the introduction to this chapter, modern browsers support a lot of new standards, but you should be aware of the issues that come from targeting only modern browsers.

Listing 8.4   A custom scale animation

```
$('.animated-elements').each(function() {      ①  Iterates over each matched element
    var $this = $(this);
    $this.animate({
        width: $this.width() * 2,               ②  Specifies individual
        height: $this.height() * 2                  target values
    },
    2000                                        ③  Sets the duration in milliseconds
    );
});
```

To implement this animation, you iterate over all the elements having `animated-elements` as their class, via jQuery's `each()` method ①. By doing so, you can apply the animation individually to each element. This is important because the property values that you need to specify for each element are based on the individual dimensions for that element ②. If you knew that you'd be animating a single element (such as if you

were using an ID selector) or applying the exact same set of values to each element, you could dispense with each() and animate the set directly.

Within the callback passed to each(), the animate() method is applied to each element, one at a time. You access the element by using this and set the style property values for width and height to double the element's original dimensions. The result is that over the course of 2 seconds (as specified by the duration parameter set to 2000 ❸), each element in the jQuery object will grow from its original size to twice that size.

We'll provide you with a demo to play with the code created in this section, but for the moment let's try something a bit more extravagant.

8.4.2 *A custom drop animation*

Let's say that you want to conspicuously animate the removal of an element from the display. The animation you'll use to accomplish this will make the element appear to drop off the page, disappearing from the display as it does so.

If you think about it for a moment, you can figure out that by adjusting the top position of the element, you 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, you can remove the element from the display. You can accomplish this drop effect with the code in the following listing.

Listing 8.5   A custom drop animation

```
$('.animated-elements').each(function() {          ◁────  Selects all elements having class
   var $this = $(this);                                   animated-elements
   $this
       .css('position', 'relative')    ◁───❶  Dislodges element from static flow
       .animate({
          opacity: 0,
          top: $(window).height() - $this.height() -      ❷  Computes drop
              $this.position().top                            distance
       },
       'normal',
       function() {                          ◁────         Executes function when
          $this.hide();     ◁────           ❸  animation is completed
       }
   );                              Removes element
});                          ❹  from display
```

There's a bit more going on here than in the previous custom effect. You once again iterate over the elements in the set, but this time adjusting the position *and* the opacity of the elements. To adjust the top value of an element relative to its original position, you first need to change its CSS position style property value to relative ❶.

Then you specify a target opacity of 0 and a computed top value. You 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. You don't want to draw their attention away from the animation—grabbing their attention is why you're animating in the first place! 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 ❷. Of course, this consideration makes sense only if the space between the elements and the bottom of the page is large enough.

When the animation is completed, you want to remove the element from the display, so specify a callback function ❸ that applies the non-animated `hide()` method to the element ❹.

> **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 completed 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.

8.4.3 *A custom puff animation*

Rather than dropping elements off the page, let's say you want an effect that makes it appear as if the element dissipates into thin air like a puff of smoke. To animate such an effect, you can combine a scale effect with an opacity effect, growing the element while fading it away. One issue you need to deal with for this effect is that this dissipation would not fool the eye if you let the element grow in place with its upper-left corner anchored. You want the *center* of the element to stay in the same place as it grows. Hence, in addition to its size, you need to adjust the position of the element as part of the animation. The code for the puff effect is shown in the next listing.

Listing 8.6   A custom puff animation

```
$('.animated-elements').each(function() {        ◁——   Selects all elements having
  var $this = $(this);                                  class animated-elements and
  var position = $this.position();                  ❶  iterates over them
  $this
      .css({
        position: 'absolute',                                Dislodges
        top: position.top,                              ❷   element from
        left: position.left                                  static flow
      })
      .animate({
        opacity: 'hide',                                    Adjusts
        width: $this.width() * 5,                           element size,
        height: $this.height() * 5,                     ❸  position, and
        top: position.top - ($this.height() * 5 / 2),      opacity
        left: position.left - ($this.width() * 5 / 2)
      },
      'fast'
    );
});
```

In this animation, you select all the elements having a class of `animated-elements` and iterate over them ❶. Then you 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 center of the element remaining in the same position ❸. You don't want the elements surrounding the animated element to be pushed out while the target element is growing, so you take it out of the layout flow completely by changing its position to `absolute` and explicitly setting its position coordinates ❷. Because you specified `hide` for the `opacity` value, the elements are automatically hidden (removed from the display) once the animation is completed.

Each of these three custom effects can be observed by loading the page found in the file chapter-8/custom.effects.html, whose display is shown in figure 8.5.
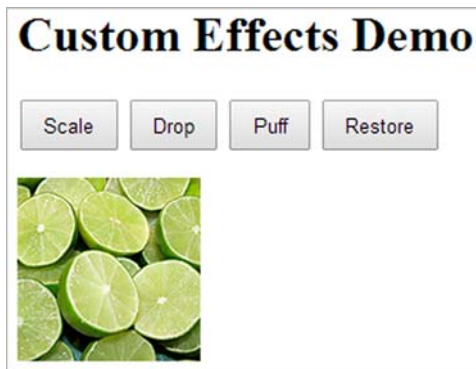
Figure 8.5   The custom effects you developed (scale, drop, and puff) can be observed in action using the buttons provided on this example page.

Although we'd love to show you how these effects behave, screenshots have obvious limitations, which we're sure you understand. But figure 8.6 shows a frame of the puff effect in progress. We'll leave it to you to try out the various effects on this page and observe their behavior.

Up to this point, all of the examples we've examined have used a single animation method. Let's discuss how things work when you use more than one.

Figure 8.6   The puff effect expands and moves the image while simultaneously reducing its opacity.

8.5 *Animations and queuing*

You've seen how multiple properties of elements can be animated using a single animation method, but we haven't examined how animations behave when you call simultaneous animation methods. In this section we'll examine how animations behave in concert with each other.

8.5.1 *Simultaneous animations*

What would you expect to happen if you were to execute the following code?

```
$('#test-subject').animate({left: '+=256'}, 'slow');
$('#test-subject').animate({top: '+=256'}, 'slow');
```

You know that the `animate()` method doesn't block while its animation is running on the page, nor do any of the other animation methods. You can prove that by experimenting with this code block:

```
console.log(1);
$('#test-subject').animate({left: '+=256'}, 'slow');
console.log(2);
```

If you were to execute this code, you'd see that the messages "1" and "2" are printed immediately on the console, one after the other, without waiting for the animation to complete. If you want to prove that this is true, take a look at the file chapter-8/asynchronous.animate.html or the JS Bin we've created for you (http://jsbin.com/pulik/edit?html,js,console,output).

What would you expect to happen when you run the code with two animation method calls (the first snippet of this section)? Because the second method isn't blocked by the first, it stands to reason that both animations fire off simultaneously (or within a few milliseconds of each other) and that the effect on the element animated would be the combination of the two effects. In this case, because one effect is adjusting the `left` style property and the other the `top` style property, you might expect that the result would be a meandering diagonal movement of the element.

Let's put that to the test. In the file chapter-8/revolutions.html we've put together an experiment that sets up two images (one of which is to be animated) and a button to start the experiment. In addition, you'll use the console to write some output. Figure 8.7 shows its initial state.
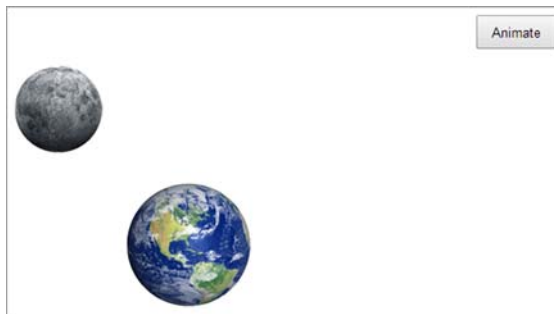
Figure 8.7 Initial state of the page where you'll observe the behavior of multiple, simultaneous animations

The Animate button is instrumented as shown in the following listing.

Listing 8.7   Instrumentation for multiple simultaneous animations

```
function formatDate(date) {
   return (date.getHours() < 10 ? '0' : '') + date.getHours() +
   ':' + (date.getMinutes() < 10 ? '0' : '') + date.getMinutes() +
   ':' + (date.getSeconds() < 10 ? '0' : '') + date.getSeconds() +
   '.' + (date.getMilliseconds() < 10 ?
        '00' : (date.getMilliseconds() < 100 ? '0' : '')) +
   date.getMilliseconds();
}

$('#button-animate').click(function() {            ← Defines the
   var $moonImage = $('img[alt="moon"]');            click handler
   console.log('At ' + formatDate(new Date()) + ' 1');  ❶ for the button

   $moonImage.animate({left: '+=256'}, 2500);
   console.log('At ' + formatDate(new Date()) + ' 2');

   $moonImage.animate({top: '+=256'}, 2500);
   console.log('At ' + formatDate(new Date()) + ' 3');

   $moonImage.animate({left: '-=256'}, 2500);
   console.log('At ' + formatDate(new Date()) + ' 4');

   $moonImage.animate({top: '-=256'}, 2500);
   console.log('At ' + formatDate(new Date()) + ' 5');
});
```

In the click handler for the button ❶, you fire off four animations, one after the other, interspersed with calls to `console.log()` that show you when the animation calls were fired off.

Bring up the page, and click the Animate button. As expected, the console messages "1" through "5" immediately appear on the console, as shown in figure 8.8, each firing off a few milliseconds after the previous one.

But what about the animations? If you examine the code in listing 8.7, you can see that you have two animations changing the `top` property and two animations changing the `left` property. In fact, the animations for each property are doing the exact opposite of each other. What should you expect? Might they just cancel each other out, leaving the moon (our test subject) to remain completely still? No. You see that each animation happens serially, one after the other, such that the moon makes a complete and orderly revolution around the Earth (albeit in a very unnatural square orbit that would have made Kepler's head explode).

What's going on? You've proven via the console messages that the animations aren't blocking, yet they execute serially just as if

```
At 03:36:19.972 1     revolutions.html:56
At 03:36:19.979 2     revolutions.html:58
At 03:36:19.980 3     revolutions.html:60
At 03:36:19.980 4     revolutions.html:62
At 03:36:19.980 5     revolutions.html:64
```

Figure 8.8   The console messages appear in rapid succession, proving that the animation methods aren't blocking until completion.

they were (at least with respect to each other). This happens because internally jQuery is queuing up the animations and executing them serially on your behalf.

Refresh the page to clear the console, and click the Animate button three times in succession. (Pause between clicks just long enough to avoid double-clicks.) You'll note that 15 messages get immediately sent to the console, indicating that your click handler has executed three times, and then sit back as the moon makes three orbits around the Earth. Each of the 12 animations is queued up by jQuery and executed in order because the library maintains a queue on each animated element named `fx` just for this purpose.

What's even better is that jQuery makes it possible for you to create your own execution queues, not just for animations but for whatever purposes you want. Let's learn about that.

8.5.2 *Queuing functions for execution*

Queuing up animations for serial execution is an obvious use for function queues. But is there a real benefit? After all, the animation methods allow for completion callbacks, so why not just fire off the next animation in the callback of the previous animation?

ADDING FUNCTIONS TO A QUEUE

Let's review the code fragment of listing 8.7 (minus the `console.log()` invocations for clarity):

```
var $moonImage = $('img[alt="moon"]');
$moonImage.animate({left: '+=256'}, 2500);
$moonImage.animate({top: '+=256'}, 2500);
$moonImage.animate({left: '-=256'}, 2500);
$moonImage.animate({top: '-=256'}, 2500);
```

Compare that to the equivalent code that would be necessary without function queuing, using the completion callbacks:

```
var $moonImage = $('img[alt="moon"]');
$moonImage.animate({left: '+=256'}, 2500, function(){
   $moonImage.animate({top: '+=256'}, 2500, function(){
      $moonImage.animate({left: '-=256'}, 2500, function(){
         $moonImage.animate({top: '-=256'}, 2500);
      });
   });
});
```

It's not that the callback variant of the code is that much more complicated, but it'd be hard to argue that the original code isn't a lot easier to read (and to write in the first place). And if the bodies of the callback functions were to get substantially more complicated…. Well, it's easy to see how being able to queue up the animations makes the code a lot less complex.

Queues can be created on any element, and distinct queues can be created by using unique names for them (except for `fx`, which is reserved for the effects queue). The method to add a function to a queue is, unsurprisingly, `queue()`. It has three variants.

Method syntax: queue

```
queue([name])
queue([name], function)
queue([name], queue)
```

The first form returns any queue of the passed name already established on the first element in the set as an array of functions.
The second form adds the passed function to the end of the named queue for all elements in the matched set. If such a named queue doesn't exist on an element, it's created.
The last form replaces any existing queue on the matched elements with the passed queue.
When the `name` parameter is omitted, the default queue, `fx`, is assumed.

Parameters

   name   (String) The name of the queue to be fetched, added to, or replaced. If omitted, the default effects queue of `fx` is assumed.
   function   (Function) The function to be added to the end of the queue. When invoked, the function context (`this`) will be set to the DOM element upon which the queue has been established. This function is passed only one argument named `next`. `next` is another function that, when called, automatically dequeues the next item and keeps the queue moving.
   queue   (Array) An array of functions that replaces the existing functions in the named queue.

Returns

An array of functions for the first form. The jQuery collection for the remaining forms.

The `queue()` method is most often used to add functions to the end of the named queue, but it can also be used to fetch any existing functions in a queue or to replace the list of functions in a queue. Note that the third form, in which an array of functions is passed to `queue()`, can't be used to add multiple functions to the end of a queue because any existing queued functions are removed. In order to add functions to the queue, you'd fetch the array of functions that are already in the queue using the first form of the method, merge the new functions, and set the modified array back into the queue using the third form of `queue()`.

"No example here?" you say. Using the `queue()` method, you can add new animations at the end of a queue, but did you think we haven't discussed how you can execute them? Let's discover how so that we can set up a demo for you.

EXECUTING THE QUEUED FUNCTIONS

Queuing up functions for execution is not all that useful unless you can somehow cause the execution of the functions to actually occur. Enter the `dequeue()` method.

Method syntax: dequeue

dequeue([name])

Removes the foremost function in the named queue for each element in the jQuery object and executes it for each element.

Parameters

name   (String) The name of the queue from which the foremost function is to be
                        removed and executed. If omitted, the default effects queue of `fx` is assumed.

Returns

The jQuery collection

When `dequeue()` is invoked, the foremost function in the named queue for each element in the jQuery object (the first in the queue) is executed, with the function context for the invocation (`this`) being set to the element. Consider the code in the following listing, available in the file chapter-8/manual.dequeue.html.

Listing 8.8   Queuing and dequeuing functions on multiple elements

```
<!DOCTYPE html>
<html>
    <head>
        <title>Manual Dequeue</title>
        <link rel="stylesheet" href="../css/main.css" />
        <style>
            button
            {
                display: block;
                margin: auto;
            }
        </style>
    </head>
    <body>
        <button>Dequeue</button>

        <img src="images/earth.png" alt="Earth" />
        <img src="images/moon.png" alt="Moon" />

        <script src="../js/jquery-1.11.1.min.js"></script>
        <script>
            var $images = $('img');

            $images
                .queue('chain', function() {
                    console.log('First: ' + $(this).attr('alt'));
                })
                .queue('chain', function() {
                    console.log('Second: ' + $(this).attr('alt'));
                })
                .queue('chain', function() {
                    console.log('Third: ' + $(this).attr('alt'));
                })
                .queue('chain', function() {
```
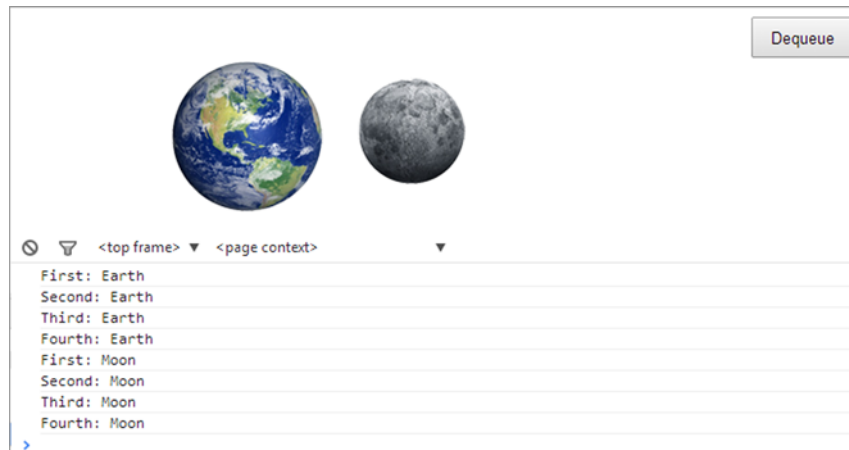
Establishes
four queued ❶
functions

```
                    console.log('Fourth: ' + $(this).attr('alt'));
               });

          $('button').click(function() {
              $images.dequeue('chain');
          });
     </script>
   </body>
</html>
```

Dequeues one
❷ function on each click

In this example, you have two images upon which you add functions to a queue
named `chain` ❶. Inside each function you emit on the console the `alt` attribute of
whatever DOM element is serving as the function context and a number relating to its
position in the queue. This way, you can tell which function is being executed and
from which element's queue. So, at the moment, you aren't doing anything special
with these images (they won't move). Upon clicking the Dequeue button, the button's
click handler ❷ causes a single execution of the `dequeue()` method. Go ahead and
click the button once and observe the messages in the console, as shown in figure 8.9.

   You can see that the first function you added to the `chain` queue for the images
has been fired twice: once for the Earth image and once for the Moon image. Clicking
the Dequeue button more times removes the subsequent functions from the queues
one at a time and executes them until the queues have been emptied, after which call-
ing `dequeue()` has no effect.

   In this example, the dequeuing of the functions was under manual control—you
needed to click the button four times (resulting in four calls to `dequeue()`) to get all
four functions executed. Frequently you want to trigger the execution of the entire set
of queued functions. For such times, a commonly used idiom is to call the `dequeue()`
method within the queued function in order or using the `next` parameter passed to
the queued function. Using one of these two techniques, you trigger the execution of
the next queued function, creating a sort of chain of calls.
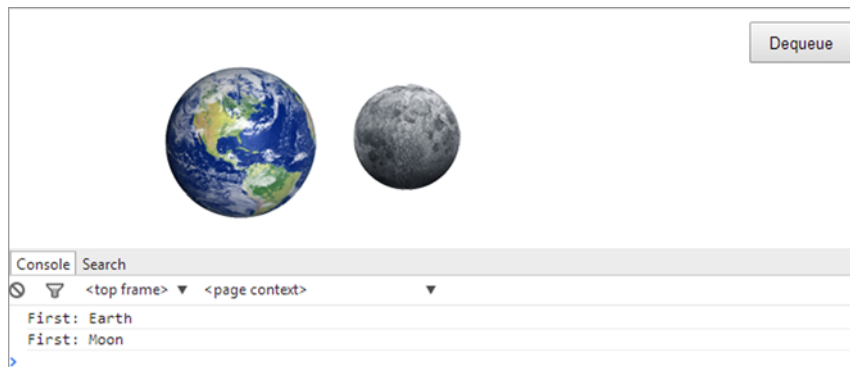
**Figure 8.8   Clicking the Dequeue button causes a single queued instance of the function
to fire, once for each image that it was established upon.**

Consider the following changes to the code in listing 8.8 that uses both techniques mentioned:

```
var $images = $('img');

$images
    .queue('chain', function(next) {
        console.log('First: ' + $(this).attr('alt'));
        next();
    })
    .queue('chain', function(next) {
        console.log('Second: ' + $(this).attr('alt'));
        next();
    })
    .queue('chain', function() {
        console.log('Third: ' + $(this).attr('alt'));
        $(this).dequeue('chain');
    })
    .queue('chain', function() {
        console.log('Fourth: ' + $(this).attr('alt'));
    });
```

The modified version of the file chapter-8/manual.dequeue.html that includes the previous snippet can be found in the file chapter-8/automatic.dequeue.html.

Bring up that page in your browser and click the Dequeue button. Note how the single click now triggers the execution of the entire chain of queued functions, as shown by figure 8.10. Also note that the last function added to the queue doesn't have a call to dequeue() because at that time the queue will already be empty.

With dequeue() you can execute the function at the front of the queue. But sometimes you may need to remove all the functions stored in the queue without executing them.

Figure 8.9   Clicking the Dequeue button causes all the functions in the queue to fire for every image that they were established upon.

CLEARING OUT UNEXECUTED QUEUED FUNCTIONS

If you want to remove the queued functions from a queue without executing them, you can do that with the clearQueue() method.

Method syntax: clearQueue

`clearQueue([name])`

Removes all unexecuted functions from the named queue.

Parameters

name   (String) The name of the queue from which the functions are to be removed
            without execution. If omitted, the default effects queue of fx is assumed.

Returns

The jQuery collection

Although similar to the stop() animation method, clearQueue() is intended for use on general queued functions rather than just animation effects.

Sometimes, instead of clearing the queue you may need to delay the execution of a queue function. Let's discuss this possibility.

DELAYING QUEUED FUNCTIONS

Another queue-oriented activity you might want to perform is to add a delay between the executions of queued functions. The delay() method enables that.

Method syntax: delay

`delay(duration[, queueName])`

Adds a delay to all unexecuted functions in the named queue.

Parameters

duration   (Number|String) The delay duration in milliseconds, or one of the strings
             "fast", "normal", or "slow", representing values of 200, 400, and 600
             respectively.

queueName   (String) The name of the queue from which the functions are to be delayed. If
             omitted, the default effects queue of fx is assumed

Returns

The jQuery collection.

To see where delay() comes in handy, imagine that you have an image, having my-image as its ID, that you want to hide and then show again after one second. You can do this with the following statement:

```
$('#my-image')
      .slideUp('slow')
      .delay(1000)
      .slideDown('fast');
```

This method is useful but not so flexible because it isn't possible to cancel a delay once it's set.

Before moving to the next chapter, there's one more thing to discuss regarding queuing functions.

8.5.3 *Inserting functions into the effects queue*

We mentioned that internally jQuery uses a queue named `fx` to queue up the functions necessary to implement the animations. What if you'd like to add your own functions to this queue in order to intersperse actions within a queued series of effects? Now that you know about the queuing methods, you can!

Think back to the previous example in listing 8.7, where you used four animations to make the moon revolve around the Earth. Imagine that you wanted to turn the background of the moon image black after the second animation (the one that moves it downward). If you just added a call to the `css()` method between the second and third animations (in bold) as follows

```
var $moonImage = $('img[alt="moon"]');
$moonImage.animate({left: '+=256'}, 2500);
$moonImage.animate({top: '+=256'}, 2500);
$moonImage.css({backgroundColor: 'black'});
$moonImage.animate({left: '-=256'}, 2500);
$moonImage.animate({top: '-=256'}, 2500);
```

you'd be disappointed because this would cause the background to change immediately, perhaps even before the first animation had a chance to start (remember that `animate()` is a non-blocking method). Rather, consider the following code (the change is in bold):

```
var $moonImage = $('img[alt="moon"]');
$moonImage.animate({left: '+=256'}, 2500);
$moonImage.animate({top: '+=256'}, 2500);
$moonImage.queue('fx',
  function() {
    $(this)
        .css({backgroundColor: 'black'});
        .dequeue('fx');
  }
);
$moonImage.animate({left: '-=256'}, 2500);
$moonImage.animate({top: '-=256'}, 2500);
```

Here you wrap the call to the `css()` method in a function that you place onto the `fx` queue using the `queue()` method. (We could have omitted the queue name, because `fx` is the default, but we made it explicit here for clarity.) This puts your color-changing function into place on the effects queue where it will be called as part of the function chain that executes as the animations progress, between the second and third animations.

But note! After you call the `css()` method, you call the `dequeue()` method on the `fx` queue. This is absolutely necessary to keep the animation queue chugging along. Failure to call `dequeue()` at this point will cause the animations to grind to a halt, because nothing is causing the next function in the chain to execute. The unexecuted

animations will just sit there on the effects queues until either something causes a dequeue and the functions commence or the page unloads and everything gets discarded.

In addition to this change, ponder this: what color will the background be after the animation is completed? Because you don't perform other changes, it will remain black. This is something you want to avoid, and so you should restore both the position and the background of the moon. The last parameter of `animate()`, the callback function, is what you need. You can restore the original background color (white) of the image inside this callback, as shown here:

```
$moonImage.animate({top: '-=256'}, 2500, function() {
    $(this).css({backgroundColor: 'white'});
});
```
← **Alternatively you can replace white with #FFFFFF or any equivalent variant.**

If you'd like to see this process in action, load the page in the file chapter-8/ revolutions.2.html into your browser and click the Animate button.

Queuing functions comes in handy whenever you want to execute functions consecutively but without the overhead, or complexity, of nesting functions in asynchronous callbacks. Today there are more advanced techniques and methods to tackle this issue (usually referred to as "callback hell"), which we'll discuss in chapter 13 when talking about the `Deferred` and `Promise` objects. But that's another chapter.

8.6   *Summary*

This chapter introduced you to the animated effects that jQuery makes available out of the box as well as to the `animate()` method that allows you to create your own custom animations.

The `show()` and `hide()` methods, when used without parameters, reveal and conceal elements from the display immediately, without any animation. You can perform animated versions of the hiding and showing of elements with these methods 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()` method toggles the displayed state of an element between hidden and shown.

Another set of methods, `fadeOut()` and `fadeIn()`, also hides and shows elements by adjusting the opacity of elements when removing or revealing them in the display. Similar to `hide()` and `show()`, the fade effect has a `fadeToggle()` method. Another method of this type, `fadeTo()`, animates a change in opacity for its elements without removing the elements from the display.

A final set of three built-in effects animates the removal or display of your selected elements by adjusting their height: `slideUp()`, `slideDown()`, and `slideToggle()`.

The previous methods introduced you to the concept of easing. The term is used to describe the manner in which the processing and pace of the frames of the animation are handled. The jQuery core provides only two easing functions to keep the library

light, but you can extend jQuery with other libraries or plugins, most notably the jQuery Easing plugin and the jQuery UI, to obtain a whole set of new easing functions.

In addition, jQuery gives you the opportunity to build your own custom animations using the `animate()` method. By using it, you can animate any CSS style property that accepts a numeric value, most commonly the opacity, position, and dimensions of the elements.

You also learned how jQuery queues animations for serial execution and how you can use the jQuery queuing methods to add your own functions to the effects queue or your own custom queues.

When we explored writing your own custom animations, you 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 custom animations as jQuery plugins. You'll learn how to do that in chapter 12, and you're encouraged to revisit these effects after you've read that chapter. Repackaging the custom effects we developed in this chapter, and any that you can think up on your own, would be an excellent follow-up exercise.

But before you write your own jQuery extensions, let's take a look at some utility functions and flags that jQuery provides.

Beyond the DOM with jQuery utility functions

This chapter covers

- The jQuery properties
- Avoiding conflict between jQuery and other libraries
- Array manipulation functions
- Extending and merging objects
- Parsing different formats
- Dynamically loading new scripts

Up to this point, we've spent a number of chapters examining the jQuery methods that operate upon a set of DOM elements selected by using the `$()` function. But you may recall that way back in chapter 1 we also introduced the concept of *utility functions*—functions namespaced by `jQuery/$` that don't operate on a jQuery object. These functions could be thought of as top-level functions except that they're defined on the `$` instance rather than `window`, keeping them out of the global scope. Generally, these functions either operate upon JavaScript objects *other* than DOM elements or perform some non-object-related operation (such as an Ajax request or the parsing of a XML string).

In addition to functions, jQuery provides some properties (sometimes referred to as *flags*) that are defined within the `jQuery/$` namespace. Some of these properties are meant for internal use only, but because they're documented in the jQuery API website and some plugins use them, we thought that they're worth a mention for the most curious among you.

You may wonder why we waited until this chapter to introduce these functions and properties. We had two reasons:

- We wanted to guide you into thinking in terms of using jQuery methods rather than resorting to lower-level operations.
- Because the methods take care of much of what you want to do when manipulating DOM elements on the pages, these lower-level functions are frequently most useful when writing the methods themselves (as well as other extensions) rather than in page-level code. (We'll tackle how to write your own jQuery plugins in chapter 12.)

In this chapter, we won't talk about the utility functions that deal with Ajax because we'll dedicate the whole of chapter 10 to them. We'll start out with those properties we mentioned.

9.1   *Using the jQuery properties*

A few features that jQuery makes available to page authors are available not via methods or functions but as properties defined on `$`. In the past, several jQuery plugin authors have relied on these features to develop their plugins. But as will be evident in a few pages, some of them have been deprecated and their use isn't recommended.

> ### jQuery 3: Properties removed
>
> jQuery 3 gets rid of the already deprecated `context` (https://api.jquery.com/context/), `support` (https://api.jquery.com/jQuery.support/), and `selector` (https://api.jquery.com/selector/) properties. If you're still using them in your project or you're employing a plugin that relies on one or more of them, upgrading to jQuery 3 will break your code.

The jQuery properties available are these:

- `$.fx.off`—Enables or disables effects
- `$.fx.interval`—Changes the rate at which animations fire
- `$.support`—Details supported features (for internal use only)

> ### For the curious: the $.browser property
>
> Before version 1.9 jQuery offered a set of properties that developers used for branching their code (performing different operations based on the value of a given property). They were set up when the library was loaded, making them available even

> *(continued)*
> before any ready handlers had executed. They were defined as properties of
> `$.browser`. The flags available were `msie`, `mozilla`, `webkit`, `safari`, and `opera`.

Let's examine these properties, starting by looking at how jQuery lets you disable
animations.

9.1.1 *Disabling animations*

There may be times when you want to conditionally disable animations in a page that
includes various animated effects. You might do so because you've detected that the
platform or device is unlikely to deal with them well, or perhaps for accessibility rea-
sons. For example, you might want to completely disable animations on low-resource
mobile devices because the effects will be sluggish, resulting in a bad experience for
the user. When you detect that you're in an animation-adverse environment or that
you don't need them, you can set the value of `$.fx.off` to `true`.

This will *not* suppress any effects you've used on the page; it'll simply disable the
animation of those effects. For example, the fade effects will show and hide the ele-
ments immediately, without the intervening animations. Similarly, calls to the
`animate()` method will set the CSS properties to the specified final values without ani-
mating them.

The `$.fx.off` flag, together with `$.fx.interval` discussed in the next section, is a
read/write flag, which means you can read as well as set its values. On the contrary,
`$.support` is meant to be read-only.

9.1.2 *Changing the animations rate*

Like any library that performs animations, jQuery has a property called `$.fx.interval`
to set the rate at which animations fire. As you know, an animation is made up of a pro-
gression of steps that, seen as a whole, create the effect. If a comparison helps, you can
think of a movie as a progression of photos shown at a regular pace. Using the
`$.fx.interval` property, you can tweak the pace at which the steps are performed.

The `$.fx.interval` property is a read/write property and its value is expressed in
milliseconds, with its default value set to 13. The latter is not random, but it's a rea-
soned choice guided by a trade-off between having smooth animations while not
stressing the CPU.

In a heavily animation-driven scenario, you may want to try to tweak this value a bit to
create even smoother animations. You can do so by setting the value of `$.fx.interval` to
a number less than 13. Modifying the rate can result in nicer effects in some browsers,
but on low-resource devices or not-so-fast browser engines (for example, those in the
older versions of Internet Explorer) this may not be true. On these browsers, not only

might you not have evident advantages, but you may also notice worse performance because of the stress of the CPU.

That being said, imagine that you want to set the value of `$.fx.interval` to `10`. To do so, you can write

```
$.fx.interval = 10;
```

In the same way that you can set a lower value, you can increase it. This change may become handy if you're working on a web page that's stressing the CPU—for example, if several animations are running at the same time or because the page performs CPU-intensive operations while running some animations. Because animations aren't as important as performing tasks, you can decide to slow down the rate at which animations fire to help the CPU. For instance, you may set the value to `100` so that the refresh happens 10 times each second:

```
$.fx.interval = 100;
```

To allow you to see the difference in how animations run depending on the value of `$.fx.interval`, we created a demo just for you that you can find in the file chapter-9/ $.fx.interval.html and also as a JS Bin (http://jsbin.com/tevoy/edit?html,css,js,output).

Now that we've finished with properties that deal with animations, let's quickly examine the one that provides information on the environment provided by the user agent.

9.1.3 *The $.support property*

jQuery has a property named `$.support` that stores the result of some feature testing that is of interest for the library. This property allows jQuery to know which features are supported and which are not supported in the browser and act accordingly. This property is intended for jQuery's internal consumption only and it's deprecated since jQuery 1.9, so we strongly encourage you to avoid using it. Some examples of the property exposed, now or in the past, are `boxModels`, `cssFloat`, `html5Clone`, `cors`, and `opacity`.

Relying on the `$.support` object is a bad idea because its properties may be removed at any time without notice when they're no longer needed internally. The deletion of these properties is done to improve the performance of the library's load because it avoids the need to perform some tests on the browser. In this book we decided to mention it because you might use an old but good plugin that relies on the `$.support` property.

This object is also one of the few cases where there's a difference between the branches of jQuery; the 1.x branch has more properties than the 2.x branch, and Compat 3.x has more properties than 3.x. For example, 1.x has `ownLast`, `inline-BlockNeedsLayout`, and `deleteExpando` that 2.x doesn't have.

Feature detection with Modernizr

If your project needs to act in a different way based on the features supported by a given browser, you should employ an approach known as *feature detection*. Instead of detecting the browser used by the user and then trying to determine whether features are supported or not (an approach known as *browser detection*), feature detection requires that you detect for the presence of the features directly.

For a project where you need to test for several features, we strongly encourage the use of an external library created for this specific purpose. The most famous and most used library is Modernizr (http://modernizr.com/). This library detects the availability of native support for features that stem from the HTML5 and CSS3 specifications.

Modernizr tests for features that are implemented in at least one major browser (there's no point in testing for a feature that nobody supports, right?) but usually supported in two or more. The following is a list of what Modernizr does for you:

- Tests for support of over 40 features in a few milliseconds
- Creates a JavaScript object (named `Modernizr`) containing the results of the tests as Boolean properties
- Adds classes to the `html` element describing what features are implemented
- Provides a script loader to allow you to use *polyfills* to backfill functionality in old browsers
- Allows using the new HTML5 sectioning elements in older versions of Internet Explorer

What's a polyfill?

A *polyfill* is a piece of code (or plugin) that provides the technology that developers expect the browser to provide natively. The term was created in 2010 by Remy Sharp, a well-known JavaScript developer and founder of the Full Frontal conference. You can find more information about how and why the term was created in the original Remy Sharp post "What is a Polyfill?" (http://remysharp.com/2010/10/08/what-is-a-polyfill/).

With this section we've covered the last property left, so we're now ready to move along and start discussing jQuery's utility functions.

9.2   *Using other libraries with jQuery*

The definition of the $ global name is usually the largest point of contention and conflict when using other libraries on the same page as jQuery. As you know, jQuery uses $ as an alias for the `jQuery` name, which is used for every feature that jQuery exposes. But other libraries, most notably Prototype, use the $ name as well.

jQuery provides the `$.noConflict()` utility function to relinquish control of the $ *identifier* to whatever other library might wish to use it. The syntax of this function is as follows.

`$.noConflict(jqueryPropertyToo)`

Restores control of the `$` identifier to another library, allowing mixed library use on pages using jQuery. Once this function is executed, jQuery features will need to be invoked using the `jQuery` identifier (the `jQuery` property of the `window` object) rather than the `$` identifier.

Optionally, the `jQuery` identifier can also be given up by passing `true` to the function.
This method should be called after including jQuery but before including the conflicting library.

Parameters

`jqueryPropertyToo`	(Boolean) If provided and set to `true`, the `jQuery` identifier is given up in addition to the `$`; otherwise it's kept.

Returns

`jQuery`

`$` is an alias for `jQuery`, so all of jQuery's functionality is still available after the application of `$.noConflict()`, albeit by using the `jQuery` property of the `window` object. But don't worry. You can still save typing some characters by defining a new, short identifier. You can compensate for the loss of the brief—yet beloved—`$`, defining your own shorter but nonconflicting alias for `jQuery`, such as

```
var $j = jQuery;
```

In case you need to give up both the `$` and the `jQuery` identifier, you can still use the jQuery methods and utility functions by using the `$.noConflict()` utility function and storing the returned value (the jQuery library) into another global property:

```
window.$new = $.noConflict(true);
```

Once this statement is executed, you're able to use the jQuery methods by employing the `$new` property just created (for example, `$new('p').find('a')`).

A design pattern you may often see employed, called Immediately-Invoked Function Expression (IIFE), consists of creating an environment where the `$` identifier is scoped to refer to the `jQuery` object. (If you've never heard of this pattern or need a refresher, please review the appendix.) This technique is commonly used in several situations: to extend jQuery, particularly by plugin authors; to simulate private variables; and to deal with closures. In the case of plugin authors, this is useful because they can't make any assumptions regarding whether page authors have called `$.noConflict()` and, most certainly, they can't subvert the wishes of the page authors by calling it themselves.

We'll cover this pattern in detail in the appendix. For the moment, understand that if you write

```
(function($) {
    // Function body here
})(jQuery);
```

you can safely use the $ identifier inside the function body, regardless of whether it's already defined by Prototype or some other library outside of the function. Pretty nifty, isn't it?

Let's prove what we've discussed in this section with a simple test. For the first part of the test, examine the HTML document in the following listing (available in chapter-9/overriding.$.test.1.html).

Listing 9.1   Overriding $ test 1

```html
<!DOCTYPE html>
<html>
   <head>
      <title>Overriding $ - Test 1</title>
      <link rel="stylesheet" href="../css/main.css"/>
   </head>
   <body>
      <button id="button-test">Click me!</button>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         $ = 'Hello world!';                          ①
         try {
            $('#button-test').on('click', function() {   ②
               alert('$ is an alias for jQuery');         ③
            });
         } catch (ex) {
            alert('$ has been replaced. The value is "' + $ + '"');  ④
         }
      </script>
   </body>
</html>
```

① Overrides the window.$ property with a custom value (a string)
② Attaches a handler to the button shown in the page
③ Alerts a successful message
④ Alerts a failure message

In this example, you import jQuery, which defines the global names jQuery and its alias $. You then redefine the global $ variable to a string value ①, overriding the jQuery definition. You replace $ with a simple string value for simplicity within this example, but it could be redefined by including another library such as Prototype.

Then you try to attach a handler to a button defined in the markup of the page ②. Inside the handler you call the JavaScript alert() function to show a success message on the screen ③. If something goes wrong, you show a failure message ④.

Loading this page in a browser proves that you see the failure alert displayed, as shown in figure 9.1. The reason for the failure is that you've redefined the value of the $ identifier. Moreover, the button doesn't trigger any action because $ was reassigned before the click handler was defined.

Now you'll make one slight change to this example to be able to use $ safely. The following code shows only the portion of the code within the try block that has been modified. The change is

$ has been replaced. The value is "Hello world!"

OK
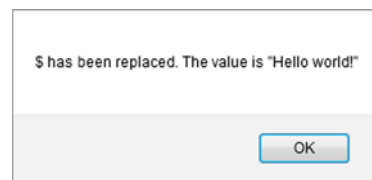
Figure 9.1   The page shows that $ has been replaced. The value is "Hello world!" because its redefinition has taken effect.

highlighted in bold for your convenience. The full code of this example can be found in the file chapter-9/overriding.$.test.2.html of the source provided with this book:

```
try {
    (function($) {
        $('#button-test').on('click', function() {
            alert('$ is an alias for jQuery');
        });
    })(jQuery);
} catch (ex) {
```

The only change you make is to wrap the statement where you attached the handler with an IIFE and pass the window.jQuery property to it. When you load this changed version and click the button, you'll see that the button now works and a different message is displayed, as shown in figure 9.2.

Changing the code as shown in the previous example lets you use the $ shortcut as an alias for jQuery, while preserving the original, global value of $ (the string "Hello world!" in this case). Now let's see how
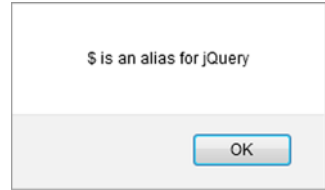
Figure 9.2   The alert now displays the success message set inside the handler.

you can restore the value of $ to whatever it was before including the jQuery library using the $.noConflict() method. A typical situation where you'd use this method is shown in the next listing and is available in the file chapter-9/$.noConflict.html and as a JS Bin (http://jsbin.com/bolok/edit?html,console).

Listing 9.2   Using the `$.noConflict()` method

```
<!DOCTYPE html>
<html>
    <head>
        <title>Using $.noConflict()</title>
        <link rel="stylesheet" href="../css/main.css"/>
    </head>
    <body>
        <script>
            window.$ = {
                customLog: function(message) {
                    console.log('The function says: ' + message);
                }
            };
        </script>
        <script src="../js/jquery-1.11.3.min.js"></script>
        <script>
            console.log('customLog: ' + ($.customLog === undefined));
            $.noConflict();
            console.log('customLog: ' + ($.customLog === undefined));
            $.customLog('Old value restored!');
        </script>
    </body>
</html>
```

❶ Creates a dummy library assigned to window.$

❷ Imports the jQuery library

❸ Prints on the console the result that tests if $.customLog is defined

❹ Restores the old value of $

❺ Prints again on the console the result that tests if $.customLog is defined

❻ Prints a message on the console using $.customLog()

In this example, you create a dummy custom library ❶ having just one method: `customLog()`. You could import any library that takes advantage of the `$` shortcut, like Prototype, in place of the dummy library, but you want to avoid adding another real library for the sake of simplicity. Then you import the jQuery library ❷ that replaces the library stored in `$`. Next you check that the dummy library has been replaced by testing that the `customLog()` method isn't defined (jQuery doesn't have such a method) ❸.

In the next line of code, you restore the value of `$` to whatever it was before the import of the jQuery library, calling jQuery's `noConflict()` function ❹. Finally, you test again if you can access the `customLog()` method via the `$` shortcut ❺ and print a message on the console using this method ❻.

Now that you've seen how you can use jQuery to avoid any interference with other libraries, it's time to learn the other jQuery utility functions.

9.3   *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 method. 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.

These functions run the gamut from simple string manipulation and type testing to complex collection filtering, serialization of form values, and even implementing a form of object inheritance through property merging. Let's start with one that's pretty basic.

9.3.1   *Trimming strings*

Almost inexplicably, before ECMAScript 5 the `String` type didn't possess a method to remove whitespace characters from the beginning and end of a string instance. Such basic functionality is part of a `String` class in most other languages, but JavaScript mysteriously lacked this useful feature until a few versions ago. What this means is that you can't use this function in versions of Internet Explorer prior to 9.

String trimming is a common need in many JavaScript applications; one prominent example is during form data validation. Because whitespace is invisible on the screen (hence its name), it's easy for users to accidentally enter extra space characters before or after valid entries in text boxes or text areas. During validation, you want to silently trim such whitespace from the data rather than alerting the user to the fact that something they can't see is tripping them up.

To help you out with older browsers, but also to help people with all browsers prior to the introduction of the JavaScript native method (`String.prototype.trim()`), the jQuery team included the `$.trim()` utility function. Behind the scenes, to improve its performance this method uses the native `String.prototype.trim()` where supported. The `$.trim()` method is defined 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 in this case 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 character `\u00A0`.

Parameters

`value`   (String) The string value to be trimmed. This original value isn't modified.

Returns

The trimmed string.

A simple example of using this function to trim the value of a text field is

```
var trimmedString = $.trim($('#some-field').val());
```

Be aware that this function converts the parameter you pass to its `String` type equivalent, so if you erroneously pass an object to it, you'll obtain the string `"[object Object]"`.

Now let's look at some functions that operate on arrays and other objects.

9.3.2 *Iterating through properties and collections*

Oftentimes when you have nonscalar values composed of other components you'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 you the means to iterate over them. For arrays, you iterate over their elements using the `for` loop; for objects, you iterate over their properties using the `for...in` loop (other constructs are available, but let's ignore them for the moment).

You can code examples of each as follows:

```
var anArray = ['one', 'two', 'three'];
for (var i = 0; i < anArray.length; i++) {
  // Do something here with anArray[i]
}
var anObject = {one: 1, two: 2, three: 3};
for (var prop in anObject) {
  // Do something here with prop
}
```

Pretty easy stuff, but some might think that the syntax is needlessly wordy and complex—a criticism frequently targeted at the `for` loop.

A few years ago, a method of the `Array` object called `forEach()` was added to JavaScript. Unfortunately, being a later addition, some browsers, most notably versions of Internet Explorer prior to 9, don't support it. In addition, belonging to the `Array` object, it can't be employed to iterate over other kinds of objects. jQuery to the rescue!

You know that jQuery defines the `each()` method, allowing you to easily iterate over the elements in a jQuery collection without the need for the `for` loop syntax. For arrays, array-like objects, and objects, jQuery provides an analogous utility function named `$.each()`.

The really nice thing is that the same syntax is used, whether iterating over the items in an array or the properties of an object. Besides, it can be used in Internet Explorer 6–8 as well. Its syntax is as follows.

Function syntax: $.each

`$.each(collection, callback)`

A generic iterator function, which can be used to seamlessly iterate over both objects and arrays. Arrays and array-like objects with a length property (such as a function's `arguments` object) are iterated by numeric index, from 0 to length-1. Other objects are iterated via their named properties.

Parameters

`collection`	(Array|Object) An array (or array-like object) whose items are to be iterated over, or an object whose properties are to be iterated over.
`callback`	(Function) A function invoked for each element in the collection. If the collection is an array (or array-like object), 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 array item or property value. The function context (`this`) of the invocation is also set to the value passed as the second parameter.

Returns

The same collection passed.

This unified syntax can be used to iterate over arrays, array-like objects, or objects using the same format. With this function, you can write the previous example as follows:

```
var anArray = ['one', 'two', 'three'];
$.each(anArray, function(i, 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 sounds good, 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 example:

```
$.each(anArray, someComplexFunction);
```

Note that when iterating over a collection, you can break out of the loop by returning `false` from the iterator function. On the contrary, returning a *truthy* value (values evaluating to `true`) is the same as using `continue`, which means that the function stops immediately and the next iteration is performed.

jQuery 3: Feature added

jQuery 3 introduces the ability to iterate over the DOM elements of a jQuery collection using the `for-of` loop, part of the ECMAScript 6 specifications. Thanks to this feature, you can now write code like the following:

```
var $divs = $('div');
for (var element of $divs) {
    // Do something with element
}
```

Please note how we didn't prepend the dollar sign in front of the `element` variable to highlight that its value will be a DOM element and not a jQuery collection made of one element at a time.

> **NOTE** Using the `$.each()` function may be convenient from a syntax point of view, but it's usually (slightly) slower than using the old-fashioned `for` loop. Whether you should use it or not is really up to you.

Sometimes you may iterate over arrays to pick and choose elements to become part of a new array. Although you could use `$.each()` for that purpose, let's see how jQuery makes that even easier.

9.3.3 *Filtering arrays*

Traversing an array to find elements that match certain criteria is a frequent need of applications that handle lots of data. You might wish to filter the data 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 `$.grep()` function might lead you to believe that the function employs the use of regular expressions like its namesake UNIX `grep` command. But the filtering criterion used by the `$.grep()` utility function isn't a regular expression; it's a callback function provided by the caller that defines the criteria to determine whether 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 their use isn't 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 value. The return value of the callback function determines whether 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.

Function syntax: $.grep *(continued)*

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) A function whose return value determines whether the current data value is to be collected. This function receives two parameters: the current value for this iteration and the index of the value within the array. 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.
`invert`	(Boolean) An optional value that if `true` inverts the normal operation of the function.

Returns

The array of collected values.

Let's say that you want to filter an array for all values that are greater than 100. You'd do that with a statement such as the following:

```
var bigNumbers = $.grep(originalArray, function(value) {
             return value > 100;
          });
```

The callback function that you pass to `$.grep()` can use whatever processing it likes to determine if the value should be included. The decision could be as easy or as complex as you need.

Even though the `$.grep()` function doesn't directly use regular expressions (despite its name), JavaScript regular expressions can be powerful tools in your callback functions to determine whether to include or exclude values from the resultant array. Consider a situation in which you 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).

U.S. 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 you could filter a source array for nonconformant 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 isn't the only operation you might perform on them. Let's look at another utility function that jQuery provides.

9.3.4 *Translating arrays*

Data might not always be in the format that you 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, or an array from an object, jQuery makes it even easier with the `$.map` utility function.

Function syntax: $.map

`$.map(collection, callback)`

Iterates through the passed array or object, invoking the callback function for each item and collecting the return values of the function invocations in a new array.

Parameters

collection	(Array|Object) An array or object whose values are to be transformed to values in the new array.
callback	(Function) 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 value and the index of that value within the original array. If an object is passed, the second argument is the property name of the current value.

Returns

The array of collected values.

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 the passed array into the following:

```
[1, 2, 3, 4, 5]
```

An 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 the one-to-one correspondence between items by order is lost.

Let's now look at a slightly more involved example. Imagine that you have an array of strings, perhaps collected from form fields, that is expected to represent numeric values. You want to convert this array of strings to an array of corresponding `Number` instances. Because there's no guarantee against the presence of an invalid numeric string, you need to take some precautions. Consider the following code, which is also

available in the file chapter-9/\$.map.html and as a JS Bin (http://jsbin.com/
zonopor/edit?html,js,console):

```
var strings = ['1', '2', '3', '4', 'S', '6'];
var values = $.map(strings, function(value) {
   var result = new Number(value);
   return isNaN(result) ? null : result;
});
```

You 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 the letter *S* instead
of the expected number *5*. The 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 `NaN`.
But the funny thing about `NaN` is that, by definition, it doesn't equal anything else,
including itself! Therefore the value of the expression `NaN===NaN` is `false`!

Because you 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 you employ to
test the result of the string-to-numeric conversion.

In this example, you return `null` in the case of failure, ensuring that the resulting
array contains only the valid numeric values with any error values elided. If you want
to collect all the values, you can allow the transformation function to return `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'],
   function(value) {
      return value.split('');
   }
);
```

This statement transforms an array of strings into an array of all 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']
```

This is accomplished by use of JavaScript's `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 here. There are a handful of minor func-
tions that you might find handy.

9.3.5 *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[, fromIndex])`
Returns the index position of the first occurrence of the passed value.

Parameters

`value`	(Any) The value for which the array will be searched.
`array`	(Array) The array to be searched.
`fromIndex`	(Number) The index of the array at which to begin the search. The default is 0, which will search the whole array.

Returns

The index of the first occurrence of the value within the array, or `-1` if the value isn't 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. Consider the following snippet:

```
var images = document.getElementsByTagName('img');
```

This populates the variable `images` with a `HTMLCollection` of all the images on the page.

Dealing with this and similar objects is a bit of a pain because they lack native JavaScript `Array` methods like `sort()` and `indexOf()`. Converting a `HTMLCollection` (and similar objects) to a JavaScript array makes things a lot nicer. The jQuery `$.makeArray` function is what you need in this case.

Function syntax: $.makeArray

`$.makeArray(object)`
Converts the passed array-like object into a JavaScript array

Parameters

`object`	(Object) Any object to turn into a native `Array`

Returns

The resulting JavaScript array

This function is intended for use in code that makes little use of jQuery, which internally handles this sort of thing on your behalf. This function also comes in handy

when handling the `arguments` array-like object within functions. Imagine that you have the following function and that internally you want to sort the arguments provided to it:

```
function foo(a, b) {
   // Sort arguments here
}
```

You can grab all the arguments at once using the `arguments` array-like. The problem is that `arguments` is not of type `Array`, so you can't write:

```
function foo(a, b) {
   var sortedArgs = arguments.sort();
}
```

This code will throw an error because `arguments` doesn't possess the JavaScript `Array`'s `sort()` method that would be useful to sort the arguments. This is a situation where the `$.makeArray()` can help. You can fix the issue by converting `arguments` into a real array and then sort its elements:

```
function foo(a, b) {
   var sortedArgs = $.makeArray(arguments).sort();
}
```

Once you make this change, `sortedArgs` will contain an array with the arguments passed to the `foo()` function sorted as required.

Let's now say that you have the following statement:

```
var arr = $.makeArray({a: 1, b: 2});
```

Once this statement is executed, `arr` will contain an array made of one element, which is the object passed as an argument to `$.makeArray()`.

Another seldom-used function that might be useful 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, sorted in document order

Parameters

   `array`   (Array) The array of DOM elements to be examined

Returns

An array of DOM elements returned in document order, consisting of the unique elements in the passed array

---

This function is intended for use on element arrays made of DOM elements created outside the bounds of jQuery. Although many people think that this function can be used with arrays of strings or numbers, we want to stress that `$.unique()` works only with arrays of DOM elements.

Before delving into the description of the next function, let's look at an example of using `$.unique()`. Consider the following markup:

```html
<div class="black">foo</div>
<div class="red">bar</div>
<div class="black">baz</div>
<div class="red">don</div>
<div class="red">wow</div>
```

Now imagine that for some reason you need to retrieve the `<div>`s having class `black` as an array of DOM elements, then add all the `<div>`s in the page to this collection, and finally filter the latter to remove the duplicates. You can achieve these requirements with the following code (which includes some statements to log the difference in the number of elements):

```js
var blackDivs = $('.black').get();
console.log('The black divs are: ' + blackDivs.length);
var allDivs = blackDivs.concat($('div').get());
console.log('The incremented divs are: ' + allDivs.length);
var uniqueDivs = $.unique(allDivs);
console.log('The unique divs are: ' + uniqueDivs.length);
```

In case you want to play with this example, you can find it in the file chapter-9/ $.unique.html and online as a JS Bin (http://jsbin.com/borin/edit?html,js,console).

> ### jQuery 3: Method renamed
>
> jQuery 3 renamed the `$.unique()` utility function in `$.uniqueSort()` to make it clear that the function sorts as well. Despite this change, in jQuery 3 you can still invoke `$.unique()`, which is now just an alias for `$.uniqueSort()`, but it's now deprecated and will be removed in following versions of the library.

Now suppose that you want to merge two arrays. jQuery has a function for this task called `$.merge`.

Function syntax: $.merge

`$.merge(array1, array2)`

Merges the values of the second array into the first and returns the result. The first array is modified by this operation and returned as the result. The second array isn't modified.

Parameters

`array1`	(Array) An array into which the other array's values will be merged.
`array2`	(Array) An array whose values will be merged (concatenated) into the first array.

Returns

The first array, modified with the results of the merge.

Consider the following code:

```
var arr1 = [1, 2, 3, 4, 5];
var arr2 = [5, 6, 7, 8, 9];
var arr3 = $.merge(arr1, arr2);
```

After this code executes, `arr2` is untouched, but `arr1` and `arr3` contain the following:

```
[1, 2, 3, 4, 5, 5, 6, 7, 8, 9]
```

Note how there are two occurrences of `5` because the `$.merge()` utility function doesn't remove the duplicates.

Now that you've seen how jQuery helps you to easily work with arrays, let's see how it helps you manipulate plain-old JavaScript objects.

9.3.6 *Extending objects*

Although we all know that JavaScript provides some features that make it act in many ways like an object-oriented language, 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* (supported in the ECMAScript 6)—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.

It's fairly easy to write JavaScript code to perform this extension by copying, but as with so many other procedures, jQuery anticipates this need and provides a ready-made utility function to help you out: `$.extend()`. As you'll see in chapter 12, which is about jQuery plugins, this function is useful for much more than extending an object. Its syntax is as follows.

Function syntax: $.extend

`$.extend([deep,] target, [source1, source2, ... sourceN])`

Extends the object passed as `target` with the properties of the remaining passed objects. The extended object is also provided as the return value.

Parameters

`deep`	(Boolean) An optional flag that determines whether a deep or shallow copy is made. If omitted or `false`, a shallow copy is executed. If `true`, a deep copy is performed.
`target`	(Object) The object whose properties are augmented with the properties of the source object(s). If this object is the only parameter provided, it's assumed as a source and the jQuery object is assumed to be the target. If more than one argument is given, 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 names exist in the sources, sources later in the argument list override those earlier in the list.

Returns

The extended target object.

The behavior of this function is interesting because of its flexibility. Almost every argument in the signature is optional and allows you to change what the function does. Arguments that are `null` or `undefined` are ignored.

If only one object is provided, then it's interpreted not as a target but as a source. In this case, the jQuery object is assumed to be the target and properties of the object are merged into the jQuery object.

It's worth noting that the merged object may have fewer properties than the sum of those of the target and the sources even if they're all different. The reason is that `$.extend()` ignores the properties whose value is `undefined`. Let's take a look at this function with an example. You'll set up three objects, a target and two sources, as follows:

```
var target = {a: 1, b: 2, c: 3};
var source1 = {c: 4, d: 5, e: 6};
var source2 = {c: 7, e: 8, f: 9};
```

Then you'll operate on these objects using `$.extend()` as follows:

```
$.extend(target, source1, source2);
```

This code takes the contents of the source objects and merges them into the target. To test this code, we've set up this example code in the file chapter-9/$.extend.test.1.html, which executes the code and displays the results on the page. Loading this page into a browser results in the display shown in figure 9.3.

As you can see, all properties of the `source` objects have been merged into the `target` object. But note the following important nuances:

$.extend() – Test 1

target (before) = {a: 1, b: 2, c: 3}
source1 = {c: 4, d: 5, e: 6}
source2 = {c: 7, e: 8, f: 9}
target (after) = {a: 1, b: 2, c: 7, d: 5, e: 8, f: 9}

Figure 9.3   The `$.extend()` function merges properties from multiple source objects without duplicates and gives precedence to instances in reverse order of specification.

- All of the objects contain a property named `c`. The value of `c` in `source1` replaces the value in the original `target`, which in turn is replaced by the value of `c` in `source2`.
- Both `source1` and `source2` contain a property named `e`. The value of `e` within `source2` overrides 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.

Before moving forward, let's look at another example of a situation that you, like your dear authors, have faced several times. Let's say that you want to merge the properties of two objects preserving both of them, which means that you don't want the `target` object to be modified. To perform this operation, you can pass an empty object as the target, as shown here:

```
var mergedObject = $.extend({}, object1, object2);
```

By passing an empty object as the first parameter, both `object1` and `object2` are treated as sources; thus they're not modified.

As the last example, we'll show you the result of performing a deep copy of two objects using the first parameter of `$.extend()`. Let's say that you have the following objects:

```
var target =  {a: 1, b: 2};
var source1 = {b: {foo: 'bar'}, c: 3};
var source2 = {b: {hello: 'world'}, d: 4};
```

You operate on these objects calling the `$.extend()` method as follows:

```
$.extend(true, target, source1, source2);
```

Once again, we've created a page that reproduces this example so that you can play with it. The code is contained in the file chapter-9/$.extend.test.2.html. Loading the latter into your browser will give you the results shown in figure 9.4.

This function is really useful and you'll use it a lot. Let's now move on because we still have a few other utility functions to examine.

$.extend() – Test 2

target (before) = {a: 1, b: 2}
source1 = {b: {foo: bar}, c: 3}
source2 = {b: {hello: world}, d: 4}
target (after) = {a: 1, b: {foo: bar, hello: world}, c: 3, d: 4}

Figure 9.4   An example of how the `$.extend()` function can merge nested objects

9.3.7   *Serializing parameter values*

It should come as no surprise that in a dynamic, highly interactive application. Submitting requests is a common occurrence. Frequently, these requests will be submitted as a result of a form submission, where the browser formats the request body containing the request parameters on your behalf. Other times, you'll submit requests as URLs in the `href` attribute of `a` elements. In these latter cases, it becomes your responsibility to correctly create and format the query string that contains any request parameters you wish to include with the request.

Server-side templating tools generally have great mechanisms that help you construct valid URLs, but when creating them dynamically on the client, JavaScript doesn't give you much in the way of support. Remember that not only do you need to correctly place all the ampersands (`&`) and equal signs (`=`) that format the query string parameters, but you also need to make sure that each name and value is properly

URI-encoded. Although JavaScript provides a handy function for that (`encodeURI-Component()`), the formatting of the query string falls squarely into your lap.

As you might have come to expect, jQuery anticipates that burden and gives you a tool to make it easier: the `$.param()` utility function.

Function syntax: $.param

`$.param(params[, traditional])`

Serializes the passed information into a string suitable for use as the query string of a submitted request. The passed value can be an array of form elements, a jQuery object, or a JavaScript object. The query string is properly formatted and each name and value in the string is properly URI-encoded.

Parameters

`params`	(Array|jQuery|Object) The value to be serialized into a query string. If an array of elements or a jQuery object is passed, the name/value pairs represented by the included form controls are added to the query string. If a JavaScript object is passed, the object's properties form the parameter names and values.
`traditional`	(Boolean) An optional flag indicating whether to perform a traditional shallow serialization. This generally affects only source objects with nested objects. If omitted, defaults to `false`.

Returns

The formatted query string.

To see this method in action, consider the following statement:

```
$.param({
    'a thing': 'it&s=value',
    'another thing': 'another value',
    'weird characters': '!@#$%^&*()_+='
});
```

Here you pass an object with three properties to the `$.param()` function, in which the names and the values all contain characters that must be encoded within the query string in order for it to be valid. The result of this function call is

```
a+thing=it%26s%3Dvalue&another+thing=another+value&weird+characters=!%40%23%2
    4%25%5E%26*()_%2B%3D
```

Note that the query string is formatted correctly and that the non-alphanumeric characters in the names and values have been properly encoded. This might not make the string all that readable to you, but server-side code lives for such strings!

One note of caution: if you pass an array of elements or a jQuery object that contains elements other than those representing form values, you'll end up with a bunch of entries such as

```
undefined=&
```

in the resulting string, because this function doesn't weed out inappropriate elements in its passed argument.

You might be thinking that this isn't a big deal because, after all, if the values are form elements, they're going to end up being submitted by the browser via the form, which will handle all of this for you. Well, hold onto your hat. In chapter 10, when we start talking about Ajax, you'll see that form elements aren't always submitted by their forms!

But that's not going to be an issue, because you'll also see later on that jQuery provides a higher-level means (that internally uses this very utility function) to handle this sort of thing in a more sophisticated fashion.

Let's now consider another example. Imagine that you have the following form in your page:

```
<form>
    <label for="name">Name:</label>
    <input id="name" name="name" value="Aurelio" />
    <label for="surname">Surname:</label>
    <input id="surname" name="surname" value="De Rosa" />
    <label for="address">Address:</label>
    <input id="address" name="address" value="Fake street 1, London, UK" />
</form>
```

If you call the `$.param()` utility function by passing to it a jQuery object containing all the `input` elements of this form as shown here,

```
$.param($('input'));
```

you'll obtain the following string as a result:

```
name=Aurelio&surname=De+Rosa&address=Fake+address+123%2C+London%2C+UK
```

This example should have clarified how `$.param()` works with form elements. But our discussion of this function isn't complete yet.

SERIALIZING NESTED PARAMETERS

Trained by years of dealing with the limitations of HTTP and HTML form controls, web developers are conditioned to think of serialized parameters, aka query strings, as a flat list of name/value pairs. For example, imagine a form in which you collect someone's name and address. The query parameters for such a form might contain names such as `firstname`, `lastname`, and `city`. The serialized version of the query string might be this:

```
firstname=Yogi&lastname=Bear&streetaddress=123+Anywhere+Lane&city=Austin&stat
    e=TX&postalcode=78701
```

The preserialized version of this construct would be as follows:

```
{
  firstname: 'Yogi',
  lastname: 'Bear',
  streetaddress: '123 Anywhere Lane',
  city: 'Austin',
  state: 'TX',
  postalcode : '78701'
}
```

As an object, that doesn't really represent the way that you'd think about such data. From a data organization point of view, you might think of this data as two major elements, a name and an address, each with its own properties, perhaps something along the lines of this:

```
{
  name: {
    first: 'Yogi',
    last: 'Bear'
  },
  address: {
    street: '123 Anywhere Lane',
    city: 'Austin',
    state: 'TX',
    postalcode : '78701'
  }
}
```

But this nested version of the element, though more logically structured than the flat version, doesn't easily lend itself to conversion to a query string. Or does it?

By using a conventional notation employing square brackets, such a construct could be expressed as the following:

```
name[first]=Yogi&name[last]=Bear&address[street]=123+Anywhere+Lane&address[ci
    ty]=Austin&address[state]=TX&address[postalcode]=78701
```

In this notation, subproperties are expressed using square brackets to keep track of the structure. Many server-side languages such as PHP can handily decode these strings.

This is a smart behavior and it's not the traditional way JavaScript treats such objects. What you might expect is something like

```
name=[object+Object]&address=[object+Object]
```

which isn't helpful at all.

Fortunately, jQuery is able to deal with nested parameters, allowing you to decide when to apply the traditional or the smart behavior. All you need to do is pass `true` to obtain the traditional object and `false` (or omit it entirely) for the smart behavior, as the second parameter of `$.param()`.

You can prove this to yourself with the `$.param()` Lab page provided in the file chapter-9/lab.$.param.html and shown in figure 9.5.

This lab page lets you see how `$.param()` will serialize flat and nested objects, using its smart algorithm, as well as the traditional algorithm.

Go ahead and play around with this lab before moving to the next section. In the page we've set two sample objects so that you can immediately start testing `$.param()`. We also added the ability to edit them and to add new properties, so you can play with different object structures that you might want to serialize.
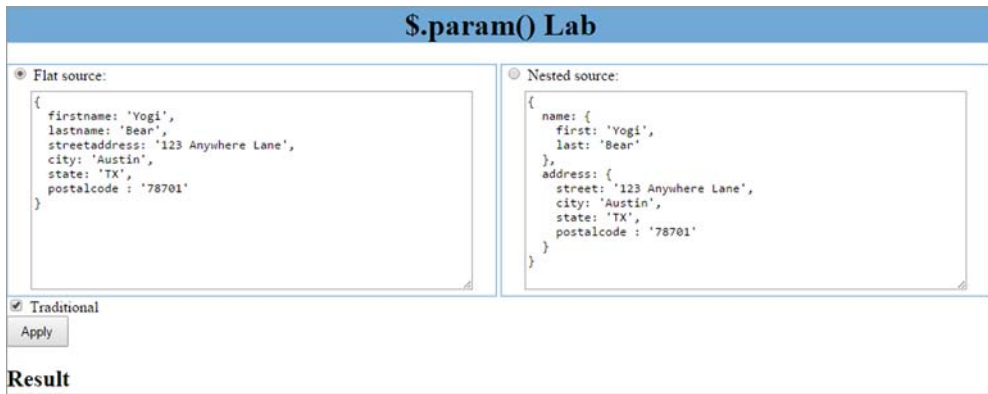
Figure 9.5   The `$.param()` Lab lets you see how flat and nested objects are serialized using the new and traditional algorithms.

9.3.8   Testing objects

You may have noticed that many of the jQuery methods and utility functions have rather malleable parameter lists; optional parameters can be omitted without the need to include `null` values as placeholders. Take the `on()` method as an example. Its most used signature is

```
on(eventType[, selector][, data], handler)
```

If you have no selector or data to pass, you can simply call `on()` with the handler function as the second parameter; there's no need for placeholders. jQuery handles this by testing the types of the arguments, and if it sees that there are only two arguments and that a function is passed as the second argument, it interprets that as the handler rather than as a selector or a data argument.

Testing arguments for various types, including whether they are functions or not, will certainly come in handy if you want to create your own functions and methods that are similarly friendly and versatile. For this reason, jQuery exposes a number of testing utility functions, as outlined in table 9.1.

Table 9.1   jQuery utility functions for testing objects

Function	Description
`$.isArray(param)`	Returns `true` if `param` is a JavaScript array (but not if `param` is any other array-like object like a jQuery set); `false` otherwise
`$.isEmptyObject(param)`	Returns `true` if `param` is a JavaScript object with no properties, including any inherited from `prototype`; `false` otherwise
`$.isFunction(param)`	Returns `true` if `param` is a function; `false` otherwise
`$.isNumeric(param)`	Returns `true` if `param` represents a numeric value; `false` otherwise
`$.isPlainObject(param)`	Returns `true` if `param` is a JavaScript object created via `{}` or `new Object()`; `false` otherwise

Table 9.1   jQuery utility functions for testing objects *(continued)*

Function	Description
`$.isWindow(param)`	Returns `true` if `param` is the `window` object; `false` otherwise
`$.isXMLDoc(param)`	Returns `true` if `param` is an XML document or a node within an XML document; `false` otherwise

Knowing these functions is nice, but you may feel that a practical example would be even nicer. Here you go!

Let's say that you want a function that accepts either an array or an object as its first parameter and multiplies each numeric item of the array or value of the object by a given number passed as its second parameter. In addition, you want to specify a function that's applied after the multiplication of an item as its third parameter. To have more fun, the second argument (which we'll call `factor`) and the third one (which we'll call `customFunction`) will be optional. This means that you can avoid both of them as well as just one. The function must return a new object of the same type as the first parameter, without modifying the latter.

Based on this description, the signature of the function can be represented like this:

```
multiplier(collection[, factor][, customFunction])
```

Thanks to the methods listed in table 9.1, you're able to deal with all these cases without much hassle. Implementing the function results in code like that in the following listing. You can also find this code with some tests in the file chapter-9/testing .functions.html and as a JS Bin (http://jsbin.com/lolub/edit?js,console).

Listing 9.3   Testing utility functions

```
function multiplier(collection, factor, customFunction) {
    function calc(value) {
        return $.isFunction(factor) ?
            factor(value) :
            $.isFunction(customFunction) ?
            customFunction(value * factor) :
            value * factor;
    }

    var result = null;

    if (factor === undefined && customFunction === undefined) {
        factor = 1;
    }

    if ($.isArray(collection)) {
        result = $.map(collection, function(value) {
            if ($.isNumeric(value)) {
                return calc(value);
            }
        });
```

① Defines the calc() support function that has the core calculation

② If the second and the third arguments are undefined, set factor to **1**.

③ If dealing with an array, use the $.map() function to call calc() on each array's value.

```
    } else if ($.isPlainObject(collection)) {
      result = {};
      for(var prop in collection) {
        if ($.isNumeric(collection[prop])) {
          result[prop] = calc(collection[prop]);
        }
      }
    }

    return result;
}
```

If dealing with an object, use a ❹ for…in loop to call calc() on each object's value.

❺ **Returns the result of the calculation**

The code of this listing is interesting and uses many of the functions covered in this chapter, so we'll describe it in more detail.

In the first part of the function, you define the `calc()` support function that has the core calculation ❶. It deals with the variable parameters of the `multiplier()` function and it performs different operations based on the value passed as the second and the third arguments. If the second and the third arguments are `undefined`, you set `factor` to 1. Then the real calculation starts. If `collection` is of type `Array`, the function uses the `$.map()` utility function to call `calc()` on each array's value, but only if it's a number ❸. If `collection` is of type `Object`, the function employs a for…in loop to call `calc()` on each object's value, but only if it's a number ❹. Finally, the result is returned ❺. The data type of the result depends on the data type of the first argument (`Array` or `Object`). In case `collection` is neither an `Array` nor an `Object`, `null` is returned.

The functions described in this section allow you to test if a variable contains a value of a particular type (`Object`, `Array`, `Function`, and so on) or not. But what if the information you want to know is the type itself?

DISCOVERING THE TYPE OF A VALUE

jQuery has one additional utility function that deals with types called `$.type()`. The syntax of this function is the following.

Function syntax: $.type

`$.type(param)`
Determines the type of a value

Parameters

param   (Any) The value to test

Returns

A string describing the type of the value

To give you an idea of the result of calling this function, let's say that you have a statement like the following:

```
$.type(3);
```

In this case you'll obtain this result:

```
"number"
```

If you have this statement

```
$.type([1, 2, 3]);
```

you'll obtain this string as the result:

```
"array"
```

This is an important difference compared to the usual way of testing types in JavaScript that will come in handy when we delve into developing plugins. In fact, if you perform the test

```
if (typeof [1, 2, 3] === 'array')
```

you'll obtain `false` because the value returned by `typeof [1, 2, 3]` is `"object"`.

Now that you have a complete overview of the utility functions that deal with data types, let's look at a bunch of functions that allow you to parse strings of different types.

9.3.9 *Parsing functions*

jQuery provides a set of utility functions to parse several formats ranging from JSON to XML and HTML. Modern browsers provide an object called `JSON` that deals with the JSON format. This object has the `parse()` method that, as the name says, parses a JSON string. Exactly, modern browsers…. As always, this means that not all of your users have support for this feature. Fortunately, jQuery comes again to your help, providing the `$.parseJSON()` utility function.

Function syntax: $.parseJSON

`$.parseJSON(json)`
Parses the passed JSON string, returning its evaluation

Parameters

  `json`   (String) The JSON string to be parsed

Returns

The evaluation of the JSON string

You've seen several times that when the browser supports the native methods, jQuery uses it, and this case is no exception. If the browser supports `JSON.parse()`, jQuery will use it; otherwise it will use a JavaScript trick to perform the evaluation. In doing so, jQuery improves the performance of the operation where possible.

The JSON string must be completely well-formed, and the rules for well-formed JSON are much stricter than JavaScript expression notation. For example, all property names must be delimited by double-quote characters, even if they form valid identifiers.

Invalid JSON will result in an error being thrown. See http://www.json.org for the nitty-gritty on well-formed JSON.

JSON isn't the only format used on the web to exchange information. Another commonly used one is XML. jQuery allows you to easily parse a string containing XML, turning it into its equivalent XML document by using the `$.parseXML()` utility function.

<div style="text-align:center">**Function syntax: $.parseXML**</div>

`$.parseXML(xml)`
Parses an XML string into an XML document

Parameters

`xml`	(String) The XML string to be parsed

Returns

The XML document derived from the string

XML documents are easy to use and traverse because jQuery supports them, so you can pass the object returned by the `$.parseXML()` function to jQuery and then use the methods you've learned so far. Sound crazy? Don't worry; in a moment we'll present a lab page that will enable you to play with this concept.

The last method belonging to this category is `$.parseHTML()`. It uses a native DOM element creation function to convert a string containing HTML markup to a set of DOM elements. Then, you can operate on these elements using the jQuery methods. The syntax of this function is the following.

<div style="text-align:center">**Function syntax: $.parseHTML**</div>

`$.parseHTML(html[, context][, keepScripts])`
Parses a string into an array of DOM nodes.

Parameters

`html`	(String) The HTML string to be parsed.
`context`	(Element) An optional element to use as the context in which the HTML fragment will be generated. If not specified or if `null` or `undefined` is passed, the default value is the (current) `document`.
`keepScripts`	(Boolean) If `true` the function will keep and include the scripts in the HTML string. The default value is `false`.

Returns

An array of DOM elements derived from the string.

In the description of the function, we specified that the default value of `keepScripts` is `false`. There's an important security concern behind this decision. When fetching the HTML string from an external source, if you include the scripts, you enable a malicious person to perform an attack on your website.

It's worth mentioning that in most environments, even if you get rid of the `script` elements, it's still possible to perform an attack, so you should make certain you

escape untrusted inputs from the source such as the URL or cookies. As an example of attacks that don't come via a `script` element, think of those that can be performed using the `onerror` attribute of an `img` element.

The three utility functions we've just discussed gave us the chance to create another lab page, which you can find in the file chapter-9/lab.parsing.html and which is shown in figure 9.6.

This lab page has three prebuilt snippets of code. The first one is a JSON object that you can query using dot notation. We made this possible by using the `$.parse-JSON()` function to turn the raw text into a JavaScript object.

Then another section allows you to query XML and HTML code in the same way you saw for the lab page of chapter 2. This time we'll only show how many elements have been selected. For this part of the lab page we've employed `$.parseXML()` and `$.parseHTML()`.

Note that you can modify all three code snippets because we put them into a `textarea` element. Therefore, you can test your own JSON object or any XML or HTML code that you want. Open the lab page and play with it.

Once you feel comfortable with these methods, you're ready to learn about the other utility functions that can't be grouped into a category.

Figure 9.6   The Parsing Functions Lab lets you play with the jQuery functions that deal with the three supported formats: JSON, XML, and HTML.

9.4   *Miscellaneous utility functions*

This section will explore the set of utility functions that are pretty much a category on their own. We'll start with one that doesn't seem to do much.

9.4.1   *Doing nothing*

jQuery defines a utility function that does nothing, literally. This function could have been named `$.uselessFunctionThatDoesNothing()`, but that's a tad long, so it's named `$.noop()`. It's defined with the following syntax.

Function syntax: $.noop

`$.noop()`

Does nothing

Parameters

  none

Returns

`undefined`

Hmmm, a function that is passed nothing, does nothing, and returns `undefined`. What's the point?

Do you recall that many jQuery methods are passed parameters or option values that are optional function callbacks? `$.noop()` serves as a handy default for those callbacks when the user doesn't supply one.

9.4.2   *Testing for containment*

When you want to test one element for containment within another, jQuery provides the `$.contains()` utility function.

Function syntax: $.contains

`$.contains(container, contained)`

Tests if one element is contained within another in the DOM hierarchy

Parameters

  `container`   (Element) The DOM element being tested as containing another element
  `contained`   (Element) The DOM element being tested for containment

Returns

`true` if the `contained` element is contained within the `container`; `false` otherwise

---

Hey, wait a minute! Doesn't this sound familiar? Indeed, we discussed the `has()` method back in chapter 2, to which this function bears a striking resemblance.

This function, used frequently internally to jQuery, is most useful when you already have references to the DOM elements to be tested and there's no need to take on the overhead of creating a jQuery collection.

To see this method in action, consider the following markup:

```
<div id="wrapper">
   <p id="description">Some text</p>
</div>
<div id="empty"></div>
```

If you execute the following two statements

```
console.log($.contains(
   document.getElementById('wrapper'),
   document.getElementById('description')
));
console.log($.contains(
   document.getElementById('empty'),
   document.getElementById('description')
));
```

you'll obtain the following result on the console:

```
true
false
```

The reason is that the element having `description` as its ID is a descendant of the element having `wrapper` as its ID, but not of the one whose ID is `empty`.

This section and the previous one were simple to digest. Now turn your attention to one of the more esoteric utility functions—one that lets you have a pronounced effect on how event listeners are called.

9.4.3 *Prebinding function contexts*

As you've seen throughout our examination of jQuery, functions and their contexts play an important role in jQuery-using code. The context of a function—what's pointed to by `this`—is determined by how the function is invoked and not how it's defined (see the appendix if you want to review this concept). When you want to call a particular function and explicitly control what the function context will be, you can use JavaScript's `call()` or `apply()` method to invoke the function.

But what if you're not the one calling the function? What if, for example, the function is a callback? In that case, you're not the one invoking the function so you can't use the previously mentioned methods to affect the setting of the function context.

jQuery provides a utility function by which you can prebind an object to a function so that when the function is invoked, the bound object will become the function context. This utility function is named `$.proxy()`, and its syntax is as follows.

Function syntax: $.proxy

```
$.proxy(function, proxy[, argument, ..., argument])
$.proxy(proxy, property[, argument, ..., argument])
```
Takes a function and returns a new one that will have a particular context

Function syntax: $.proxy *(continued)*

Parameters

`function`	(Function) The function whose context will be changed
`proxy`	(Object) The object to which the context (`this`) of the function should be set
`argument`	(Any) An argument to pass to the function referenced in the `function` parameter
`property`	(String) The name of the function whose context will be changed (should be a property of the `proxy` object)

Returns

The new function whose context is set to the proxy object

Bring up the example in the file chapter-9/ $.proxy.html and you'll see the display shown in figure 9.7.

In this example page you have a Test button, whose ID is `test-button`, which you store in a variable as follows:

```
var $button = $('#test-button');
```

When the Normal radio button is clicked, a click handler is established on the Test button and its container:

```
$button.click(customLog);
```

Figure 9.7   The `$.proxy` example page will help you see the difference between normal and proxied callbacks.

The `customLog()` handler shows on the screen the ID of the function context (whatever is referred to by `this`):

```
function customLog() {
    $('#log').prepend(
      '<li>' + this.id + '</li>'
    );
}
```

When the button is clicked, you'd expect the established handler to have as the function context the element upon which it was established: the Test button. The result of clicking the Test button is shown in figure 9.8.

But when the Proxied radio button is clicked, the handler is established as follows:

Figure 9.8   The result of the `$.proxy` example page when using the normal handler

```
$button.click($.proxy(customLog, $('#control-panel').get(0)));
```

This establishes the same handler as before, except that the handler function has been passed through the `$.proxy()` utility function, prebinding an object to the handler. In this case you bind the element with the ID of `control-panel`. The bound object doesn't have to be an element—in fact, most often it won't be. We chose it for

this example because it makes the object easy to identify via its ID.

Now, when you click the Test button, you obtain the result shown in figure 9.9, which proves how the function context has been forced to be the object that you bound to the handler with `$.proxy()`.

This ability is useful for providing data to a callback that it might not normally have access to via closures or other means.

The most common use case for `$.proxy()` is when you want to bind a method of an object as a handler and have the method's owning object established as the handler's function context exactly as if you had called the method directly. Consider an object such as this:

```
var obj = {
  id: 'obj',
  hello: function() { alert('Hi there! I am ' + this.id); }
};
```

If you were to call the `hello()` method via `obj.hello()`, the function context (`this`) would be `obj`. But if you establish the function as a handler like so,

```
$(whatever).click(obj.hello);
```

you'll find that the function context is the current bubbling element, not `obj`. And if your handler relies on `obj`, you're rather screwed. You can get out of this predicament by using `$.proxy()` to force the function context to be `obj` with the following statement:

```
$(whatever).click($.proxy(obj.hello, obj));
```

Alternatively, using the second signature of the function, you can achieve the same goal with

```
$(whatever).click($.proxy(obj, 'hello'));
```

where `obj` is the object to which the context (`this`) of the function should be set and the string `"hello"` represents the name of the function that belongs to `obj`, whose context will be changed.

Be aware that going this route means that you won't have any way of knowing the current bubble element of the event propagation—the value normally established as the function context.

9.4.4 Evaluating expressions

Although the use of `eval()` isn't recommended by many developers, there are times when it's useful (take a look at the code of the Parsing Functions Lab page for an example).

$.proxy() Example

Establish handler as: ○ Normal ● Proxied

[Test]

Log

1. control-panel

Figure 9.9 This example shows the effects of prebinding an object to the click handler for the Test button.

The problem is that `eval()` executes in the current context. When writing plugins and other reusable scripts, you might want to ensure that the evaluation always takes place in the global context. Enter the `$.globalEval()` utility function.

Function syntax: $.globalEval

`$.globalEval(code)`

Evaluates the passed JavaScript code in the global context

Parameters

   `code`   (String) The JavaScript code to be evaluated

Returns

The evaluation of the JavaScript code

We'll wrap up our investigation of the utility functions with one that you'll use when learning how to write jQuery plugins but that you might use in other situations, too.

9.4.5   *Throwing exceptions*

Under some circumstances you'll want to throw an error in a function or a plugin you've authored. For example, you may want to throw an error if a developer passes to your plugin an unexpected parameter. For this purpose, jQuery provides a utility function called `$.error()`.

Function syntax: $.error

`$.error(string)`

Takes a string and throws an exception containing it

Parameters

   `string`   (String) A string specifying the message to send out

Returns

`undefined`

This method exists primarily for plugin developers who wish to override it and provide a better display (or more information) for the error messages. A simple example of use of this function is the following:

```
function isPrime(number) {
   if (typeof number !== 'number') {
      $.error('The argument provided is not a number');
   }
   // Remaining code here...
}
```

We'll use this function again when we discuss how to develop plugins for jQuery, but that's another chapter.

9.5   *Summary*

In this chapter we surveyed the features that jQuery provides outside of the methods that operate on a jQuery object. 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).

First, you learned about the flags that deal with animations. Setting `$.fx.off` lets you completely disable animations on your website so that changes will happen immediately. Then we introduced `$.fx.interval`, a flag to change the smoothness with which animations run.

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 `$`.

jQuery also provides a set of functions that are useful for dealing with data sets in arrays. `$.each()` makes it easy to traverse through items in collections; `$.grep()` allows you to create new arrays by filtering the data using whatever filtering criteria you'd like to use; `$.map()` allows you to easily apply your own transformations to a source to produce a corresponding new array with the transformed values.

You can also use jQuery to test if a value is in an array with `$.inArray()` and even test if a value is an array itself with `$.isArray()`. You can also test for functions using `$.isFunction()` or check the type of an object using the `$.type()` utility.

Another set of functions that the library offers allows you to deal with different formats. Two of them, `$.parseJSON()` and `$.parseXML()`, parse the two best known formats used on the web to exchange information: JSON and XML. The remaining one, `$.parseHTML()`, lets you parse HTML markup.

In addition to a bunch of minor functions, you also discovered how to merge objects using jQuery's `$.extend()` function. This function allows you to merge the properties of any number of source objects into a target object. Finally, the `$.proxy()` function lets you change the function context of any function in your code.

With these additional tools safely tucked away in your toolbox, you're ready to tackle the utility functions that jQuery provides to perform Ajax requests.

10

Talk to the server with Ajax

This chapter covers

- A brief overview of Ajax
- Loading preformatted HTML from the server
- Making GET and POST requests
- Exerting fine-grained control over requests
- Setting default Ajax properties
- Handling Ajax events

Ajax is one of the technologies that has heavily transformed the landscape of the web. The ability to make asynchronous requests back to the server without the need to reload entire pages has enabled a whole new set of user-interaction paradigms and made DOM-scripted applications possible.

A few years after Microsoft introduced Ajax, a handful of events launched it 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 technologies 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 means by which we can enable DOM-scripted 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 10.2), and then we'll look at how jQuery makes using Ajax a snap. Let's start off with a refresher on what Ajax technology is all about.

10.1 Brushing up on Ajax

Although we'll take a quick look at Ajax in this section, this isn't intended as a complete Ajax tutorial or an Ajax primer. If you're completely unfamiliar with Ajax (or worse, 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 toward teaching you all about Ajax.

Some people may argue that the term *Ajax* applies to any method of making 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 `XMLHttpRequest` (XHR) or the Microsoft XMLHTTP ActiveX control. A diagram of the overall process, which we'll examine one step at a time, is shown in figure 10.1.
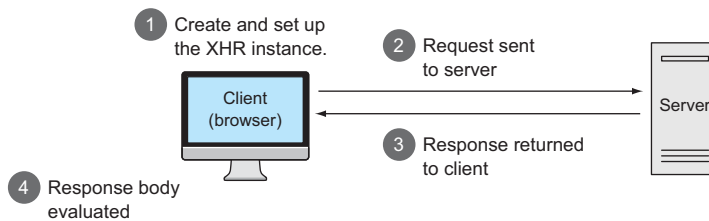
Figure 10.1 The lifecycle of an Ajax request as it makes its way from the client to the server and back again

Let's take a look at how those objects are used to generate requests to the server, beginning with creating an XHR instance.

10.1.1 Creating an XHR instance

In a perfect world, computer code would work in all commonly used browsers, but, as you've already learned, we don't live in such a world. Things are no different when it comes to Ajax. There's a standard way to make asynchronous requests via the JavaScript XHR object, and there's an (old) Internet Explorer proprietary way that uses an ActiveX control. With Internet Explorer 7, a wrapper that emulates the standard interface is available, but IE 6 requires divergent code.

NOTE jQuery's Ajax implementation—which we'll address throughout the remainder of this chapter—uses the ActiveX object when available. This is good news for us! By using jQuery for our Ajax needs, we know that the best approaches have been researched and will be utilized. If you don't need to support old versions of Internet Explorer, your job will be much easier.

Once created, 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 of *feature detection* that we introduced in the previous chapter. The code in the following listing shows a typical idiom used to instantiate an instance of XHR using this technique.

Listing 10.1   Capability detection resulting in code that can use Ajax in many browsers

```
var xhr;
if (window.ActiveXObject) {                           Tests for the presence
  xhr = new ActiveXObject('Microsoft.XMLHTTP');        of ActiveX

} else  if (window.XMLHttpRequest) {                  Tests for the
  xhr = new XMLHttpRequest();                           presence of XHR

} else {                                              Throws an
  throw new Error('Ajax is not supported by this browser');   error if there's
}                                                      no Ajax support
```

Once created, 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 10.1, and the most commonly used of these will be discussed in the sections that follow.

Table 10.1   `XMLHttpRequest` (XHR) methods and properties

Methods	Description
`abort()`	Causes the currently executing request to be cancelled.
`getAllResponseHeaders()`	Returns a single string containing the names and values of all response headers, or `null` if no response has been received.
`getResponseHeader(name)`	Returns the string containing the text of the specified header, or `null` if either the response has not yet been received or the header doesn't exist in the response.
`open(method, url[, async[, username[, password]]])`	Sets the HTTP method (such as `GET` or `POST`) and the 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.
`overrideMimeType(mime)`	Sets the Content-Type header for the response to `mime`.
`send([content])`	Initiates the request. The optional `content` parameter provides the request body. `content` is ignored if request method is GET or HEAD.
`setRequestHeader(name, value)`	Sets a request header using the specified name and value.

Table 10.1 `XMLHttpRequest` (XHR) methods and properties *(continued)*

Properties	Description
`onreadystatechange`	The event handler to be invoked when the state of the request changes.
`readyState`	An integer value that indicates the current state of the request as follows: `0` = UNSENT `1` = OPENED `2` = HEADERS_RECEIVED `3` = LOADING `4` = DONE
`response`	The response entity body according to `responseType`.
`responseText`	The body content returned in the response.
`responseType`	Can be set to change the response type. Its value can be one of `""` (empty string), `arraybuffer`, `blob`, `document`, `json`, or `text`.
`responseXML`	If the body content is identified as XML, the XML DOM is created from the body content.
`status`	The response status code returned from the server. For example: `200` for *success* or `404` for *not found*. See the HTTP specification[a] for the full set of codes.
`statusText`	The status text message returned by the response.
`timeout`	The number of milliseconds a request can take before being forced to terminate. The default value is `0`, which means there is no timeout.
`ontimeout`	The event handler to be called when the request times out.
`upload`	The upload process can be tracked by adding an event listener to `upload`.
`withCredentials`	Indicates whether or not cross-site `Access-Control` requests should be made using credentials such as cookies or authorization headers. The default is `false`.

a. HTTP 1.1 status code definitions from RFC 2616:
http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.

> **NOTE** Want to get it from the horse's mouth? You can find the XHR specification at http://www.w3.org/TR/XMLHttpRequest/.

Now that you have an XHR instance created, let's look at what it takes to set up and fire off the request to the server.

10.1.2 *Initiating the request*

Before you can send a request to the server, you need to perform the following 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 you of its progress.
4 Provide any body content for requests such as POST.

You set up the first two items by calling the `open()` method of an XHR object as follows:

```
xhr.open('GET', '/some/resource/url');
```

Note that this method doesn't 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 whether the request is to be asynchronous (`true`, which is the default) or synchronous (`false`). There's seldom a good reason to make the request synchronous (even if doing so means you 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.

In the third step, you must provide a means for the XHR instance to tap you on the shoulder to let you know what's going on. You 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 other properties of XHR, you 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 final steps to initiate the request are providing any body content for requests such as POST and sending it off to the server. Both of these steps are accomplished via the `send()` method. For GET or HEAD requests, which typically have no body, no body content parameter is passed, as follows:

```
xhr.send();
```

When request parameters are passed to the other requests type, the string passed to the `send()` method must be in the proper format (which you might think of as *query string* format) in which the names and values are 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 you), but if you're curious, in JavaScript you can use `encodeURIComponent()`.

An example of such a call is as follows:

```
xhr.send('a=1&b=2&c=3');
```

Now let's see what the ready state handler is all about.

10.1.3 *Keeping track of progress*

An XHR instance informs you 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 10.1). That's nice, but more often than not, you're only interested in when the request completes and whether it was successful or not. Frequently you'll see ready handlers implemented using the idiom shown in the next listing.

Listing 10.2   Ready state handlers written to ignore all but the DONE state

```
xhr.onreadystatechange = function() {            Ignores all but DONE state
  if (this.readyState === 4) {           ◁
    if (this.status >= 200 && this.status < 300) {   ◁── Branches on response status
      // Success                     ◁
    } else {
      // Problem       ◁── Executes on failure        Executes on success
    }
  }
}
```

This code ignores all but the DONE state (state with value of 4), and once that has been detected, it examines the value of the `status` property to determine whether 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 failure or redirection.

Now let's explore dealing with the response from a completed request.

10.1.4 *Getting the response*

Once the ready handler has determined that the `readyState` is complete and that the request completed successfully, you can retrieve the body of the response 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. The response to Ajax requests could be plain text, an HTML fragment, or any data represented using the 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 `application/xml` or a MIME type that ends with `+xml`, 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.

At this point, you might want to review the diagram of the whole process shown in figure 10.1. In this short overview of Ajax, we've identified the following pain points that page authors using Ajax need to deal with:

- Instantiating an XHR object requires browser-specific code.
- Ready handlers need to sift through a lot of uninteresting state changes.
- The response body needs to be dealt with in numerous ways, depending on its format.

The remainder of this chapter will describe how jQuery's Ajax methods and utility functions make Ajax a lot easier (and cleaner) to use on your pages. There are a lot of choices in the jQuery Ajax API, and we'll start with some of the simplest and most-used tools.

10.2   *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.

Setting up for the examples

Unlike most 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 Ajax 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 skip the setup of the server.

The code that we'll use on the server side is developed in PHP, so your server should be able to process it. In case you've never done this, here's a list of tools to start with regardless of your operating system:

- Windows users: http://www.wampserver.com/en
- Mac users: https://www.mamp.info

If you're used to another language, such Java or ASP .NET, you should be able to port the code in your language of choice because the pages are very simple. If you decide to convert the pages to Java or ASP.NET, you should also set up a web server that's able to understand these languages, like Tomcat and IIS, respectively. Here's a list of resources that will help you with this process:

- Tomcat for Windows and Linux users: https://tomcat.apache.org/tomcat-7.0-doc/setup.html
- Tomcat for Mac users: http://serverfault.com/questions/183496/how-do-i-start-apache-tomcat-at-boot-on-mac-os-x
- IIS for Windows users: http://www.iis.net/learn/install/installing-iis-7/installing-iis-on-windows-vista-and-windows-7

Let's imagine that you want to grab a chunk of HTML from the server using a resource named some-resource and make it the content of a <div> element with an ID of elem. For the final time in this chapter, let's look at how you'd do this without jQuery's assistance.

Using the patterns set out earlier in this chapter, you can write the code shown in listing 10.3. The full HTML file for this example can be found in the file chapter-10/listing.10.3.html.

> **NOTE** Again, you must run this example using a web server—you can't just open the file in the browser—so the URL should be http://localhost:8080/chapter-10/listing.10.3.html. Omit the port specification of :8080 if you're using Apache and leave it in if you're using Tomcat. In future URLs in this chapter we'll use the notation [:8080] to indicate that the port number might or might not be needed, but be sure not to include the square brackets as part of the URL.

Listing 10.3 Using native XHR to fetch and include an HTML fragment

```
var xhr;
if (window.ActiveXObject) {
   xhr = new ActiveXObject('Microsoft.XMLHTTP');
} else if (window.XMLHttpRequest) {
   xhr = new XMLHttpRequest();
} else {
   throw new Error('Ajax is not supported by this browser');
}
xhr.onreadystatechange = function() {
   if (this.readyState === 4) {
      if (this.status >= 200 && this.status < 300) {
         document.getElementById('elem').innerHTML = this.responseText;
      }
   }
}
xhr.open('GET', 'some-resource');
xhr.send();
```

Although there's nothing tricky going on here, that's a nontrivial amount of code (17 lines). The equivalent code you'd write using jQuery is as follows:

```
$('#elem').load('some-resource');
```

We're betting that we know which code you'd rather write and maintain!

Let's now take a close look at the jQuery method used in this statement.

10.2.1 *Loading content with jQuery*

The simple jQuery statement at the end of the previous section loads content from a server-side resource using one of the most basic but useful jQuery Ajax methods: load(). The full syntax description of this method is as follows.

Method syntax: load

`load(url[, data][, callback])`

Performs an Ajax request to the specified URL passing optional data. A callback function can be specified that's invoked when the request completes and the DOM has been modified. The response text replaces the content of all matched elements.

Parameters

`url`	(String) The URL of the server-side resource to which the request is sent, optionally modified via selector (explained below).
`data`	(String|Object|Array) Specifies any data that's to be passed as request parameters. This parameter can be a string that will be used as the query string or response body, an object whose properties are serialized, or an array of objects whose `name` and `value` properties specify the name/value pairs. If specified as an object or as an array, the request is made using the `POST` method. If omitted or specified as a string, the GET method is used.
`callback`	(Function) An optional 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, a status string (usually `"success"`), and the `jqXHR` instance (explained in a few moments). This function will be invoked once for each element in the jQuery collection with the target element set as the function context (`this`).

Returns

The jQuery collection.

In the description of this method we've introduced a new object called `jqXHR`. This name is an abbreviation for *jQuery XMLHTTPRequest*, which is a superset of the `XMLHTTPRequest` (XHR) object. For example, it contains the `responseText` and `responseXML` properties, as well as a `getResponseHeader()` method. It implements the `Promise` interface that we'll discuss in detail in chapter 13.

Though simple to use, this method has some important nuances. For example, when the `data` parameter is used to supply the request parameters and the argument passed is an object, the request is made using the POST HTTP method; otherwise, a GET request is initiated. If you want to make a GET request with parameters, you can include them as a query string on the URL. But be aware that when you do so, you're responsible for ensuring that the query string is properly formatted and that the names and values of the request parameters are URI-encoded. The JavaScript `encode-URIComponent()` method is handy for this, or you can employ the services of the jQuery `$.param()` utility function that we covered in chapter 9.

Sometimes you need to perform an action just after you've injected content into one or more elements. Let's say that you're polling the server to have updates of the status of the London underground and that you want to show a message on the web page each time an update is retrieved. For the sake of the example, you'll repeat the request one second after the callback function has been executed. A basic implementation that satisfies this request is shown here:

```
var updates = 1;
function pollInfo() {
```

```
$('#container').load(
    '/check-updates',
    function(responseText, textStatus, jqXHR) {
        if (textStatus === 'success') {
            $('#status-update').text('Data updated. Update #' + updates);
            updates++;
        }
        setTimeout(pollInfo, 1000);
    }
);
}
pollInfo();
```

⟵ **Invokes the pollInfo() function after 1000 milliseconds**

⟵ **Invokes pollInfo() the first time**

Most of the time, you'll use the `load()` method to inject the complete response into whatever elements are contained within the jQuery object, but sometimes you may want to filter elements coming back as the response. To do that, jQuery allows you to specify a selector that will be used to limit which response elements are injected into the elements in the set. You can specify the selector by suffixing the URL with a space followed by the selector itself.

For example, to filter response elements so that only `<div>` instances are injected, you have to write the following:

```
$('.inject-me').load('/some-resource div');
```

The selector used can be arbitrarily complex, which means that you can also write statements like this:

```
$('.inject-me').load('/some-resource div .some-class a');
```

In this case jQuery will search for all the elements that match the selector specified.

When it comes to supplying the data to be submitted with a request, sometimes you'll wing it with ad hoc data, but frequently you'll find yourself wanting to gather data that a user has entered into form controls.

As you might expect, jQuery has some assistance up its sleeve.

SERIALIZING FORM DATA

If the data that you want to send as request parameters comes from form controls, a helpful jQuery method for building a query string is `serialize()`, whose syntax is as follows.

Method syntax: serialize

`serialize()`

Creates a properly formatted and encoded query string from all form elements in the jQuery collection

Parameters

Returns

The formatted query string

The `serialize()` method is smart enough to collect information only from the form control elements in the set of matched elements 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.[1] Controls such as unchecked check boxes and radio buttons, dropdowns with no selections, and disabled controls aren't considered successful and don't participate in form submission, so they're also ignored by `serialize()`.

The ability to serialize values to send them to the server is a nice addition to jQuery, but unfortunately our lovely library doesn't provide a utility to perform the opposite operation: deserialize.

Deserialization is the operation of populating and changing the state of form fields based on a serialized string. It's useful if you have a complex search form and you want to store some commonly performed searches of a given user. In this case you can save the values of the fields in a cookie or database so that the user can click a button and restore those values without filling the form over and over again.

When jQuery fails to offer a solution, the vibrant community around the project takes care of it. That's another reason why jQuery is so awesome. For situations like the one described previously, you can employ a jQuery plugin called jQuery.deserialize (https://github.com/kflorence/jquery-deserialize).

Deserialize data with jQuery.deserialize

To use jQuery.deserialize you first need to include it in your page. The method is the same one seen many times in this book—placing a link to the library inside a `<script>` tag after the jQuery library, as shown here:

```
<script href="path/to/jquery.js"></script>
<script href="path/to/jquery.deserialize.js"></script>
```

With the plugin in place, let's say that you have the following form:

```
<form id="my-form">
    <input name="name" />
    <input name="surname" />
</form>
<button id="btn">Auto fill</button>
```

You also have the following string that comes from a previous serialization:

```
name=Aurelio&surname=De+Rosa
```

With this premise in mind, if you want to autofill the form as soon as the user clicks the `btn` button, you can write

```
$('#btn').click(function() {
    $('#my-form').deserialize('name=Aurelio&surname=De+Rosa');
});
```

---

[1] HTML 4.01 Specification, section 17.13.2, "Successful controls": http://www.w3.org/TR/html401/interact/forms.html#h-17.13.2.

If you'd rather get the form data in a JavaScript array (as opposed to a query string), jQuery provides the `serializeArray()` method.

Method 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

Returns

The array of form data

The array returned by `serializeArray()` is composed of object literals, each of which contains a `name` property and a `value` property that contain the name and value of each successful form control. Note that this is (not accidentally) one of the formats suitable for passing to the `load()` method to specify the request parameter data.

With the `load()` method at your disposal, let's put it to work solving some common real-world problems that many web developers encounter.

10.2.2 *Loading dynamic HTML fragments*

In business applications, particularly for commerce websites, often you'll want to grab real-time data from the server in order to present your users with the most up-to-date information. After all, you wouldn't want to mislead customers into thinking that they can buy something that's not available, would you?

In this section, you'll begin to develop a page that you'll add to throughout the course of the chapter. This page is part of a website 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. It will be important for you to always make sure that you're displaying the latest info!

To begin your page (which will omit site navigation and other boilerplate to concentrate on the lessons at hand), you want to present your customers with a dropdown containing the styles that are currently available and, when a style is selected, display detailed information regarding that style. On initial display, the page will appear as shown in figure 10.2.

Figure 10.2   The initial display of your commerce page with a simple dropdown inviting customers to click it

After the page first loads, a dropdown with the list of styles currently available in the inventory is displayed. When no style is selected, you'll display a helpful message as a placeholder for the selection: "- choose a style -". This invites the user to interact with the dropdown, and when a user selects a boot style from this dropdown, here's what you want to do:

- Display the detailed information about that style in the area below the drop-down.
- Remove the "- choose a style -" entry; once the user picks a style, it has served its purpose and is no longer useful.

Let's start by taking a look at the HTML markup for the body that defines this page structure:

```
<body>
   <div id="banner"></div>

   <h1>Choose your boots</h1>
   <div>                                            ❶ Contains selection control
      <div id="selections-pane">              ⟵
         <label for="boot-chooser-control">Boot style:</label>
         <select id="boot-chooser-control" name="model"></select>
      </div>
      <div id="product-detail-pane"></div>    ⟵❷ Holds place for product details
   </div>
</body>
```

Not much to it, is there? As would be expected, you define all the visual rendition information in an external style sheet (not shown here), and adhering to the precepts of unobtrusive JavaScript, you include no behavioral aspects in the HTML markup.

The most interesting parts of this markup are a container ❶ that holds the select element that will allow customers to choose a boot style and another container ❷ into which product details will be injected.

Note that the boot style control needs to have its option elements added before the user can interact with the page. Let's add the necessary behavior to this page. The first thing you'll add is an Ajax request to fetch and populate the boot style dropdown.

NOTE   Under most circumstances, initial values such as these would be handled on the server prior to sending the HTML to the browser. This means that even if your users have JavaScript disabled or can't execute JavaScript code, they can still use the web page. There may be circumstances where prefetching data via Ajax may be appropriate, but we're doing that here for instructional purposes only.

To add the options to the boot style control, you use the handy `load()` method:

```
$('#boot-chooser-control').load('actions/fetch-boot-style-options.php');
```

How simple is that? The only complicated part of this statement is the URL, which isn't all that long or complicated, that specifies a request to a server-side PHP page.

One of the nice things about using Ajax (with the ease of jQuery making it even nicer) is that it's completely independent of the server-side technology. You make HTTP requests, sometimes with appropriate parameter data, and as long as the server returns the expected responses, you can ignore whether the server is powered by Java, Ruby, PHP, or even old-fashioned CGI.

In this particular case, you expect that the server-side resource will return the HTML markup representing the boot style options—supposedly from the inventory database. The faux backend code returns the following as the response:

```
<option value="">- choose a style -</option>
<option value="7177382">Caterpillar Tradesman Work Boot</option>
<option value="7269643">Caterpillar Logger Boot</option>
<option value="7332058">Chippewa 9" Briar Waterproof Bison 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>
```

This response then gets injected into the `select` element, resulting in a fully functional control.

Your next act is to instrument the dropdown so that it can react to changes, carrying out the duties that we listed earlier. The code for that is only slightly more complicated:

```
$('#boot-chooser-control').change(function(event) {        ◁────┐  ❶ Establishes event
    $('#product-detail-pane').load(                                  handler for change
         'actions/fetch-product-details.php',
         {                                                        ❷ Fetches and displays product
            model: $(event.target).val()                            detail sending to the server
         },                                                          the chosen model
         function() {
            $('[value=""]', event.target).remove();            ❸ Removes the
         }                                                        placeholder option
    );
});
```

In this code, you select the boot style dropdown and bind a `change` handler to it ❶. In the event handler for the `change` event, which will be invoked whenever a customer changes the option of the dropdown selected, you obtain the current value of the selection by calling jQuery's `val()` method on the event target after you've wrapped it using `$()`. In this case, the target element is the `select` element that triggered the event.

You employ the `load()` method ❷ on the `product-detail-pane` element to send a request to the page `actions/fetch-product-details.php`. To this page you send the boots model by using an object literal whose only property is named `model`. Finally, you remove the placeholder option ❸ inside the callback of the `load()` method.

After the customer chooses an available boot style, the page will appear as shown in figure 10.3.

The most notable operation performed is the use of the `load()` method to quickly and easily fetch snippets of HTML from the server and place them within the DOM as the children of existing elements. This method is extremely handy and well suited to web applications that are powered by servers capable of server-side templating.

Figure 10.3   The server-side resource returns a preformatted fragment of HTML to display the detailed boot information.

The following listing shows the complete code for the Boot Closet page, which can be found at http://localhost[:8080]/chapter-10/phase.1.html. You'll revisit this page to add further capabilities to it as you progress through this chapter.

Listing 10.4 The first phase of the Boot Closet commerce page

```html
<!DOCTYPE html>
<html>
    <head>
        <title>The Boot Closet - Phase 1</title>
        <link rel="stylesheet" href="../css/main.css" />
        <link rel="stylesheet" href="../css/bootcloset.css">
    </head>
    <body>
        <div id="banner"></div>

        <h1>Choose your boots</h1>
        <div>
            <div id="selections-pane">
                <label for="boot-chooser-control">Boot style:</label>
                <select id="boot-chooser-control" name="model"></select>
            </div>
            <div id="product-detail-pane"></div>
        </div>

        <script src="../js/jquery-1.11.3.min.js"></script>
        <script>
            $('#boot-chooser-control')
                    .load('actions/fetch-boot-style-options.php')
                    .change(function(event) {
                        $('#product-detail-pane').load(
                                'actions/fetch-product-details.php',
                                {
                                    model: $(event.target).val()
                                },
                                function() {
                                    $('[value=""]', event.target).remove();
                                }
                        );
                    });
        </script>
    </body>
</html>
```

The load() method is tremendously useful when you 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 you either want more control over how the Ajax request gets made or need to do something more complex with the returned data in the response body.

Let's continue our investigation of what jQuery has to offer for these more complex situations.

10.3   *Making GET and POST requests*

The `load()` method makes either a GET or a POST request, depending on how the request parameter data (if any) is provided, but sometimes you want to have a bit more control over which HTTP method gets used. Why should *you* care? Because your *server* may care.

   Web authors have traditionally played fast and loose with the GET and POST methods, using one or the other without heeding how HTTP intends for these methods to be used. The intentions for each method are as follows:

- *GET requests*—Intended to be *idempotent*; 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 or updating records in a database or removing information from the server.

A GET request should, therefore, be used whenever the purpose of the request is to merely get data, as its name implies. It may be required to send some parameter 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 on the HTTP method used, and GET requests are highly subject to caching. Using the proper HTTP method ensures that you don't get crossways with the browser's or server's expectations regarding the intentions of the requests. This is just a glimpse into the realm of RESTful principles, where other HTTP methods such as PUT and DELETE also come into play. But for our purposes, we'll limit our discussion to the GET and POST methods.

With that in mind, if you look back to our phase-one implementation of The Boot Closet (in listing 10.4), you'll discover that *you're doing it wrong!* Because jQuery initiates a POST request when you supply an object hash for the `data` parameter, you're making a POST when you really should be making a GET request. If you glance at Chrome's Developer Tools log (as shown in figure 10.4), when you display your page in Chrome, you can see that your second request, submitted when you make a selection from the style dropdown, is indeed a POST.

> **NOTE**   The result you'll see on your console is the same as the figure only if you enable the option Log XMLHttpRequests. If you don't want to enable this option, you can take a look at the Network tab.

Does it really matter? That's up to you, but if you want to use HTTP in the manner in which it was intended, your request to fetch the boot detail should be a GET rather than a POST.
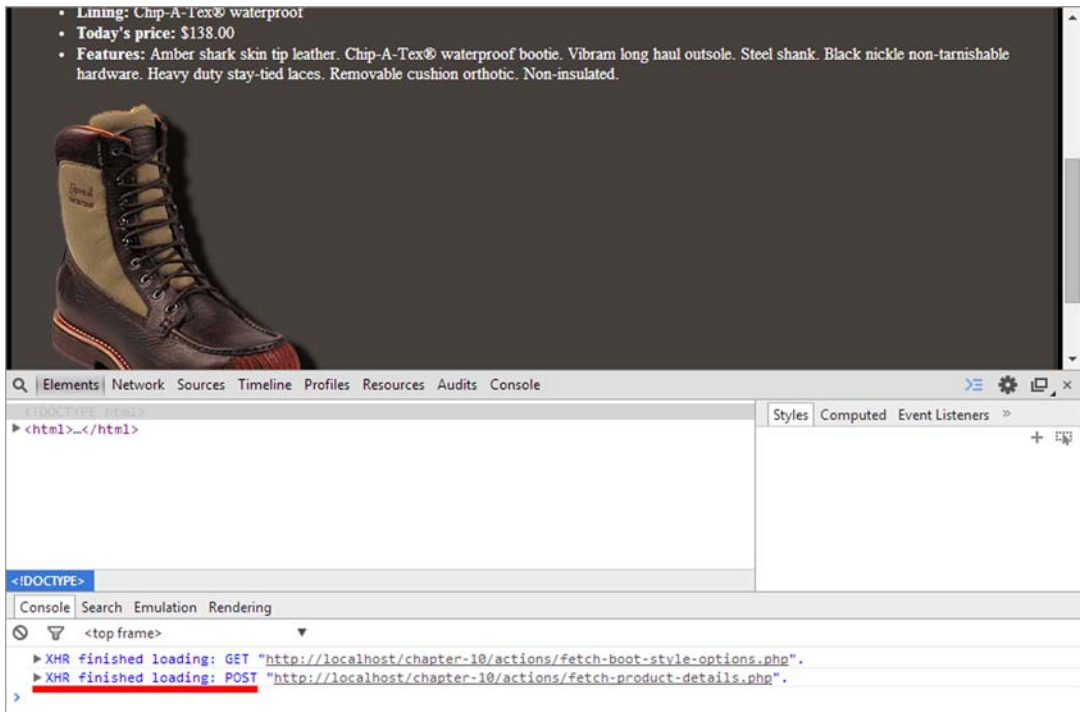
Figure 10.4   An inspection of the Chrome console shows that you're making a POST request when you should be making a GET.

Developer tools

Trying to develop a DOM-scripted application without the aid of a debugging tool is like trying to play concert piano while wearing welding gloves. Why would you do that to yourself?

Depending on the browser you're using, there are different options you can choose to inspect your code. All modern major browsers have a set of built-in tools for this purpose, each with a different name, that you can adopt. For example, in Chrome these tools are called *Developer Tools* (https://developer.chrome.com/devtools/), whereas in Internet Explorer they're called *F12 developer tools* (http://msdn.microsoft.com/en-us/library/bg182326(v=vs.85).aspx). Firefox has its own built-in tools as well, but developers usually tend to use a plugin called *Firebug* (http://getfirebug.com). These tools not only let you inspect the JavaScript console, but they also allow you to inspect the live DOM, the CSS, the script, and many other aspects of your page as you work through its development.

One feature most relevant for your current purposes is the ability to log Ajax requests along with both the request and response information.

You could make the parameter that specifies the request information a string rather than an object (and we'll revisit that a little later), but for now, let's take advantage of another way that jQuery lets you initiate Ajax requests.

10.3.1 *Getting data with GET*

jQuery gives you a few means to send GET requests, which, unlike `load()`, are implemented not as jQuery methods but as utility functions. These are the functions we mentioned but didn't cover in the previous chapter.

When you want to fetch some data from the server and decide what to do with it by yourself (rather than letting the `load()` method set it as the content of one or more elements), you can use the `$.get()` utility function.

Function syntax: $.get

`$.get(url[, data][, callback][, dataType])`
Sends a GET request to the server using the specified URL with any passed parameters as the query string.

Parameters

`url`	(String) The URL of the server-side resource to contact via the GET method. If an empty string is passed, the request is sent to the current URL at the time the method is invoked.
`data`	(String|Object) Specifies any data that's to be passed as request parameters in the query string. This parameter is optional and can be a string or an object whose properties are serialized into properly encoded parameters to be passed to the request.
`callback`	(Function) An optional function invoked when the request completes successfully. The response body is passed as the first parameter to this callback, interpreted according to the setting of the `dataType` parameter, and the status string is passed as the second parameter. A third parameter contains a reference to the `jqXHR` instance. Inside the callback, the context (`this`) is set to an object that represents the Ajax settings used in the call. This parameter becomes required if `dataType` is provided. In case you don't need a function, you can pass `null` or `$.noop()` as a placeholder.
`dataType`	(String) Optionally specifies how the response body is to be interpreted, and it can be one of the following values: `html`, `text`, `xml`, `json`, `script`, or `jsonp`. The default value is determined by jQuery depending on the response obtained and will be one among `xml`, `json`, `script`, or `html`. See the description of `$.ajax()` later in this chapter for more details.

Returns

The `jqXHR` instance.

> ### jQuery 3: Signature added
>
> jQuery 3 adds a new signature for the `$.get()` utility function:
>
> `$.get([options])`
>
> `options` is an object that can possess many properties. To learn more about it, please refer to the description of the `$.ajax()` utility function discussed later in this chapter. It's worth noting that the `method` property that the `options` object can contain will automatically be set to `"GET"`.

The `$.get()` utility function allows you to perform GET requests in a more versatile way. In addition to the request parameters and the callback to be invoked upon a successful response, you can even direct how the response is to be interpreted and passed to the callback. If even that's not enough versatility, you'll be see a more general function, `$.ajax()`, where you'll also examine the `dataType` parameter in greater detail. For now you'll let it default to `html` or `xml` depending on the content type of the response. By using `$.get()` in your Boot Closet page, you can replace the use of the `load()` method, as shown in the next listing.

Listing 10.5   Changing The Boot Closet to use a GET when fetching style details

```
$('#boot-chooser-control')
        .change(function(event) {
            $.get(                                        ◁———❶ Performs a GET request
                'actions/fetch-product-details.php',
                {
                    model: $(event.target).val()
                },
                function(response) {
                    $('#product-detail-pane').html(response);   ◁—— ❷ Injects the response HTML
                    $('[value=""]', event.target).remove();
                }
            );
        });
```

The changes for this second phase of our page are subtle but significant. You call `$.get()` ❶ in place of `load()`, passing the same URL and the same request parameters. Because you're using the `$.get()` utility function in this case, you can be assured that a GET request will be performed even if you pass an object. `$.get()` doesn't automatically inject the response anywhere within the DOM, so you need to do that yourself by using jQuery's `html()` method ❷.

The code for this version of our page can be found at http://localhost[:8080]/chapter-10/phase.2.html. Loading the page and selecting a style dropdown, you can see that a GET request has been made, as shown in figure 10.5.
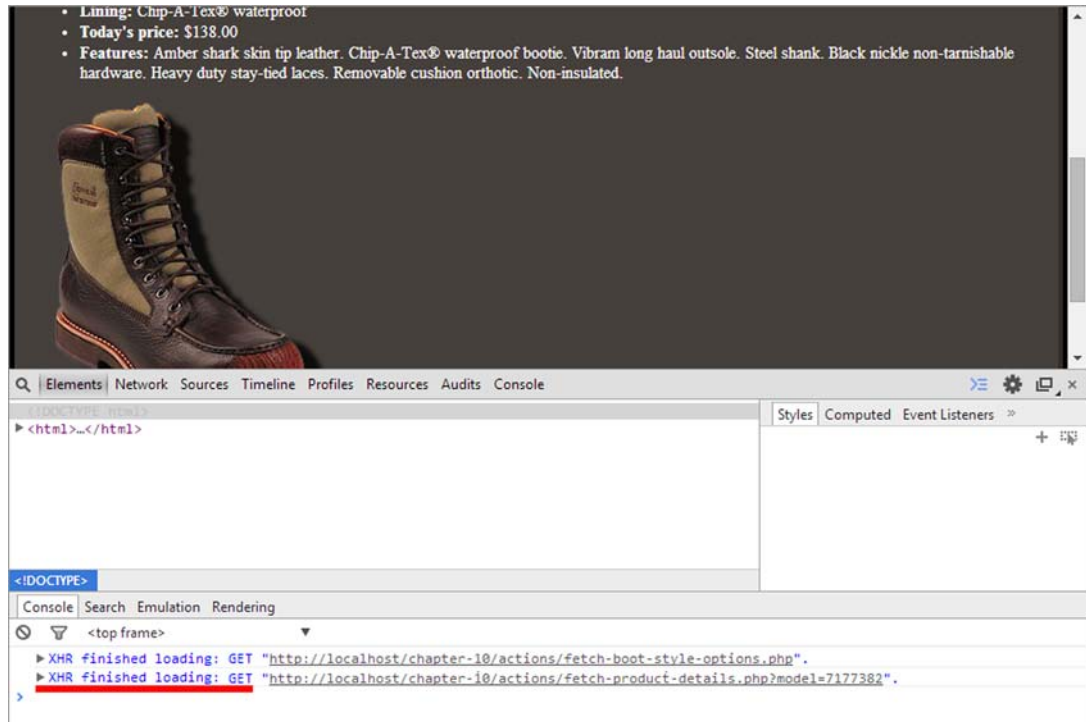
Figure 10.5   Now you can see that the second request is a GET rather than a POST, as befitting the operation.

In this example you returned HTML code from the server and inserted it into the DOM, but as you can see from the values available for the dataType parameter of $.get(), there are many possibilities other than HTML.

Let's look at another jQuery utility function that's quite useful when your data needs suggest that you should employ the JSON format.

10.3.2 *Getting JSON data*

When XML is overkill or otherwise unsuitable as a data-transfer mechanism, JSON is often used in its place. One reason for this choice is that JSON is astoundingly easy to digest in client-side scripts, and jQuery makes it even easier.

For times when you know that the response will be JSON, the $.getJSON() utility function automatically parses the returned JSON string and makes the resulting JavaScript value available to its callback. The syntax of $.getJSON(), shown next, has parameters with the same meaning described for $.get(), so we won't repeat them.

Function syntax: $.getJSON

`$.getJSON(url[, data][, callback])`

Sends 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 value is passed to the callback function.

Returns

The `jqXHR` instance.

As you can see from the description, this function is simply a convenience function for calling `$.get()` with a `dataType` of `"json"`.

$.getJSON() isn't the only one, as you'll discover in the next section.

10.3.3 *Dynamically loading script*

Most of the time, you'll load the external scripts that your page needs from script files when the page loads via `<script>` tags at the bottom of your page. But every now and then you might want to load a script under script control. You might do this because you don't know if the script will be needed until after some specific user activity has taken place, and you don't want to include the script unless it's absolutely needed. Or perhaps you need to use some information not available at load time to make a conditional choice between various scripts.

Regardless of why you might want to dynamically load new scripts into the page, jQuery provides the `$.getScript()` utility function to make it easy. This utility function also has parameters with the same meaning described for `$.get()`, so we won't repeat them.

Function syntax: $.getScript

`$.getScript(url[, callback])`

Fetches the script specified by the `url` parameter performing a GET request to the specified server, optionally invoking a callback upon success. The URL isn't restricted to the same domain as the containing page.

Returns

The `jqXHR` instance.

Under the covers, this function invokes `$.get()` by setting the `data` parameter to `undefined` and the `dataType` parameter to `"script"`. In the source of jQuery the `$.getScript()` utility function is defined as follows:

```
getScript: function( url, callback ) {
   return jQuery.get( url, undefined, callback, "script" );
}
```

When this function is executed, the script in the file is evaluated, any inline script is executed, and any defined variables or functions become available.

Let's see this in action. Consider the following script file (available in chapter-10/external.js):

```
alert('I am 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 the next listing and can be found in the file chapter-10/$.getScript.html.

Listing 10.6   Dynamically loading a script file and examining the results

```
<!DOCTYPE html>
<html>
   <head>
      <title>$.getScript() Example</title>
      <link rel="stylesheet" href="../css/main.css"/>
   </head>
   <body>
      <button id="load-button">Load</button>
      <button id="inspect-button">Inspect</button>          ❶ Defines test buttons

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         $('#load-button').click(function() {                  Fetches script on
            $.getScript('external.js');                  ❷ clicking Load button
         });

         $('#inspect-button').click(function() {              Displays result
            someFunction(someVariable);                        on clicking
         });                                            ❸ Inspect button
      </script>
   </body>
</html>
```

This page defines two buttons ❶ that you use to trigger two different actions. The Load button causes the "external.js" file to be dynamically loaded through the use of the `$.getScript()` utility function ❷. Click this button and, as expected, the inline statement within the file triggers an alert message, as shown in figure 10.6.

Clicking the Inspect button executes its `click` handler ❸, which executes the dynamically loaded `someFunction()` function, passing the value of the dynamically loaded `someVariable` variable. If the alert
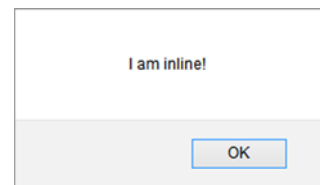
Figure 10.6   The dynamic loading and evaluation of the script file results in the inline alert statement being executed.

Figure 10.7 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.

appears as shown in figure 10.7, you know that both the variable and function are loaded correctly.

In addition to providing these utility functions to perform GET requests, jQuery also lets you make POST requests. Let's see how.

10.3.4 Making POST requests

There are a number of reasons why you might choose a POST over a GET. First, the intention of HTTP is that POST will be used for any non-idempotent requests. Therefore, if your request has the potential to cause a change in the server-side state, resulting in varying responses, it should be a POST. Moreover, 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 a URL in a query string—a limit that's a browser-dependent value. And sometimes the server-side resource you contact might perform different functions depending on whether your request uses the GET or POST method. These are just a few of the many reasons why you might want to choose a POST request over a GET request.

For those occasions when a POST is needed, jQuery offers the `$.post()` utility function. This utility function is identical to `$.get()` except for employing the POST HTTP method. For this reason, in the description of the syntax of the method, we won't repeat the meaning of the parameters. The syntax of `$.post()` is as follows.

Function syntax: $.post

`$.post(url[, data][, callback][, dataType])`
Sends a POST request to the server using the specified URL, with any parameters passed within the body of the request

Returns

The `jqXHR` instance

---

jQuery 3: Signature added

jQuery 3 adds a new signature for the `$.post()` utility function:

`$.post([options])`

`options` is an object that can possess many properties. To learn more about it, please refer to the description of the `$.ajax()` utility function discussed later in this chapter. It's worth noting that the `method` property that the `options` object can contain will automatically be set to `"POST"`.

jQuery takes care of the details of passing the request data in the request body (as opposed to the query string) and sets the HTTP method appropriately.

Now, getting back to the Boot Closet project, you've made a really good start, but there's more to buying a pair of boots than selecting a style; customers are sure to want to pick which color they like, and certainly they'll need to specify their size. We'll use these additional requirements to show how to solve one of the most-asked questions on the web, that of…

10.3.5 *Implementing cascading dropdowns*

The implementation of cascading dropdowns—where subsequent dropdown options depend on the selections of previous dropdowns—is one of the most used patterns on the web. In this section we're going to implement a solution on the Boot Closet page that demonstrates how ridiculously simple jQuery makes it.

You've already seen how easy it is to load a dropdown dynamically with server-powered option data, but now you'll see that tying multiple dropdowns together in a cascading relationship is only slightly more work.

Let's dig in by listing the changes you need to make in the next phase of your page:

- Add dropdowns for color and size.
- When a style is selected, add options to the color dropdown that show the colors available for that style.
- When a color is selected, add options to the size dropdown that show the sizes available for the selected combination of style and color.
- Make sure things remain consistent. This includes removing the "- please make a selection -" options from newly created dropdowns once they've been used and making sure that the three dropdowns never show an invalid combination.

You're also going to revert to using `load()` again, this time coercing it to initiate a GET rather than a POST. The reason is that `load()` seems more natural when you're using Ajax to load HTML fragments.

To start off, let's examine the new HTML markup that defines the additional dropdowns. A new container for the select elements is defined to contain three labeled elements:

```
<div id="selections-pane">
    <label for="boot-chooser-control">Boot style:</label>
    <select id="boot-chooser-control" name="model"></select>

    <label for="color-chooser-control">Color:</label>
    <select id="color-chooser-control" name="color" disabled></select>

    <label for="size-chooser-control">Size:</label>
    <select id="size-chooser-control" name="size" disabled></select>
</div>
```

Dropdown menu for the color, which is initially disabled

Dropdown menu for the model

Dropdown menu for the size, which is initially disabled

The previous model's `select` element remains, but it has been joined by two more: one for color and one for size, each of which is initially empty and disabled (by using the `disabled` attribute).

The style selection dropdown must now perform double duty. Not only must it continue to fetch and display the boot details when a selection is made, but its change handler must now also populate and enable the color-selection dropdown with the colors available for whatever style was chosen.

Let's refactor the fetching of the details first. You want to use `load()`, but you also want to force a GET, as opposed to the POST that you were initiating earlier. In order to have `load()` induce a GET, you need to pass a string rather than an object to specify the request parameter data. Luckily, with jQuery's help, you won't have to build that string yourself. The first part of the change handler for the style dropdown gets refactored like this:

```
var $bootChooser = $('#boot-chooser-control');        Defines variables used
var $colorChooser = $('#color-chooser-control');      throughout the code
var $sizeChooser = $('#size-chooser-control');

$bootChooser.change(function() {
   $('#product-detail-pane').load(
         'actions/fetch-product-details.php',
         $(this).serialize()                      ◁——— Provides data as a query string
   );
   // More to follow
});
```

By using jQuery's `serialize()` method, you create a string representation of the value of the style dropdown, thereby coercing the `load()` method to initiate a GET, just as you wanted.

The second duty that the change handler needs to perform is to load the color-choice dropdown with appropriate values for the chosen style and then enable it. Take a look at the rest of the code to be added to the handler:

```
$colorChooser.load(
      'actions/fetch-color-options.php',            ❶ Fetches and loads
      $(this).serialize(),                             color options
      function() {
        $(this).prop('disabled', false);          ◁——— ❷ Enables color control
        $sizeChooser
                .prop('disabled', true)           ❸ Disables and
                .html('');                            empties size control
      }
);
```

This code should look familiar. It's another use of `load()`, this time loading data from a page named `actions/fetch-color-options.php`, which is designed to return a set of formatted `<option>`s representing the colors available for the chosen style passed ❶.

You also specify a callback to be executed when the GET request successfully returns a response. In this callback, you perform two important tasks. First, you enable the color-chooser control ❷. The call to `load()` injects the `<option>` tags, but once populated, it would still be disabled if you didn't enable it. Second, the callback disables and empties the size-chooser control ❸. Why? (Pause a moment and think about it.)

Even though the size control will already be disabled and empty the first time the style chooser's value is changed, what about later on? What if, after the customer chooses a style and a color (which you'll soon see results of in the population of the size control), they change the selected style? The sizes displayed depend on the combination of style and color, so the sizes previously displayed are no longer applicable and don't reflect a consistent view of what's chosen. Therefore, whenever the style changes, you need to blow the size options away and reset the size control to initial conditions.

Before you sit back and enjoy a lovely beverage, you have more work to do. You still have to instrument the color-chooser dropdown to use the selected style and color values to fetch and load the size-chooser dropdown. The code to do this follows a familiar pattern:

```
$colorChooser.change(function() {
    $sizeChooser.load(
            'actions/fetch-size-options.php',
            $colorChooser
                    .add($bootChooser)
                    .serialize(),
            function() {
                $(this).prop('disabled', false);
            }
    );
});
```

Upon a `change` event in the color control, the size information is obtained via the `actions/fetch-size-options.php` page, passing both the boot style and color selections, and the size control is enabled.

There's one more thing that you need to do. When each dropdown is initially populated, it's seeded with an `option` element with a blank value and display text along the lines of "- choose a something -". You may recall that in the previous phases of this page, you added code to remove that option from the style dropdown upon selection.

Well, you could add such code to the change handlers for the style and color dropdowns and add instrumentation for the size dropdown (which currently has none) to add that. But let's be a bit more suave about it.

One capability of the event model that often gets ignored by many a web developer is *event bubbling*. Page authors frequently focus only on the targets of events and forget that events will bubble up the DOM tree, where handlers can deal with those events in more general ways than at the target level.

If you recognize that removing the option with a blank value from any of the three dropdowns can be handled in the exact same fashion regardless of which dropdown is the target of the event, you can avoid repeating the same code in three places by establishing a single handler, higher in the DOM, that will recognize and handle the change events. This trick should remind you of our discussion about event delegation in chapter 6.

Recalling the structure of the document, the three dropdowns are contained within a `<div>` element with an ID of `selections-pane`. You can handle the removal of the temporary option for all three dropdowns with the following single listener:

```
$('#selections-pane').change(function(event){
   $('[value=""]', event.target).remove();
});
```

This listener will be triggered whenever a `change` event happens on any of the enclosed dropdowns, and it will remove the `option` with the blank value within the context of the target of the event (which will be the changed dropdown).

With that, you've completed phase three of The Boot Closet, adding cascading dropdowns into the mix, as shown in figure 10.8. You can use the same techniques in any pages where dropdown values depend on previous selections. The page for this phase can be found at http://localhost[:8080]/chapter-10/phase.3.html.
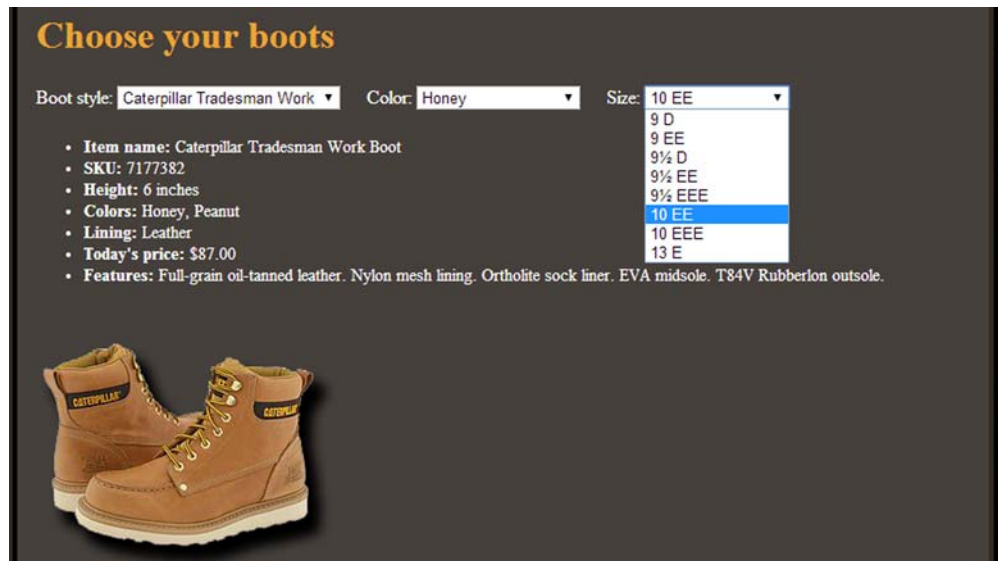
Figure 10.8   **The third phase of The Boot Closet shows how easy it is to implement cascading dropdowns.**

The full code of the page is now as shown in the following listing.

Listing 10.7   The Boot Closet, now with cascading dropdowns!

```
<!DOCTYPE html>
<html>
    <head>
        <title>The Boot Closet - Phase 3</title>
        <link rel="stylesheet" href="../css/main.css" />
        <link rel="stylesheet" href="../css/bootcloset.css" />
    </head>
    <body>
        <div id="banner"></div>

        <h1>Choose your boots</h1>
        <div>
        <div id="selections-pane">
            <label for="boot-chooser-control">Boot style:</label>
            <select id="boot-chooser-control" name="model"></select>
            <label for="color-chooser-control">Color:</label>
            <select id="color-chooser-control" name="color" disabled></select>
            <label for="size-chooser-control">Size:</label>
            <select id="size-chooser-control" name="size" disabled></select>
        </div>
        <div id="product-detail-pane"></div>

        <script src="../js/jquery-1.11.3.min.js"></script>
        <script>
            var $bootChooser = $('#boot-chooser-control');
            var $colorChooser = $('#color-chooser-control');
            var $sizeChooser = $('#size-chooser-control');

            $bootChooser
                    .load('actions/fetch-boot-style-options.php')
                    .change(function() {
                        $('#product-detail-pane').load(
                            'actions/fetch-product-details.php',
                            $(this).serialize()
                        );

                        $colorChooser.load(
                                'actions/fetch-color-options.php',
                                $(this).serialize(),
                                function() {
                                    $(this).prop('disabled', false);
                                    $sizeChooser
                                            .prop('disabled', true)
                                            .html('');
                                }
                        );
                    });

            $colorChooser.change(function() {
                $sizeChooser.load(
                        'actions/fetch-size-options.php',
                        $colorChooser
                                .add($bootChooser)
```

```
                               .serialize(),
                    function() {
                        $(this).prop('disabled', false);
                    }
                );
            });

            $('#selections-pane').change(function(event){
                $('[value=""]', event.target).remove();
            });
        </script>
    </body>
</html>
```

As you've seen, with the `load()` method and the various GET and POST jQuery Ajax functions at your disposal, you can exert some measure of control over how your request is initiated and how you're notified of its completion. But for those times when you need full control over an Ajax request, jQuery has a means for you to get as picky as you want.

10.4 *Taking full control of an Ajax request*

The functions and methods you've seen so far are convenient for many cases, but there may be times when you want to take control of all the nitty-gritty details into your own hands. For example, you may want to be sure that each time your Ajax request is performed, you'll receive fresh data (that is, avoid the browser cache). Another situation where the use of a lower-level method may come in handy is when you need to perform an Ajax request but its result is important only if retrieved within a certain amount of time. The last example we want to mention is that sometimes you could receive the result of the Ajax call in a certain format—for example, as plain text—but you prefer it to be converted into another one, such as HTML or XML.

In this section, we'll explore how jQuery lets you exert such dominion.

10.4.1 *Making Ajax requests with all the trimmings*

For those times when you want or need to exert fine-grained control over how you make Ajax requests, jQuery provides a general utility function for making Ajax requests: `$.ajax()`. Under the covers, all other jQuery features that make Ajax requests eventually use this function to perform the request. Its syntax is as follows.

Function syntax: $.ajax

`$.ajax(url[, options])`
`$.ajax([options])`

Performs an Ajax request using the URL and the options passed to control how the request is made and callbacks notified. In the second version of this utility function, the URL is specified in the options. If no parameters are specified, the request is made to the current page.

Parameters

`url`   (String) The string containing the URL to which the request is sent.

`options`	(Object) An object whose properties define the parameters for this operation. See table 10.2 for details.

Returns

The `jqXHR` instance.

Looks simple, doesn't it? But don't be deceived. The `options` parameter can specify a very large range of values that can be used to tune the operation of this function, including the URL to which to send the request. These options (in general order of their importance and the likelihood of their use) are defined in table 10.2.

Table 10.2   Options for the `$.ajax()` utility function

Name	Description
`url`	(String) The string containing the URL to which the request is sent. If an empty string is passed, the request is sent to the current URL at the time the method is invoked.
`method`	(String) The HTTP method to use. Usually either POST or GET. If omitted, the default is GET. If you're using versions of jQuery prior to 1.9.0, this same property must be named as `type` instead.
`data`	(String|Object|Array) Defines the values that will be sent to the server. If the request is a GET, the values are passed as the query string. If a POST, the values are passed as the request body. In either case, the encoding of the values is handled by the `$.ajax()` utility function. This parameter can be a string that will be used as the query string or response body, an object whose properties are serialized, or an array of objects whose `name` and `value` properties specify the name/value pairs.

`dataType`	(String) In its basic form, it's a keyword that identifies the type of data that's expected to be returned by the response. This value determines what, if any, postprocessing occurs upon the data before being passed to callback functions. The valid values are as follows:
	
	   `xml`—The response text is parsed as an XML document and the resulting XML DOM is passed to the callbacks.
	   `html`—The response text is passed unprocessed to the callback functions. Any `<script>` blocks within the returned HTML fragment are evaluated.
	   `json`—The response text is evaluated as a JSON string and the resulting object is passed to the callbacks.
	   `jsonp`—Similar to `json` except that remote scripting is allowed, assuming the remote server supports it.
	   `script`—The response text is passed to the callbacks. Prior to any callbacks being invoked, the response is processed as a JavaScript statement or statements.
	   `text`—The response text is assumed to be plain text.
	
	The server is responsible for setting the appropriate content-type response header. The default value is determined by jQuery depending on the response obtained and will be one among `xml`, `json`, `script`, or `html`.
	   The value of this option can also be a string of space-separated values. In this case, jQuery converts a data type into another. For example, if the response is text and you want it to be treated as XML, you can write `"text xml"`.

Table 10.2   Options for the `$.ajax()` utility function *(continued)*

Name	Description
cache	(Boolean) If `false`, ensures that the response won't be cached by the browser. Note that this works correctly only with HEAD and GET requests. Defaults to `true` except when `dataType` is specified as either `script` or `jsonp`.
context	(Object|Element) Specifies an object or DOM element that is to be set as the context of all callbacks related to this request. By default, the context is an object that represents the Ajax settings used in the call.
timeout	(Number) Sets a timeout for the Ajax request in milliseconds. The timeout period starts at the point the `$.ajax()` call is made. If the request doesn't complete before the timeout expires, the request is aborted and the error callback (if defined) is called.
global	(Boolean) If `false`, disables the triggering of *global Ajax events*. These are jQuery-specific custom events that trigger at various points or conditions during the processing of an Ajax request. We'll discuss them in detail in an upcoming section. If omitted, the default (`true`) is to enable the triggering of global events.
contentType	(String) The content type to be specified on the request. If omitted, the default is `application/x-www-form-urlencoded; charset=UTF-8`, the same type used as the default for form submissions.
success	(Function|Array) A function or an array of functions 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 evaluated according to the specification of the `dataType` property. The second parameter is a string containing a status value—in this case, always the string `"success"`. A third parameter provides a reference to the `jqXHR` instance.
error	(Function|Array) A function or an array of functions invoked if the response to the request returns an error status code. Three arguments are passed to this function: the `jqXHR` instance, a status message string (in this case, one of `"error"`, `"timeout"`, `"abort"`, or `"parseerror"`), and an optional exception object, sometimes returned from the `jqXHR` instance, if any. This handler is not called for cross-domain script and cross-domain JSONP requests.
complete	(Function|Array) A function or an array of functions called upon completion of the request. Two arguments are passed: the `jqXHR` instance and a status message string of `"success"`, `"error"`, `"notmodified"`, `"timeout"`, `"abort"`, or `"parseerror"`. If either a success or error callback is also specified, this function is invoked after that callback is called.
beforeSend	(Function) A function invoked prior to initiating the request. This function is passed the `jqXHR` instance and can be used to set custom headers or to perform other prerequest operations. Returning `false` from this handler will cancel the request.
async	(Boolean) If specified as `false`, the request is submitted as a synchronous request. By default, the value is `true` and the request is asynchronous. Cross-domain requests and `dataType: "jsonp"` requests do not support synchronous operation.

Table 10.2   Options for the `$.ajax()` utility function *(continued)*

Name	Description
`processData`	(Boolean) If set to `false`, prevents the data passed from being processed into URL-encoded format. By default, the value is `true` and the value of `data` is URL-encoded into a format suitable for use with requests of type `application/x-www-form-urlencoded`.
`contents`	(Object) An object of string/regular-expression pairs that determine how jQuery will parse the response, given its content type.
`converters`	(Object) An object containing `dataType`-to-`dataType` converters. Each converter's value is a function that returns the transformed value of the response.
`crossDomain`	(Boolean) Set it to `true` to force a `crossDomain` request on the same domain. By default, its value is `false` for same-domain requests and `true` for cross-domain requests.
`headers`	(Object) An object of additional header key/value pairs to send along with requests. By default, its value is an empty object.
`dataFilter`	(Function) A callback invoked to filter the response data. This function is passed the raw response data and the `dataType` value and is expected to return the "sanitized" data.
`ifModified`	(Boolean) If `true`, allows a request to succeed only if the response content has not changed since the last request, according to the `Last-Modified` header. If omitted, no header check is performed. Defaults to `false`.
`isLocal`	(Boolean) Allow the current environment to be recognized as *local* (for example, the filesystem). The protocols that jQuery recognizes as local by default are `file`, `*-extension`, and `widget`.
`jsonp`	(String) Specifies a query parameter name to override the default `jsonp` callback parameter name of `callback`.
`jsonpCallback`	(String|Function) Specifies the callback function name for a JSONP request. This value will be used instead of the random name automatically generated by jQuery.
`username`	(String) The username to be used in the event of an HTTP authentication request.
`password`	(String) The password to be used in the case of an HTTP authentication request.
`scriptCharset`	(String) The character set to be used for `script` and `jsonp` requests when the remote and local content are of different character sets.
`statusCode`	(Object) An object containing a set of numeric HTTP codes and functions to be called when the response has the corresponding code. By default, its value is an empty object.
`xhr`	(Function) A callback used to provide a custom implementation of the XHR instance.
`xhrFields`	(Object) An object of name-value pairs to set on the native XHR object. By default, the object is empty.
`accepts`	(Object) The content type sent in the request header that tells the server what kind of response it will accept in return. By default, its value depends on `dataType`.

Table 10.2 Options for the `$.ajax()` utility function *(continued)*

Name	Description
`mimeType`	(String) A mime type to override the XHR mime type.
`traditional`	(Boolean) If `true`, the traditional style of parameter serialization is used. See the description of `$.param()` in chapter 9 for details on parameter serialization.

Don't be scared by this list. We know it can be a bit overwhelming, but first of all, you don't have to remember all these options (this book and the official documentation serve this purpose), and second, it's unlikely that more than a few of them will be used for any one request.

What's JSONP all about?

JSON is a lightweight and heavily used data-interchange format. Websites usually retrieve data in such a format by performing Ajax requests using an XHR object. This mechanism abides by the same-origin policy, which dictates that certain types of data transfer must be restricted to only occur if the target resource's domain is identical to the page making the request. To bypass this limit, a new mechanism called JSONP was proposed in December 2005 by Bob Ippolito in his article "Remote JSON - JSONP" (http://bob.ippoli.to/archives/2005/12/05/remote-json-jsonp/).

JSONP (an abbreviation of "JSON with padding") works by creating a `script` element (either in HTML markup or inserted into the DOM via JavaScript), with a reference to a resource that returns JSON data that's wrapped by a function declared on the page performing the request, whose name is provided by the `script` element. Usually the name of the function is passed using a parameter named `callback`. For example, you might create the following `script` element:

```
<script src="http://www.example.com/data?callback=myFunction"></script>
```

In this case, `myFunction()` is a function defined in the page that performs the request that has to deal with the JSON returned. A server able to deal with such requests will usually respond as shown here:

```
myFunction({"name": "jQuery in Action"});
```

This causes the `myFunction()` function to be executed with the data returned by the server passed as an argument.

To learn more about JSONP, visit the website www.json-p.org.

"No examples to use `$.ajax()`?" we hear you say. Don't worry; the next chapter will be dedicated to creating an Ajax-powered project.

Sometimes it might be convenient if you could set default values for the options presented in table 10.2 for pages where you're planning to make a large number of requests. Let's discover how you can do that.

10.4.2 *Setting request defaults*

jQuery provides a way for you to define a default set of Ajax properties that will be used when you 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()`.

<div style="text-align:center">

Method syntax: $.ajaxSetup

</div>

`$.ajaxSetup(options)`

Establishes the passed set of option properties as the defaults for subsequent calls to `$.ajax()` or its derived methods like `$.get()` and `$.post()`, including those performed by third-party libraries or plugins.

Parameters

`options`	(Object) An object instance whose properties define the set of default Ajax options. These are the same properties described for the `$.ajax()` function in table 10.2. This function shouldn't be used to set callback handlers for success, error, and completion. (You'll see how to set these up using an alternative means in an upcoming section.)

Returns

`undefined`

At any point in script processing this function can be used to set up defaults to be used for all subsequent calls to `$.ajax()`. This method has to be used carefully because it'll also change the way plugins and other libraries you're using on your web pages perform Ajax calls using `$.ajax()` and similar methods.

> **NOTE** Defaults set with this function aren't applied to the `load()` method. Also, for utility functions such as `$.get()` and `$.post()`, the HTTP method can't be overridden by these defaults. For example, setting a default `type` of `"GET"` won't cause `$.post()` to use the GET HTTP method.

Let's say that you're setting up a page where, for the majority of Ajax requests (made with the utility functions rather than the `load()` method), you want to set up some defaults so that you don't need to specify them on every call. You can write the following as the first statement in the `script` element:

```
$.ajaxSetup({
  type: 'POST',
  timeout: 5000,
  dataType: 'html'
});
```

This would ensure that every subsequent Ajax call (except as noted previously) would use these defaults, unless explicitly overridden in the properties passed to the Ajax utility function being used. Specifically you're setting that all the requests will be POST

requests, that the maximum time after which the request has to time out is 5 seconds (5000 milliseconds), and that the response expected has to be interpreted as HTML.

Now, what about those *global events* we mentioned that were controlled by the `global` option?

10.4.3 Handling Ajax events

Throughout the execution of Ajax requests, jQuery triggers a series of custom events for which you can establish handlers in order to be informed of the progress of a request or to take action at various points along the way. jQuery classifies these events as local events and global events.

Local events are handled by the callback functions that you can directly specify using the `beforeSend`, `success`, `error`, and `complete` options of the `$.ajax()` function or indirectly by providing callbacks to the convenience methods (which, in turn, use the `$.ajax()` function to make the actual requests). You've been handling local events all along, without even knowing it, whenever you've registered a callback function to any jQuery Ajax function.

Global events are triggered for any Ajax request on the web page. You can establish event handlers for them via the `on()` method (just like any other event) used on `document` (attaching them on any other element won't work). The global events, many of which mirror local events, are `ajaxStart`, `ajaxSend`, `ajaxSuccess`, `ajax-Error`, `ajaxStop`, and `ajaxComplete`.

The attached handlers receive three parameters: the `jQuery.Event` instance, the `jqXHR` instance, and the object containing the options passed to `$.ajax()`. Exceptions to this parameter list are noted in table 10.3, which shows the jQuery Ajax events in the order in which they're delivered.

Table 10.3   jQuery Ajax event types

Event name	Type	Description
`ajaxStart`	Global	Triggered when an Ajax request is started, as long as no other requests are active. For concurrent requests, this event is triggered only for the first of the requests. Only the `jQuery.Event` instance is passed.
`beforeSend`	Local	Invoked prior to sending the request in order to allow modification of the XHR instance. You can cancel the request by returning `false`.
`ajaxSend`	Global	Triggered prior to sending the request in order to allow modification of the XHR instance.
`success`	Local	Invoked when a request returns a successful response.
`ajaxSuccess`	Global	Triggered when a request returns a successful response.
`error`	Local	Invoked when a request returns an error response.

Table 10.3   jQuery Ajax event types *(continued)*

Event name	Type	Description
`ajaxError`	Global	Triggered when a request returns an error response. An optional fourth parameter referencing the thrown error, if any, is passed.
`complete`	Local	Invoked when a request completes, regardless of its status. This callback is invoked even for synchronous requests.
`ajaxComplete`	Global	Triggered when a request completes, regardless of its status. This callback is invoked even for synchronous requests.
`ajaxStop`	Global	Triggered when an Ajax request completes and there are no other concurrent requests active. Only the `jQuery.Event` instance is passed.

To make sure things are clear, we want to stress that local events represent callbacks passed to `$.ajax()` (and its shortcuts), whereas global events are custom events that are triggered and can be handled by established handlers (to `document`), just like other event types.

In table 10.3 we reported that `ajaxStart` and `ajaxStop` receive a `jQuery.Event` instance as their only parameter. This parameter doesn't have a real use case, so it isn't reported in the official documentation. But we still wanted to report it for the sake of precision (and because we'll use it in the next demo). You can read more about this topic at https://github.com/jquery/api.jquery.com/issues/478.

In addition to using `on()` to establish event handlers, jQuery provides a handful of convenience functions to establish the handlers, as follows.

Method syntax: jQuery Ajax event establishers

`ajaxComplete(callback)`
`ajaxError(callback)`
`ajaxSend(callback)`
`ajaxStart(callback)`
`ajaxStop(callback)`
`ajaxSuccess(callback)`
Establishes the passed callback as an event handler for the jQuery Ajax event specified by the method name.

Parameters

`callback`   (Function) The function to be established as the Ajax event handler. The function context (`this`) is the DOM element upon which the handler is established. Parameters may be passed as outlined in table 10.3.

Returns

The jQuery collection.

Let's put together a simple example of how some of these methods can be used to easily track the progress of Ajax requests. The layout of our test page (it's too simple to be called a lab) is shown in figure 10.9 and is available at http://localhost[:8080]/chapter-10/ajax.events.html.

This page exhibits three controls: a count field, a Good request button, and a Bad request button. These buttons are instrumented to issue the number of requests specified by the count field. The Good request button will issue requests for a valid resource, whereas the Bad request button will issue the same number of requests for an invalid resource that will result in failures.

Figure 10.9 The initial display of the page we'll use to examine the jQuery Ajax events by firing multiple events and observing the handlers

In the code of this page, you define a number of event handlers as follows:

```
var $log = $('#log');
$(document).on(
   'ajaxStart ajaxStop ajaxSend ajaxSuccess ajaxError ajaxComplete',
   function(event) {
      $log.text($log.text() + event.type + '\n');
   }
);
```

This statement establishes a handler on the document object for each of the various jQuery Ajax event types. The handler writes a message showing the event type that was triggered into a textarea element having log as its ID.

Leaving the request count at 1, click the Good request button and observe the results. You'll see that each jQuery Ajax event type is triggered in the order described in table 10.3. But to understand the distinctive behavior of the ajaxStart and ajaxStop events, set the count control to 2 and click the Good request button. You'll see the display shown in figure 10.10.

Here you can see how, when multiple requests are active, the ajaxStart and ajaxStop events are triggered only once for the entire set of concurrent requests, whereas the other event types are triggered on a per-request basis.

Figure 10.10 When multiple requests are active, the **ajaxStart** and **ajaxStop** events are called around the set of requests rather than for each.

Now try clicking the `Bad request` button to generate an invalid request and observe the event behavior. You'll obtain the result shown in figure 10.11, which proves that this time the `ajaxError` event is fired.

As you've seen, `$.ajax()` gives you a lot of options to use, offering you great flexibility, but there may be times (not a lot, to be honest) when you want to do even more. For instance, you may want to handle requests based on some options or modifying existing ones before a request is made or to manage the transfer of the data of an Ajax call. A

Figure 10.11   The result of a bad request shows that the `ajaxError` event is called.

possible use case, as we'll discuss in the next section with an example, is to prevent an Ajax call to some domains you want to deny access to. Let's see what jQuery has to offer for such situations.

10.4.4 Advanced Ajax utility functions

In addition to all the methods and utility functions we've discussed so far, jQuery has other goodies to offer. Probably you won't have a lot of chances to see these two utility functions in action, but because they exist, we want to introduce you to them. Say "Hi!" to `$.ajaxPrefilter()` and `$.ajaxTransport()`.

`$.ajaxPrefilter()` can be used to prevent an Ajax request based on some custom options you set when you called `$.ajax()`. Its syntax is reported here.

Method syntax: $.ajaxPrefilter

`$.ajaxPrefilter([dataTypes,] callback)`
Handles custom Ajax options or modifies existing options before each request is sent and before they're processed by subsequent calls to `$.ajax()`.

Parameters

`dataTypes`	(String) An optional string containing one or more space-separated `dataTypes` as described for the `$.ajax()` function in table 10.2. If this parameter is passed, the handler is called only if the `dataTypes` of the request match.
`callback`	(Function) A function to set default values for future Ajax requests. This function receives three parameters: `options` containing the request options, `originalOptions` that stores the options provided to the `$.ajax()` call (without the defaults from `ajaxSettings`), and `jqXHR`, which is the `jqXHR` object of the request.

Returns

`undefined`

To see a concrete example of its use, imagine that you want to abort all the requests of type XML directed to a certain set of domains. You may want to do so because you know that they'll always fail.

To achieve this goal you can write code like the following, also available in the file http://localhost:8080/chapter-10/$.ajaxPrefilter.html and as a JS Bin (http://jsbin .com/bitiv/edit?js,console):

```
$.ajaxPrefilter('xml', function(options, originalOptions, jqXHR) {
   if ($.inArray(options.url, originalOptions.deniedDomains) !== -1) {
      console.log('Ajax request to ' + options.url + ' aborted');
      jqXHR.abort();
   } else {
      console.log('Ajax request performed');
   }
});
```

If the domain is not allowed, abort the request.

```
$.ajax(
   'http://www.google.com',
   {
      dataType: 'xml',
      deniedDomains: [
         'http://www.google.com',
         'http://www.manning.com'
      ]
   }
);
```

Performs an Ajax request

Prefilters requests based on the dataType specified and a set of denied domains

The aim of this function isn't limited to changing the behavior of the Ajax calls based on the options set. It can also be employed in cases where you want to redirect a request from the original `dataType` to another, which is achieved by returning the `dataType` you want.

The other less-known function we want to mention is `$.ajaxTransport()`. This is a low-level function that allows you to take control of how `$.ajax()` issues the transport of a request's data. Its syntax is shown here.

Function syntax: $.ajaxTransport

`$.ajaxTransport([dataType,] callback)`
Creates an object that handles the actual transmission of Ajax data.

Parameters

`dataType`	(String) An optional string containing the data type to use. If this parameter is passed, the handler is called only if the `dataType` of the request matches.
`callback`	(Function) A function to return the new transport object to use with the `dataType` provided. This function receives three parameters: `options` containing the request options, `originalOptions` that stores the options provided to the `$.ajax()` call (without the defaults from `ajaxSettings`), and `jqXHR`, which is the `jqXHR` object of the request.

Returns
`undefined`

The `callback` function of this method has to return a new transport object, which is a JavaScript object that provides two methods, `send()` and `abort()`, that are used internally by `$.ajax()`.

The `send()` function receives two parameters called `headers` and `completeCallback`. The former is an object of key-value pairs of request headers that the transport can transmit if it supports it, whereas the latter is the callback used to notify `$.ajax()` of the completion of the request.

With this last somewhat complicated utility function, we've completed our overview of the methods and functions jQuery provides to deal with Ajax. The examples shown so far are a good start to sink your teeth into jQuery's way of dealing with Ajax. Nonetheless you, our dear reader, deserve much more than that.

The aim of the next chapter is to show a real-world example that employs the power of Ajax to solve a common problem that you may face—and have probably already faced.

10.5   *Summary*

Ajax is a key part of modern applications, and jQuery is no slouch in providing a rich set of tools for you to work with.

For loading HTML content into DOM elements, the `load()` method provides an easy way to grab the content from the server and make it the content of any set of matched elements. Whether a GET or POST method is used is determined by the type of the `data` parameter provided.

When a GET is required, jQuery provides the `$.get()` and `$.getJSON()` utility functions. `$.getJSON()` is useful when JSON data is returned from the server. To force a POST, the `$.post()` utility function can be used.

When maximum flexibility is required, the `$.ajax()` utility function, with its ample assortment of options, lets you control the most minute 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 `$.ajaxSetup()` utility function that allows you to set default values for any frequently used options to the `$.ajax()` function (and for all of the other Ajax functions that use the services of `$.ajax()`).

To round out the Ajax toolset, jQuery also allows you to monitor the progress of Ajax requests by triggering Ajax events at the various stages, allowing you to establish handlers to listen for those events. You can bind the handlers using the `on()` method or use the convenience methods: `ajaxStart()`, `ajaxSend()`, `ajaxSuccess()`, `ajaxError()`, `ajaxComplete()`, and `ajaxStop()`.

Thanks to this impressive collection of Ajax tools under your belt, it's easy to enable rich functionality in your web applications. With this in mind, let's delve into our real-world demo.

Demo: an Ajax-powered
contact form

11

This chapter covers

- Effects with jQuery
- jQuery's utility functions
- Making Ajax requests
- Creating an accessible form

In the previous chapter we covered even more topics that belong to jQuery's core. In between the discussion of the methods, utility functions, and flags, we showed you a lot of snippets of code, demos, and lab pages. All these examples should have given you more confidence with the arguments treated.

In chapter 7, we developed a demo to show you the power of jQuery in a real-world example. It employed many of the methods and techniques you had learned up to that point in the book. Then we introduced you to more advanced topics like effects and animations, utility functions, and, even more important, Ajax. The latter is a key concept and also a great technique to adopt to build your web pages.

In this chapter, we'll tackle another real-world problem that many of you, sooner or later, will face: creating a contact form. Relying on the knowledge you've

acquired so far, you'll build not only a complete working contact form but also one that doesn't require a reload of the page to inform the user about the failure or success in sending the message. Just like the previous chapter, you'll use PHP as the language to develop the server-side part of the demo. If you don't know much about PHP, don't worry. The code will be so simple and well explained that you won't have a hard time understanding it.

With this in mind, and without any further ado, let's start.

11.1 The features of the project

Before delving into the development of your project, we'll discuss its requirements. The demo needs only two pages: one that contains the form (that we'll call `index.html`) and another for the backend business logic (that we'll name `contact.php`). You can play with this example by accessing the chapter-11 folder of this book's sources.

To keep things as simple as possible but still interesting, you'll create a form that contains four fields: full name, email, subject of the message, and the message itself. You want all of these fields to be mandatory. Finally, you want the email address to be well formatted and all the other fields to have at least four characters.

The form will be highly interactive and will check that the fields conform to the constraints established without the need to reload the page. To achieve this goal, you'll employ some of the techniques you learned in the previous chapters, and in particular the concepts discussed in the chapter dedicated to Ajax, to send requests to the server.

The tests to validate the input of the user will be performed for all the fields each time the user clicks the Submit button. In addition, you'll perform them on a single field every time that field loses the focus. If a value isn't valid, you'll warn the user with an informative message placed beneath the field.

Why server-side validation?

Some of you might wonder why we decided to employ server-side validation for the data instead of performing the task on the client using only JavaScript. The reason is that every validation you perform with JavaScript is unreliable and unsafe because a user could easily disable JavaScript. Thus, you'd end up allowing your users to send invalid or potentially dangerous data or duplicate the same validations on the server. To avoid these issues you'll employ server-side validation while keeping the page interactive through Ajax.

Figure 11.1 shows an example of the feature described.

Figure 11.1   An example of an invalid field and the error message shown
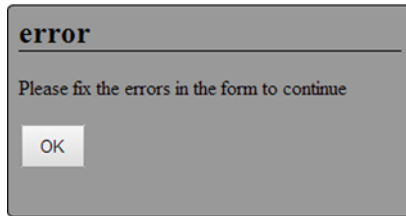
Figure 11.2a   The dialog box displayed in case the form contains one or more errors.

Figure 11.2b   The dialog box displayed in case all the form's fields are valid and the message is successfully sent. In the success message the name of the sender (in this case, Aurelio De Rosa) is included.

In addition to this feedback, when the user submits the form you'll show a dialog box with either an error message, shown in figure 11.2a, or a success message, shown in figure 11.2b.

The messages shown will be returned by the PHP page using a JSON object and injected into the dialog using jQuery. The structure of the JSON returned is shown in figure 11.3.

The structure of the JSON object is straightforward. It's made up of three properties: `status`, `message`, and `info`. `status` is a string used to specify whether the values written by the user are all correct. If so, the value will be `success`; otherwise, the value is `error`. The `message` property contains a string meant to be shown in the dialog box. In the case of the validation of a single field, if the value of the latter is valid, `message` will be an empty string. The `info` property provides an array containing objects related to the fields of the form that are invalid. Each of these objects exposes two properties: `field` and `message`. The aim of `field` is to specify the name of the field that contains the error. `message` has the same role described before, but this time it's used to provide the specific error of the invalid field.
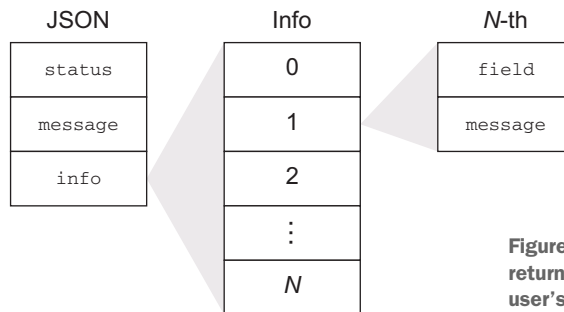
Figure 11.3   The structure of the JSON object returned by the PHP page that validates the user's inputs and sends the email

Let's move on to the next step: the markup of the form.

11.2   Creating the markup

In the previous section we discussed the constraints you want to apply to your form's fields. These constraints can be set using some of the new HTML5 attributes and types. For example, to have an email field that is well formatted and mandatory, you can have the following input in a form:

```
<input type="email" name="email" id="email" required />
```

In modern browsers, as soon as the user submits the form, this input will trigger the tests you want. Unfortunately, old browsers like Internet Explorer 9 and below don't recognize the email type and the required attribute. When something isn't recognized, the default behavior of the browsers is to ignore it. In this case it's like you defined type="text" and the required attribute wasn't specified. Therefore, you're on your own with these browsers.

If you want to use these new HTML5 attributes and types and then fall back on your own controls for older browsers, you can employ the technique you learned in section 4.1:

```
if (!('required' in document.createElement('input'))) {
    // Set our own controls
}
```

In this demo, to keep things as simple as possible, you'll pretend that you've forgotten HTML5. You'll always use your own controls and avoid using HTML5 attributes. With these considerations in mind, take a look at the code of your form, as shown in the following listing.

Listing 11.1   The markup of the contact form

```
<form id="contact-form" name="contact-form" class="box" method="post"
      action="contact.php">
  <div class="form-field">
    <label for="name">Full name:</label>
    <input name="name" id="name" />
    <span class="error"></span>
  </div>
  <div class="form-field">
    <label for="email">Email:</label>
    <input name="email" id="email" />
    <span class="error"></span>
  </div>
  <div class="form-field">
    <label for="subject">Subject:</label>
    <input name="subject" id="subject" />
    <span class="error"></span>
  </div>
  <div class="form-field">
    <label for="message">Message:</label>
    <textarea name="message" id="message"></textarea>
```

```
      <span class="error"></span>
   </div>
   <input type="submit" value="Submit"/>
   <input type="reset" value="Reset"/>
</form>
```

As you can see from the listing, each field is composed of three elements: a `label`, an `input` (textarea for the message), and a `span`. The `label` element is used to specify the name of the field and has the `for` attribute set to improve the accessibility of the form. The `<input>`s and the `<textarea>` allow users to write the information required. Finally, `span` is used to show the feedback illustrated in figure 11.1. In addition to these elements, you have the Submit and the Reset buttons at the bottom of the form.

The contact form isn't the only component of the page. You also need a dialog box to show the messages. The markup of the latter is pretty simple because it needs only a title, a paragraph, and a button to close the dialog. These three elements are wrapped into a container so you can treat them as a unique component.

The HTML code of the dialog is shown here:

```
<div class="dialog-box">
   <h2 class="title"></h2>
   <p class="message"></p>
   <button>OK</button>
</div>
```

With this snippet we've completed the overview of the HTML code of `index.html`. If you run the demo in its current state, it's neither interactive nor useful because it doesn't do anything at all.

Let's fix this by adding the backend code. Don't worry if you don't understand PHP; we'll highlight only the key points.

11.3 *Implementing the PHP backend*

The backend of your project has two main responsibilities: validating the user's input and sending the email. The former is the most interesting one for your purposes because, based on the specifications of your project, you have to deal with two different cases. The first case is a partial request resulting from the loss of the focus of a field. The second is a request containing the values of all the fields, resulting from the click of the Submit button.

To distinguish between these two cases, you'll add a custom parameter called `partial` to your partial requests (more on this when we discuss the JavaScript code). In such situations, the PHP page has to skip all the validations but the one related to the field that has lost the focus. Then it has to return a JSON object according to the result of the test. The result will be stored in a variable called `$result` (we have such an imagination!) and returned (using the `echo` language construct) using a PHP function called `json_encode()` as shown here:

```
echo json_encode($result);
```

To distinguish between the partial and the complete request, you'll employ a variable called $isPartial whose value is set as follows:

```
$isPartial = empty($_POST['partial']) ? false : true;
```

To help you understand the backend code, let's analyze the part of the code relative to the name of the user, shown in the next listing.

Listing 11.2   The code for the name field test

① The request isn't partial and the value is empty.

```
if (!$isPartial && $_POST['name'] === '') {          ←
        $result['info'][] = array(
            'field' => 'name',
            'message' => sprintf($messages['required'], 'Full name')
        );
} else if ((!$isPartial || isset($_POST['name']))
    && strlen($_POST['name']) <= 3
) {
        $result['info'][] = array(
            'field' => 'name',
            'message' => sprintf($messages['short'], 'Full name', 4)
        );
}
```

Sets the appropriate error data ②

③ If the request isn't partial or if a name has been entered but it's shorter than four characters

Sets the appropriate error data ④

The most interesting parts of this listing are the two if conditions. In the first one, you test if the current request isn't partial (negating the $isPartial variable) and the value is empty ($_POST['name'] === '') ①. If both evaluate to true, you set the appropriate error message specifying that the value is mandatory and thus must be filled ②.

The second if ③ is a bit trickier. You want to verify whether the value of the field is shorter than four characters (strlen($_POST['name']) <= 3) and if so, return an error. But it's when you want to perform the validation that things get interesting. You have to verify the length if the request isn't partial (!$isPartial), so all the fields must be validated, or if the request is partial and it concerns the name field ($isPartial && isset($_POST['name'])). Based on this discussion, you might think that the final condition to use is this:

```
!$isPartial || ($isPartial && isset($_POST['name']))
```

But the part on the right of the OR operator will only be evaluated if the part on the left is false—that is, when the request is partial. Relying on this information, you can shorten the condition, obtaining the following:

```
!$isPartial || isset($_POST['name'])
```

If the condition evaluates to true, you set the appropriate error message ④.

Understanding this condition may be hard at first, but reading the code carefully a couple of times should prove that we wrote the code properly. In case you're still unsure, you can take a look at the diagram in figure 11.4.

The validation of the other fields is similar to the one we just described, so we'll omit their discussion.

Now that we've covered the backend code, it's time to delve into the most interesting part of our project: the JavaScript code. In the next section you'll discover how you can employ jQuery to tie together the pieces you've built in order to bring your page to life.
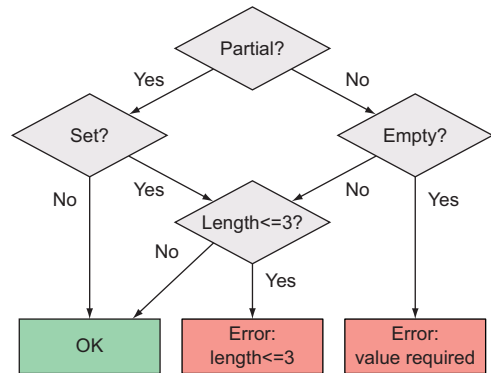
Figure 11.4   The process used to validate the user's input

11.4   *Field validation using Ajax*

Your form is in place and you have a PHP page able to process the incoming requests. In an old-fashioned, synchronous world this would be enough. The user fills the form and clicks the Submit button to send the data to the server; the latter uses a server-side language to process the data, validate them, and eventually produce an output page. Today, users expect websites to be highly interactive, and you can meet this expectation using Ajax.

The first feature you'll develop is the ability to give feedback to the user about the validity of a field, without the need to submit the form. This is a nice enhancement; it saves your users' bandwidth because there's no need to reload all the assets.

The idea is to offer a quick response to the user as soon as a given field loses the focus. To implement it, you have to perform an Ajax request to the server containing the value of the field that has just lost the focus. Then, if it isn't valid, you have to show the error returned by the server. In case an issue with the request occurs, you want to log it on the console so that you can eventually debug the project. The code implementing the described behavior is shown in the following listing.

Listing 11.3   Validating the data as the user fills the form

```
$('input, textarea', '#contact-form').blur(function() {
   var $this = $(this);

   $.ajax(
      'contact.php',
      {
         method: 'POST',
```

1 Attaches a handler for the blur event of the input and textarea elements

2 Performs an Ajax request to the server

3 The page to which to send the data

4 Specifies that the HTTP method to use is POST

Sets the type of data expected from the server ❺

Defines the handler to run after the result is returned successfully ❼

Serializes the field to send in the request's body with an additional parameter ❻

Sets the error message for the field if any ❽

Defines a callback in case of errors with the request ❾

```
dataType: 'json',
data: $this.serialize() + '&partial=true',
success: function(data) {
    if (data.status === 'error') {
        $this
            .next('.error')
            .text(data.info[0].message);
    }
},
error: function(data) {
    console.log(data);
}
    }
  );
});
```

The first thing you do in this snippet is to listen for the `blur` event triggered by either an `input` or a `textarea` element of your form ❶. Inside the handler for this event you send the request to the server employing the `$.ajax()` utility function because you want to have more control over the request ❷.

The first parameter you pass to this function is the page that will receive the data, which is `contact.php` ❸. In addition to the page, you pass an object as the second parameter to refine your request. Based on what you learned in the previous chapter, if you want to use the HTTP request methods correctly, you have to use a `POST` request. You'll set the `method` property to `POST` ❹. The second property you'll define is `dataType`. You know that your PHP page will return a JSON object; hence you assign the type of the data expected to the appropriate value (`json`) ❺. You also serialize the field and its value, adding your special parameter (`partial=true`) to mark the request as partial ❻. Once you've set all these properties, it's time to establish the callback for the two main statuses: `success` and `error`.

In the `success` callback ❼, you check the value of the `status` property of the object returned. If it's `error`, you set the next `span` element (described in section 11.2) to the value of the `message` property of the first item of the `info` property ❽. Inside the callback of the error status, which indicates an error in the request itself, you log on the console the message obtained ❾.

With this code you're able to inform your users about the validity of their input. Once an error has been shown beneath a field, the user may want to return to that field to correct the error. As the field regains the focus, you have to hide the error. By doing so, you prevent your users from feeling that even the new value they're writing is incorrect. Implementing this functionality is easy. All you have to do is attach a handler on the `focus` event for the same set of elements selected in previous code. Inside the handler, you hide the `span` element. Because you have to operate on same set of

elements, you can save some lines by chaining the code to implement this feature (in bold) with the one from listing 11.3. The relevant code for this change is as follows:

```
$('input, textarea', '#contact-form').blur(function() {
   // Code omitted here...
})
.focus(function() {
   $(this)
      .next('.error')
      .text('');
});
```

In this snippet and in listing 11.3, you use a few methods you learned throughout the book. You use the `blur()` shortcut method to attach a handler to the `blur` event. You employ the `$.ajax()` utility function to send an asynchronous request to the server. The `serialize()` method allows you to obtain the name and the value of a field as a string. The `next()` method lets you retrieve the `span` right after the field that lost the focus. Finally, using the `text()` method, you set the text of the `span` element. These are just examples of how using a few of jQuery's methods and utility functions enables you to create nice features for your web pages.

Regardless of the validity of the values inserted, the user may still click the Submit button. At the moment, this action triggers the classic synchronous request that you want to avoid. Let's see how you can change this default behavior.

11.5 *Even more fun with Ajax*

The ability to provide a quick feedback to your users is a nice feature. Nonetheless, you want to instruct your form to know what to do in case the user clicks the Submit button.

To build this feature, you need to attach a handler to the `submit` event triggered by the form. Your handler's aim is to perform an Ajax request to `contact.php`, sending all the values inserted. As soon as the server returns the result, you need to analyze it. If it contains errors, you need to show each error message in the respective field's `span`. Then you need to show the dialog box containing the general message of the request, which will indicate either success or failure. The code for this feature is shown in the listing that follows.

Listing 11.4   The handler to manage the submit request via an Ajax request

```
$('#contact-form').submit(function(event) {                 ◁── ❶ Attaches a handler to the
   event.preventDefault();                                         submit event of the form

   $.post(                                                   ◁──
      'contact.php',
      $(this).serialize(),                                        ❸ Sends an
      function (data) {                                              asynchronous POST
         if (data.status === 'error') {                             request to the server
            $.each(data.info, function(index, elem) {        ◁──
               $('#' + elem.field)                                ❹ Sets the error messages
                                                                     for each field
```

Prevents the default action ❷

```
                    .next('.error')
                    .text(elem.message);
            });
        }

        var $dialogBox = $('.dialog-box');

        $dialogBox
            .children('.title')
            .text(data.status);

        $dialogBox
            .children('.message')
            .text(data.message);

        $dialogBox
            .finish()
            .show();
    },
    'json'
    );
});
```

❺ Sets the title and the message of the dialog box

❻ Stops the currently running animation and shows the dialog box

This code isn't very different from that shown in the previous section. Here you establish a listener for the submit event of the form **❶**. In it you prevent the default action—that is, sending a synchronous POST request—using the preventDefault() method. You use it because you don't want the POST request to be sent anyway after executing your code **❷**.

The next operation you must perform is to send the request to the server. Once again, you need it to be a POST request. This time you don't need to specify a callback for the error status, so you can use the $.post() utility function because its arguments are enough for your needs **❸**. The first argument passed is the page that will receive the data, contact.php. Then you serialize the form so that the values contained in its fields will be sent in the body of the request. The third parameter is the success callback. Finally, specify the type of data expected (json).

Inside the success handler of $.post(), check the value of the status property of the object returned. If it's error, iterate over the info property to set the error message for each field containing errors **❹**.

Regardless of the validity of the fields, you have to show general feedback to your users using the dialog box you created. To do so, set the title and the message of the dialog box **❺** and then show it **❻**. Before showing the dialog box, also ensure that any previous animation is stopped by calling jQuery's finish() method.

With this code in place, when the user clicks the Submit button, the handler we just discussed is executed. Then the dialog box is shown, reporting the success or failure message. At this point the interaction will hang. Can you image why? Take a few seconds to think about it.

The reason is that you haven't instructed the button of the dialog to close (hide) the dialog once the user clicks it. Let's fix this issue.

11.5.1 *Hiding the dialog box*

Closing the dialog box shouldn't be hard at this point of your path. But before delving into the code, let's make a small optimization.

In the handler of the Ajax request shown in listing 11.4, you defined the variable `$dialogBox`:

```
var $dialogBox = $('.dialog-box');
```

Because you need to operate again on this element of the page, you can save some keystrokes and slightly optimize the performance of your code by moving that statement to the beginning of your code (at the top of the `script` element).

To hide the dialog box when the button inside it is clicked, you need to attach a listener for the `click` event of the button itself. This can be done with the following snippet:

```
$dialogBox.children('button').click(function() {
   $(this)
      .parent()
      .hide();
});
```

With this addition your demo does everything you planned. If you're excited about what you did so simply and you want to use this contact form in your next project, go ahead.

For those of you who want to go further, let's see how you can improve the project using some effects.

11.6 *Improving the user experience using effects*

Effects and animations are never an indispensable part of any application, in the sense that with or without them people should still be able to perform the task they want. Under some circumstances, though, they can be useful for improving the experience of your users. In this project you can allow error messages and the dialog box to appear and disappear gradually instead of being displayed and hidden in a snap.

The first effect you can add affects the handler of the button of the dialog box. You can update your code so that the dialog box will be hidden slowly. Instead of using `hide()`, you can employ jQuery's `slideUp()` method. You won't pass any parameter to it so that the effect will last 400 milliseconds (the default). The resulting code is shown here:

```
$dialogBox.children('button').click(function() {
   $(this)
      .parent()
      .slideUp();
});
```

If you want to take control of the duration, feel free to pass a parameter to the method.

In the same way that you added an effect when the dialog is hidden, you can modify the way it's shown. You can use `slideDown()` for this purpose, but for the sake of diversity, you'll pass an argument to `show()`. As you'll recall from section 8.2.1, you can pass either a number that specifies the number of milliseconds the effect will last or a string with a value of `slow`, `normal`, or `fast`. You don't want your users to wait too long before the dialog appears, so you'll pass the `fast` string and the effect will last 200 milliseconds.

Other effects can be added to the error messages, but we'll leave this to you as a simple exercise to complete.

The effects you introduced may please some users but annoy others. In an attempt to provide a great experience for as wide an audience as possible, you have to keep many different points of view in mind. Let's see what you can do for those who don't want animations to run.

11.6.1 Toggling the effects

In chapter 8 we introduced you to jQuery's flags. Among others, we discussed the `fx.off` flag that allows you to globally disable all the animations. To give the user this opportunity, you need to provide them an HTML element to use. In the demo you'll employ a selection box with two options, On and Off, but you're free to use any other HTML element that fits the purpose, like a check box or two radio buttons (one for each option).

The code of the `select` element is as follows:

```
<div class="animations-box">
   <label for="animations">Animations are:</label>
   <select id="animations">
      <option value="true" selected>On</option>
      <option value="false">Off</option>
   </select>
</div>
```

You'll place this markup just above your form. Once you've done this, you need to add the logic so that this `select` will actually do something. To achieve this goal you have to listen for changes (using the `change` event) of the selected `option` and update the `fx.off` flag accordingly. The following code serves the purpose:

```
$('#animations').change(function() {
   $.fx.off = $(this).val() === 'false';
})
.change();
```

As you may have noticed, you not only listened for the `change` event but also triggered it just after having set the handler. This ensures that the flag is set to whatever default value the `<select>` assumes. This trick comes in handy if you want to use the `Off` option as the default value without the need to update the JavaScript code (setting the `fx.off` flag to `true`).

Before concluding this project, there's one last point we want to discuss.

11.7 A note on accessibility

JavaScript is a powerful and ubiquitous language that allows you to perform an incredible number of tasks. As you've seen in this book, jQuery enables you to take your code to the next level, doing a lot with few lines of code. When developing for the web, however, you have to keep in mind that not everyone is able or even allowed to load and execute JavaScript. Some users may use a computer with JavaScript disabled. Or their server may fail to serve a JavaScript library, a module, or a file in general. For such occasions, as professionals of the web, you must have a plan B.

In this project you used JavaScript to enhance the experience of your users. If your JavaScript code fails to load for any reason, your contact form will still work. Your Submit button will be able to send the data of the form to the server. This is possible because you developed it by building on top of the native behavior of HTML. The only drawback is that the process will turn into a synchronous one and your users will see a new page (or the same page with different content) served by the server. But is this true? Will your users actually see a page?

You developed your `contact.php` page so that it always serves a JSON object. If the JavaScript code fails to load and the user submits the form, all they'll see is an unintelligible and unpleasant string representing the JSON object. What a shame! Is this really the best you can do?

As it turns out, you can slightly update your project to solve this issue. What you need to do is edit `contact.php` so that it can distinguish whether the request is an Ajax one or not. If it's an Ajax request, it can serve the JSON object; otherwise, it should use the data collected to fill a complete page and serve it to the user. Although it may seem like a simple change, it improves the accessibility of the demo and can save your users a lot of frustration. The presented approach, called *progressive enhancement*, has many advantages and is valid for more than just this project.

Planning for JavaScript failures isn't the only way you can improve the accessibility of your web pages. You can and should adopt WAI-ARIA (http://www.w3.org/TR/wai-aria/) as well. Explaining it in detail is outside the scope of this book, but to give you an

Progressive enhancement

Progressive enhancement is a methodology that emphasizes accessibility, semantic HTML markup, and external style sheet and scripting technologies. The expression was coined by Steven Champeon in a series of articles and presentations for Webmonkey and the SXSW Interactive conference in 2003.

This methodology evangelizes the creation of web pages so that everyone can access the basic content and functionality and then provide an enhanced version to those using a better technology (for example, a modern browser). It not only improves accessibility of web pages but can also improve their rank in the SERPs (Search Engine Result Pages), so you should adopt this methodology in all of your future projects.

idea, it provides an ontology of roles, states, and properties that define accessible user interface elements. These improve the accessibility and interoperability of web content and applications. One of the exposed roles, `dialog`, is a perfect fit for your dialog box. To employ it you have to add some attributes (highlighted in bold) to your markup as follows:

```
<div class="dialog-box" role="dialog" aria-labelledby="dialog-title"
    aria-describedby="dialog-desc">
  <h2 id="dialog-title" class="title"></h2>
  <p id="dialog-desc" class="message"></p>
  <button>OK</button>
</div>
```

Another improvement can be obtained using the HMTL5 `required` attribute we cited in the introduction of this chapter. Using it will enable User Agents that support HTML5 to provide the information that the field is mandatory to the users. Assistive Technologies (ATs) don't always proceed at the same speed as the usual suspects (Chrome, Firefox, and so on). To fill this gap, you can set the WAI-ARIA attribute `aria-required` to the mandatory elements as shown here:

```
<input name="name" id="name" required aria-required="true" />
```

Even if the UA supports HTML5, adding some WAI-ARIA attributes won't hurt.

The enhancements discussed are just a small set of what you can do to improve the accessibility of your contact form. But these changes should have opened your mind on this matter enough to spur you to consider accessibility in your next project.

11.8   Summary

In this chapter we put your knowledge into action by developing a simple but fully functional Ajax contact form. While developing the demo, we touched on a lot of topics covered in this book. We used the selectors, including the `context` parameter, which you learned in chapter 2. Methods like `parent()`, `next()`, and `find()`, introduced in chapter 3, were used to refine the selection of elements. The `text()` method, discussed in chapter 5, was employed to update the text of the `span` containing the errors. We employed some methods related to events, discussed in chapter 6, to listen for and trigger them (like `blur()`, `click()`, and `focus()`). We added some effects, covered in chapter 8, to let elements appear gradually. The `$.each()` utility function, introduced in chapter 9, helped you in iterating over the array of errors. Finally, we used the `$.ajax()` and the `$.post()` utility functions, discussed in chapter 10, to perform asynchronous requests to the server.

This wrap-up should have demonstrated to you that what we've covered so far isn't theoretical but has a lot of applications in the real world. In this demo we employed at least one concept from almost every chapter of this book.

We hope that throughout these pages you've come to understand how each piece of the jQuery library is important to achieve a certain goal (the contact form, in this

case) and how combining them gives you incredible power. We hope that you had fun developing this project and that you're more confident using the topics discussed.

With this example we've completed the second part of the book. Starting with the next chapter, we'll delve into more advanced concepts like creating plugins and unit-testing code.

Part 3

Advanced topics

In part 2 of this book we introduced you to an incredible number of jQuery selectors, methods, and utility functions. If you've mastered them, with a bit of time and a pinch of patience, you're now able to create every feature you want. In the last chapter, we proved to you that this is true and that with enough knowledge, the only limit is your imagination.

Although jQuery is powerful, it doesn't have a method or a function for everything your projects may require. To fill this gap jQuery has been built to be easily extensible, allowing web developers to include their functionalities as if they were part of the jQuery core. In the next chapters you'll learn how to create plugins for jQuery. Then we'll discuss the `Deferred` object and its methods. The `Deferred` object belongs to the jQuery core, but we've chosen to treat it separately because it's not easy to digest.

Unless you're developing a very small project—something that only you will use—you'll write code that needs to be refactored, updated, and changed in some way. For such situations you want to be sure that all the code that used to work before the changes won't break after the updates. A way to ensure this is to test your project. Because we're talking about testing, why not adopt the same solid unit-testing framework, called QUnit, that has been developed by the jQuery team and employed to test the jQuery library? It sounds pretty reasonable, doesn't it? This is what we'll do in chapter 14.

Finally, in the last chapter of this book we'll share with you some tools, techniques, tips, and tricks that are useful when dealing with large projects and show you how jQuery fits into them.

All the topics mentioned here make the difference between a midweight developer who knows how to use the library and a professional who's able to improve and optimize code while not forgetting to develop future-proof code (thanks to testing). We'll dissect these topics in the upcoming chapters.

Without wasting your precious time, let's dive into the advanced topics of this book.

*When jQuery
is not enough…
plugins to the rescue!*

This chapter covers
- Why extend jQuery with custom code
- Using third-party plugins
- Guidelines for effectively extending jQuery
- Writing custom utility functions
- Writing custom methods for jQuery objects

Over the course of this book, you've seen that jQuery gives you a large toolset of useful methods and utility functions, and you've also seen that you can easily tie these tools together to give your pages whatever behavior you choose. Sometimes that code follows common patterns you'll want to use again and again. When such patterns emerge, it makes sense to capture these repeated operations as reusable tools that you can add to your original toolset. In this chapter, we'll explore how to capture these reusable fragments of code as extensions to jQuery called *jQuery plugins*. A jQuery plugin comes in two forms: as a jQuery method for jQuery collections (like `find()` or `animate()`) or as a utility function (like `$.grep()` and `$.extend()`). In this chapter we'll cover both flavors.

When developing a project, it's unlikely that you have the time to build everything you need from scratch, especially if the code has been developed by someone else. Therefore, we'll also introduce some popular plugins you may want to look at.

But before any of that, let's discuss why you'd want to pattern your own code as extensions to jQuery in the first place.

12.1   Why extend jQuery?

If you've been paying attention while reading through this book, you undoubtedly have noted that adopting jQuery for use in your pages has a profound effect on how a script is written within a page.

jQuery promotes a certain style for a page's code: generally forming a jQuery collection and then applying a jQuery method, or chain of methods, to that collection. When writing your own code, you can write it however you 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 and one that we recommend. One good reason to pattern your code as jQuery extensions is to help maintain a consistent code style throughout the site. Another good reason is that by creating a reusable component, your future projects will benefit from it and other developers can also benefit if you publish the code on the web.

The final reason we'll consider (though it's possible others could list even more reasons) is that, by extending jQuery, you can use the existing code base that jQuery makes available to you. For example, by creating new jQuery methods, you automatically inherit the use of jQuery's powerful selector mechanism and the cross-compatibility fixes to browser issues the library provides. Why write everything from scratch when you can layer on such powerful tools?

Given these reasons, it's easy to see that writing your reusable components as jQuery extensions is a smart way of working. In the remainder of this chapter, we'll examine the guidelines and patterns that allow you to create jQuery plugins and you'll create a few of your own.

Before you start learning how to develop your own extensions, let's see how you can find, judge, and use other developers' plugins.

12.2   Where to find plugins

After several months' hard work, the jQuery team announced the release of the new (improved!) jQuery plugin registry with an official blog post (http://blog.jquery.com /2013/01/16/announcing-the-jquery-plugin-registry/) published on January 16, 2013. The new registry, accessible at http://plugins.jquery.com/, replaced the old one that was affected by a lot of problems. But circa two years later, the new registry was set in read-only mode, meaning that new plugin releases won't be processed. As a replacement for the new registry, the jQuery team recommends using npm (https:// www.npmjs.com/).

Accessing the URL, you'll be prompted with a clean interface that shows a search bar that you can use to find the plugin(s) you need. Once the results of a search are returned, you can click a name to find a lot of other information on the plugin, including the link to download it.

Although npm is the recommended channel where you can find the plugins, it's not the only one. An alternative with a nice UI but with a rather limited number of plugins is Unheap (http://www.unheap.com/).

If neither of them satisfies you or you can't find what you need, keep in mind that Google is your friend. But also remember to verify the source. After all, you're going to include one or more JavaScript files in your website! Not satisfied yet? Wait until the next few sections and we'll instruct you on how to create your own plugin.

Knowing where to look to find jQuery's extensions isn't enough. You don't want your well-crafted project to be filled with bad code or, even worse, to be slowed down by an incorrectly developed plugin. The next section gives you some tips on how to evaluate a plugin.

12.2.1 *How to use a (well-written) plugin*

When working on a project, relying on a third-party plugin is a smart way to save time because you don't have to build, test, and maintain it on your own. But is this always true? Based on our experience, the answer is no.

Adding a plugin to your project means adding a dependency to it. Choosing a plugin is an important decision because your whole project will be built on top of it. You should take some time to check several factors before committing to the use of a given plugin. Some of these factors are strictly related to the code whereas others are external. The combination of these clues gives you an idea of the stability and the quality of the component. Let's start analyzing some code-external factors.

CODE-EXTERNAL FACTORS

You rely on third-party components to free yourself from the burden of developing one or more features from scratch. But if you don't pay attention to the component you're using, you may find yourself losing more time than you saved in fixing existing bugs or understanding how to use it. Although we're talking about jQuery plugins, keep in mind that these points are also applicable to tools, libraries, and any third-party software.

> **NOTE**   For a more in-depth discussion on this and related topics we suggest you take a look at the chapter "Writing Maintainable, Future-Friendly Code" by Nicholas Zakas included in the fourth Smashing ebook titled *New Perspectives on Web Design* (2013, https://shop.smashingmagazine.com/smashing-book-4-new-perspectives-on-web-design.html).

The first thing to look at is the last time the plugin was updated. It gives you an idea of how much attention the author gives to this project. A plugin not updated for a while may indicate an abandoned plugin, something you want to avoid. Before using it take

the time to look at its changelog. The last update isn't always a good metric because a plugin built for a certain version of jQuery will still work in a new minor or patch release (more on these terms at http://semver.org/) because for the most part they're backward compatible. The case where an update breaks a plugin leads us to the second point.

The second factor to consider is the rate at which an author tackles issues. Software is never perfect and an issue can arise for many reasons. The speed at which the author fixes them is very important. If a plugin stops working due to an update of jQuery and you're using it, the lack of a rapid action forces you to either keep using the old version of jQuery or to fix the issue yourself, nullifying any advantage leading to the externalization of a feature.

Another important factor is the version of the plugin. Unless it's developed by a newbie, the version of the library has a precise meaning (described in the link provided previously). Therefore, don't ever use a 0.1.0 version of a plugin, unless you want to try it for fun. Often, software that hasn't reached version 1.0.0 is subject to a lot of changes that break backward compatibilities.

The fourth point is to check who the author is. Is it a company or a lone gunner? Does the company or the developer have a good reputation? Usually components developed by a company are well maintained because companies can invest money, whereas a single developer usually works on these projects in their spare time. But even if the author is a single developer, if they're an established authority, it can be worthwhile to use it.

Another factor is the documentation of the plugin. If it's poorly documented or lacks any documentation, it's better to continue your research. Such extensions will force you to invest a lot of time trying to understand how they work and how to use them.

In conclusion, keep in mind that not all jQuery plugins have the same quality and it's your responsibility to check them out to the best of your knowledge.

The factors analyzed in this section are important but represent only one side of the coin. To correctly evaluate a plugin, you have to have a good grasp of its code, too. The point here isn't to suggest that you go on the web and read the entire code of each plugin you want to use. This practice may take many hours of work. The idea is to have an overview of the quality of the source, parsing a sample, in the attempt to spot flags of bad code. In order to have the knowledge to evaluate the source's quality, you need to be instructed on the principles required to create a plugin. Therefore, we ask you to wait until the next few pages where we'll mentor you on the creation of a plugin through a step-by-step guide.

Now let's take a look at how you can use a third-party plugin.

USING A PLUGIN

Once you've found a plugin that fits your needs and you've checked that it deserves your attention, you can add it to your project. Using a well-written plugin is usually

easy. All you have to do is to store it in a folder accessible by the web server and add it after the jQuery library.

As the first example, we'll take a look at the jQuery Easing plugin (https://github.com/gdsmith/jquery.easing) that we introduced in section 8.3 when talking about easing functions. Once you've downloaded it, store it in a folder that your page can access. For example, you may store it in a folder called "javascript." Then you have to add it to your page using a `script` element, after the jQuery library. If you add the plugin before jQuery, you'll receive an error and all the JavaScript of your page will stop working. In your page you should have markup that resembles this:

```
<script src="javascript/jquery.1.11.3.min.js"></script>
<script src="javascript/jquery.easing.min.js"></script>
```

With the markup in place, what happens next depends on the plugin used. In this case, you don't have any new jQuery methods or utility functions to call. jQuery Easing only injects easing functions into the jQuery core, allowing you to use them as if they were native.

This plugin is a special case because many plugins require you to add some markup in your web page or to add a few classes to an element or even an ID. To see one of these extensions in action, you'll take a look at slick (https://github.com/kenwheeler/slick), a jQuery plugin used to create carousels. In the next example you'll write the code to create a carousel of images.

The first step required to use slick is to add its JavaScript file after the jQuery library. If you stored it in a folder called "javascript" at the same level of your HTML page, you'll have markup like the following:

```
<script src="javascript/jquery.1.11.3.min.js"></script>
<script src="javascript/slick.min.js"></script>
```

After adding the JavaScript file, you also have to add a CSS file included in slick. As you learned in chapter 1, the JavaScript files should always be located before the closing `</body>` tag, whereas the CSS files should be placed in the `<head>` of the page. If you've stored the CSS file in a folder called "css," you should have code like the following:

```
<head>
   <link rel="stylesheet" href="css/slick.css" />
```

Once you've finished doing this, you have to set up the markup of your page. Because you want a carousel of images, you have to wrap the images with a container element (in this case you'll use a `<div>`) as shown here:

```
<div class="carousel">
   <img src="images/image1.jpg" />
   <img src="images/image2.jpg" />
   <img src="images/image3.jpg" />
   <img src="images/image4.jpg" />
</div>
```

With this code in place, the only step left is to call the `slick()` method to run the magic:

```
<script>
    $('.carousel').slick();
</script>
```

This statement relies on the default configuration of the plugin, but you can change it to fit your need. If you want to deepen your knowledge about this plugin, you can take a look at the repository and its documentation.

These two examples should give you an idea of what you should expect when integrating a third-party plugin into your web pages. Now, before you start to create your own extensions, let's take a brief look at some popular and useful jQuery plugins.

12.2.2 *Great plugins for your projects*

This section gives you a short list of some of the most popular and most used jQuery plugins that you can use in your projects to perform common tasks. This isn't a comprehensive list by any means, but it's a good start.

The first plugin we suggest is typeahead.js (https://github.com/twitter/typeahead.js). It's a fast and full-featured autocomplete plugin developed by Twitter. This means that you can pass it a set of data and it will allow you to have an `<input>` that shows suggestions to the users while they're typing.

The second jQuery extension worth mentioning is isotope (https://github.com/metafizzy/isotope). It allows you to filter and sort UI elements using nice animations with different types of placement. Some examples are row, column, and the famous masonry.

Another very interesting plugin is pickadate.js (https://github.com/amsul/pickadate.js). It's a mobile-friendly, responsive, and lightweight jQuery date and time input picker. It adds a widget to an `<input>` so that once a user focuses on the element, a date or time picker is shown to facilitate the selection.

The fourth plugin in this list is Chosen (https://github.com/harvesthq/chosen). It's a library for making long, unwieldy select boxes more user-friendly and nicer looking.

Velocity (https://github.com/julianshapiro/velocity) is a jQuery plugin that reimplements jQuery's `animate()` method to improve its performance and include new features.

The last two plugins we suggest are jCarousel (https://github.com/jsor/jcarousel) and Magnific Popup (https://github.com/dimsemenov/Magnific-Popup). jCarousel is defined as a plugin to create a carousel that works with other objects in addition to images. Magnific Popup is a light and responsive light-box script with a focus on performance.

These plugins have become popular because they solve a real-world problem in a smart and efficient way or because they tackled the problem before any other plugin. Regardless of the reason, we're sure that you want your plugins to be as successful as

the ones presented here. To achieve this goal, you need to learn how to develop a good plugin. That's exactly the aim of the next section.

12.3 The jQuery plugin authoring guidelines

This section contains a set of guidelines to help you to name and structure a plugin. These guidelines ensure not only that your code plugs into the jQuery architecture properly, but also that it'll work and play well with other jQuery plugins and even other JavaScript libraries. Here we'll outline the basics and the best practices to follow when authoring a plugin.

Extending jQuery takes one of two forms:

- Methods to operate on a jQuery collection (what we've been calling jQuery **methods**)
- Utility functions defined directly on `$` (the alias for `jQuery`)

In the remainder of this section, we'll go over some guidelines that are common to both and then we'll dedicate other sections to each specific type.

To help you with the learning process, as you discover more rules to follow you'll also put them into action. The goal is to build Jqia Context Menu (Jqia being short for jQuery in Action), a jQuery plugin to show a custom context menu on one or more specified elements of a page. The context menu is the one that's shown on a PC screen when you click the right mouse button on a page or press the menu key on the page when it's focused.

For the context menu, the plugin will use an element of the page (typically a list), hidden by default, that has to be set up in the page. The element of the page acting as the menu will be retrieved through its ID. Your plugin will allow two actions, so it'll have two methods: one to initialize the plugin and one to destroy the effect. When initialized, the plugin will override the default behavior of the right click (which shows the usual context menu) to display the custom menu. When performing the destroy action, the plugin will clean up the resources and restore the default behavior.

Finally, to make things even more interesting, you'll enable developers who use your plugin to override the left click too. In this case, regardless of the mouse button clicked, the custom menu will be displayed. By default, this option will be disabled.

Now that we've explained our plan, roll up your sleeves because you have a lot of work to do.

12.3.1 File- and function-naming conventions

The first decision to make when developing a plugin is its name. When naming a plugin, you must avoid name collisions. It's important that the plugin you develop doesn't conflict with other files or plugins, which would lead to big headaches for web page authors.

The guideline recommended by the jQuery team is simple but effective, advocating the following format:

- Choose a name for the plugin that's short but also reasonably descriptive.
- Prefix the filename with *jquery*.
- Optionally add the name of the company or the suite.
- Follow that with the name of the plugin.
- Optionally include the version number of the plugin.
- Conclude with *.js*.

Taking our Jqia Context Menu plugin as an example, if you want to follow all these recommendations, you should name the file jquery.jqia.contextMenu-1.0.0.js.

The use of the jquery prefix eliminates (ideally) 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. In our example, the name of the plugin consists of more than a word (at this point a lot of names are already taken). We wrote the words using camel-case syntax, but you can use lowercase for all the words or even separate them by dots or dashes. Which one you use is a matter of personal taste, and our suggestion is to pick a convention and stick with it.

One way to ensure that your plugin filenames are *unlikely* (you can never be 100% sure) to conflict with others is to subprefix them with a name that's unique to you or your organization or a suite. For example, if we wanted to subprefix plugins that belong to this book, we could use the filename prefix jquery.jqia, as you did in the previous example.

The third point of the list is optional for a good reason. Let's say that some developers are using a plugin we've published. Things are going nice and our plugin is successful. We want to ship a new version having new features, and here comes the problem. Our file is called something like jquery.jqia.contextMenu-1.1.0.js. Thus, the developers using our plugin not only have to update the JavaScript file but also have to change the markup to update the version suffix. It would have been much simpler if our plugin file was called jquery.jqia.contextMenu.js, specifying the version with a comment inside the file. Doing so, the developers could have replaced only the JavaScript file without updating the markup.

These explanations should clarify why you need such rules. You can put them into action with your plugin project. You'll ignore the optional rule about the version, so go on and create a new file, naming it jquery.jqia.contextMenu.js.

In this section we stressed the importance of naming files and how you can't make any assumption about what's been used in other developers' websites. The same concern applies to our lovely $ shortcut. Let's dig in.

12.3.2 *Beware the $*

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 web developer intends to use the `$.noConflict()` function (discussed in section 9.2) to allow the `$` alias to be used by another library (most notably Prototype). We could employ the `jQuery` name in place of the `$` alias, but dang it, we *like* using `$`.

In section 9.2 we introduced a design pattern called IIFE (Immediately-Invoked Function Expression), covered in more details in the appendix, which is often used to make sure that the `$` alias refers to the `jQuery` name in a localized manner, without affecting the remainder of the page. This pattern can and should also be employed when defining jQuery plugins, as follows:

```
(function($){
//
// Plugin definition goes here
//
})(jQuery);
```

By passing `jQuery` to a function that defines the parameter as `$`, the latter is guaranteed to reference `jQuery` within the body of the function. You can now happily use `$` to your heart's content in the definition of the plugin.

With this new wisdom in mind, open the jquery.jqia.contextMenu.js file and put the previous snippet inside it (you can omit the comments if you want).

Now take a look at another guideline for authoring plugins that deal with parameters.

12.3.3 *Taming complex parameter lists*

Most plugins tend to be simple affairs that require few, if any, parameters. 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.

jQuery's `on()` method is a good example of such behavior; if the optional `data` parameter is omitted, the listener function, which is normally specified as the fourth parameter, can be supplied as the third. If the `selector` parameter is missing, too, you can even pass the handler as the second argument. The dynamic nature of JavaScript allows you to write such flexible code, but this sort of thing can start to break down and get complex (for both web developers and plugin authors) when the number of parameters grows larger. The possibility of a breakdown increases when many of the parameters are optional.

Consider a function whose signature is as follows:

```
function complex(p1, p2, p3, p4, p5, p6, p7) {
    // Code here...
}
```

This function defines seven parameters. Now let's say that all but the first are optional. There are too many optional parameters to make any intelligent guess about the intention of the caller when optional parameters are omitted. If a caller of this function is omitting only trailing parameters, this isn't much of a problem, because the optional trailing arguments can be detected as `undefined`. But what if the caller wants to specify `p7` but allow `p2` through `p6` by default? And what if some of the omitted parameters

accept the same data type (nullifying any chance to resort to the data type of the values passed)? 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);
```

Web developers using this function are forced to carefully keep track of counting `null`s and the order of the parameters; plus, the code is difficult to read and understand. But short of not allowing the caller so many options, what can you do?

Again, the flexible nature of JavaScript comes to the rescue. A pattern that allows you 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});
```

The second would be as follows:

```
complex(valueA, {
   p3: valueC,
   p4: valueD,
   p7: valueB
});
```

Much, much better!

You don't have to account for omitted parameters with placeholder `null`s, and you also don't need to count parameters. Each optional parameter is conveniently labeled so that it's clear exactly what it represents (when you use better parameter names than `p1` through `p7`, that is).

> **NOTE**   Some APIs follow this convention of bundling optional parameters into a single `options` parameter (leaving required parameters as standalone parameters). Others bundle the complete set of parameters, required and optional alike, into a single object. We prefer to use the second approach because the amount of required parameters may increase over the time, so this solution is more future-proof.

Although this is obviously a great advantage to the caller of your complex functions, what about the ramifications for you as the plugin author? As it turns out, you've already seen a jQuery-supplied mechanism that makes it easy for you to gather these optional parameters together and merge them with default values. Let's reconsider our example function with a required parameter and six optional parameters. The new, simplified signature is

```
complex(p1, options)
```

Within this function, you can merge those options with default values with the handy
`$.extend()` utility function. Consider the following:

```
function complex(p1, options) {
   var settings = $.extend({
         p2: defaultValue1,
         p3: defaultValue2,
         p4: defaultValue3,
         p5: defaultValue4,
         p6: defaultValue5,
         p7: defaultValue6
      },
      options || {}
   );

   // Remainder of the function...
}
```

By merging the values passed by the web developer in the `options` parameter with an
object containing all the available options with their default values, the `settings` vari-
able ends up with the default values superseded by any explicit values specified by the
web developer.

> **TIP** Rather than creating a new `settings` variable, you could use the
> `options` reference itself to accumulate the values. That would cut down on
> one reference on the stack, but let's stay on the side of clarity for the moment.

In the previous code you guard against an `options` object that's `null` or `undefined`
with `|| {}`, which supplies an empty object if `options` evaluates to `false` (as you know
`null` and `undefined` do). Easy, versatile, and caller-friendly!

Now that you've moved another step forward in the process of learning how to cre-
ate beautiful and well-written plugins, let's apply it to your project. Recalling the
description of Jqia Context Menu, you'll remember that you need an optional param-
eter that specifies whether the custom menu must also be displayed when the left
mouse button is clicked. In addition to this option, you need to specify the ID of the
element that will act as the menu. Although you need only two parameters, one man-
datory and one optional, you'll use the approach of passing a single object to your
plugin. The reason, as we mentioned before, is that this approach is more future-
proof.

Turning this description into code, which must replace the contents of the
JavaScript file you're working on, results in the following:

```
(function($) {
   var defaults = {
      idMenu: null,
      bindLeftClick: false
   };
})(jQuery);
```

In this code you define an object, called `defaults`, containing a property to specify the ID of the menu (`idMenu`) and another one to know if the click with the left button should be overwritten (`bindLeftClick`).

At the moment you've only defined an object with two properties, so there's nothing special here. Let's move on to the guidelines regarding how you should develop the methods of your plugin.

12.3.4 *Keep one namespace*

Similarly to what you've seen in regard to naming files, you should ensure the names you give to your functions, whether they're new utility functions or methods for the jQuery collections, don't collide with methods of other extensions you might be using.

When creating plugins for your own use, you're usually aware of what other components you'll use; unless your project is huge, it's an easy matter to avoid any naming collisions. What if you're creating your plugins for public consumption? What if your plugins, which you initially intended to use privately, turn out to be so useful that you want to share them with the rest of the community?

To better understand this concept, let's see a concrete example. As we said, Jqia Context Menu needs two methods: `init()` and `destroy()`. You might be tempted to add the following code in your JavaScript file:

```
var init = function(options) {
   // Code here...
};
var destroy = function() {
   // Code here...
};
```

Unfortunately this isn't what you really need.

One of the main points of using an IIFE is to create an environment where variables and functions declared inside the IIFE can't be accessed from outside. This behavior is indeed useful for your `defaults` variable that you don't want to be visible from outside your plugin but not for your methods. Inside your IIFE you need a way to expose your methods to the outside world.

The plugin you're creating operates on jQuery collections, and to add a method for jQuery collections you must assign them to a property named `$.fn`. Updating your JavaScript file to comply with these considerations results in the code shown in the following listing.

> **Listing 12.1   A first version of the Jqia Context Menu plugin**

```
(function($) {
   var defaults = {
      idMenu: null,
      bindLeftClick: false
   };
```

```
    $.fn.init = function() {
        // Code here...
    };
    $.fn.destroy = function() {
        // Code here...
    };
})(jQuery);
```

The code shown in this listing can't do anything because you haven't defined the body of `init()` and `destroy()`. Nonetheless, you can already call them as if they were native jQuery methods.

Playing with new toys is always exciting, so you might be tempted to add a `console.log()` statement inside each of the two methods defined (just to prove they're executed), add the jQuery library and the jquery.jqia.contextMenu.js file to a web page, and then write a statement like the following to see your project in action:

```
$('p').init();
```

Unfortunately, if you run the page, right after the output of the `console.log()` statement you'll get a scary error message. The reason is that you've chosen a common name for your method, so common that jQuery has one of its own. Your method is in conflict with the previously defined jQuery `init()` method, and this issue raises an important point: namespacing methods.

Properly namespacing your plugin is an important part of your development. It assures that your extension will have a low chance of being in conflict with other plugins or even jQuery's core methods.

Applying this guideline, you could rename your methods as `jqiaCustomMenu-Init()` and `jqiaCustomMenuDestroy()`. With this change you can safely call both because they won't conflict with any jQuery native method. Although the change works, the jQuery guidelines discourage claiming more than one namespace to avoid cluttering `$.fn`. The suggested solution, employed by a lot of top plugins, is to collect all the plugin's methods in an object literal (usually called `methods`) and call them using a single method that accepts a string containing the method's name to execute.

To give you an idea, let's say that your call-all-the-methods method is named `jqia-ContextMenu`. Using it, you can execute the `destroy()` method as follows (let's forget about parameters for now):

```
$('#element').jqiaContextMenu('destroy');
```

Following this rule, you can turn the code in listing 12.1 into that shown in the next listing.

Listing 12.2   Revisited version of Jqia Context Menu plugin

```
(function($) {
    var defaults = {                          ⟵————————— ❶ Defines the default options
        idMenu: null,
        bindLeftClick: false
    };
```

```
var methods = {
    init: function(options) {
        // Code here...
    },
    destroy: function() {
        // Code here...
    }
};

$.fn.jqiaContextMenu = function(method) {
    if (methods[method]) {
        return methods[method].apply(
            this,
            Array.prototype.slice.call(arguments, 1)
        );
    } else if ($.type(method) === 'object') {
        return methods.init.apply(this, arguments);
    } else {
        $.error('Method ' + method +
         ' does not exist on jQuery.jqiaContextMenu');
    }
};
})(jQuery);
```

2 Declares the object literal containing the methods

3 Claims the jqiaContextMenu namespace and assigns it an anonymous function

4 If the required method exists, calls the method passing the other parameters as its arguments

5 If the argument is an object, calls the init() method

6 If none of the above, raises an exception

Inside the outermost anonymous function, you set up the default options for the plugin parameters **1**. Then you declare an object literal containing the methods you need (with an empty body at the moment) **2**.

The second part of the code is the most interesting and it's really clever. First, you assign an anonymous function to a new property of `$.fn`, called `jqiaContextMenu` **3**. Doing so, you're claiming only one name for all your methods instead of one for each method, as the guidelines suggest. Inside this function, you test if the first argument passed, `method`, has a corresponding property inside the `methods` variable **4**. If so, you use an advanced JavaScript technique that uses JavaScript's `apply()` and `call()` functions to execute the required method. `apply()` is used to set the context of the function (`this`) to the set of elements in the jQuery collection and to forward to the invoked method all the parameters passed but the first one to your plugin (because the first is the name of the method to invoke). Since `arguments` isn't a real array (it's called an array-like object), you use array's `slice()` method and `call()` to remove the first argument from `arguments`.

NOTE   If you need to learn or refresh your knowledge of `apply()` and `call()`, please refer to the appendix.

If the first test fails, you check if the first parameter is an object **5**. In such cases you invoke your `init()` method, forwarding to it all the parameters passed to `jqia-ContextMenu()`. The reason is that if the user calls `jqiaContextMenu()` passing the object containing the options, you assume that the user wanted to initialize the plugin and invoke its default action. In this case, too, you use `apply()` to set the context of the function to the set of elements in the jQuery collection and to forward the

arguments to the invoked method. Finally, if both tests fail, you raise an exception using the `$.error()` utility function ❻.

As you can see, the update you've made to your code is effective and enables you to use only one namespace (`jqiaContextMenu`) for all the methods. The same rule we discussed in this section is valid for events bound and data stored by your plugin. Let's discover more.

12.3.5 *Namespacing events and data*

In chapter 6 you learned about the possibility of namespacing events. This feature is particularly useful when authoring plugins. It's a best practice that plugins that attach handlers to events namespace them. Following this principle, if you later need to unbind them, you can perform the action without fear of interfering with other plugins that may be listening for the same events.

In addition to listening for events, some plugins need to store data on one or more elements of a web page. They can be useful to keep track of the state of an element or to check if the plugin has already been called on that element. This can be done using jQuery's `data()` method we introduced in chapter 4. Easy to retrieve, easy to delete.

By following all the guidelines described so far, you'll end up with a great plugin structure. But you're still missing the most important pieces: the methods body. Let's start with `init()`.

THE INIT() METHOD OF JQIA CONTEXT MENU

The `init()` method has the following responsibilities:

1. Check that the options are passed to the plugin, especially that mandatory ones are provided.
2. Merge the passed options with the default values.
3. Test if the plugin has already been initialized on the selected element(s).
4. Store the options on the element(s) in the jQuery collection.
5. Listen for the mouse right button's click event, named `contextmenu`, on the element(s) in the jQuery collection to show the custom menu. Optionally, listen for the mouse left button's click event (`click`).
6. Hide the custom menu when a `click` event is fired *outside* the element(s) in the jQuery collection.

To perform the first step, you must verify that `idMenu`, the property containing the ID of the element that acts as the custom menu, is set and the element exists on the page. This is accomplished with the following code:

```
if (!options.idMenu) {
   $.error('No menu specified');
} else if ($('#' + options.idMenu).length === 0) {
   $.error('The menu specified does not exist');
}
```

In this code, you use the `length` property to test if the element exists on the page.

The second step is also easy to achieve. You need to call jQuery's `extend()` utility function to merge the values:

```
options = $.extend(true, {}, defaults, options);
```

As you can see in this statement, you're reusing `options` to avoid adding an extra (unneeded) variable.

Steps three and four are closely related to each other. Once the plugin has been initialized on the selected elements, you use jQuery's `data()` method to store the options under the same name. In this case you'll also use them to verify whether the element has already been initialized by your plugin. You can use the stored information with other functionalities you may want to add, like changing the configuration for a given element later on.

To store the data you can write the following statement:

```
this.data('jqiaContextMenu', options);
```

When the plugin is executed for the first time on the page, you're sure that no elements have already been initialized. What if you run Jqia Context Menu on the same set of elements? The double initialization on an element is something you want to avoid because, for example, it'll add the same event handler twice. You need to check that each element in the set of matched elements doesn't have any data stored using your plugin's namespace (`jqiaContextMenu`). This task is performed with the following code:

```
if (
   this.filter(function() {
      return $(this).data('jqiaContextMenu');
   }).length !== 0
) {
   $.error('The plugin has already been initialized');
}
```

Although short, this snippet gives us the opportunity to reinforce an important point: the meaning of `this` in a plugin. Within the function attached to `$.fn`, the `this` keyword refers to the jQuery instance (the jQuery collection on which the plugin is called). You can use every jQuery method directly without the need to wrap it using the `$()` method (for example, `$(this)`). The same is true for the functions defined in the `methods` object because you've changed their context using `apply()`. If you didn't use `apply()`, inside `init()` the `this` keyword would refer to the `methods` object.

Due to how you've structured the plugin, inside a callback executed within the plugin, the `this` keyword refers to a specific DOM element. In your code you use jQuery's `filter()` method, which iterates over the elements in the set. At the first iteration, the `this` of the callback will refer to the first element in the set of matched elements, at the second iteration `this` will refer to the second element in the set, and so on. That's why inside the anonymous function passed to `filter()` you passed `this` as the argument of `$()`: to use jQuery's `data()` method.

Now that you have a better understanding of the meaning of `this` inside a jQuery plugin, let's continue our discussion of the `init()` method.

Step five is the core of your project. To accomplish it you need to add a callback to the `contextmenu` event, which is typically fired when the user on a PC clicks the right mouse button. As you'll recall, you're also providing the opportunity to listen for a click performed using the left mouse button. Based on the options passed by the developer, you may need to listen for both `contextmenu` and `click`.

Inside the callback you need to prevent the default behavior; otherwise, the native context menu will be displayed as well. Once that's done, you have to set the position of the custom menu according to the position of the mouse at the time the click was performed (this information is stored in the `Event` object passed to your callback). Finally, you have to show the menu.

The code that performs these actions is shown here:

```
this.on(
    'contextmenu.jqiaContextMenu' +
       (options.bindLeftClick ? ' click.jqiaContextMenu' : ''),
    function(event) {
        event.preventDefault();

        $('#' + options.idMenu)
            .css({
                top: event.pageY,
                left: event.pageX
            })
            .show();
    }
);
```

In this code you're using the ternary operator to establish if you need to listen for the `click` event, too. You're passing an object to jQuery's `css()` method to set the position of the menu (you also need to set `position: absolute` on the menu, but this declaration is set in a CSS file). You aren't setting the unit (pixels, in this case), because when not specified, jQuery assumes the value is in pixels.

The last step consists of hiding the custom menu when a click is performed, regardless of the mouse's button, outside the elements initialized by Jqia Context Menu. This means that you should attach a handler that hides the custom menu to all the elements of the page except the ones initialized by your plugin. Attaching a listener to every element on the page has serious drawbacks in terms of performance, so you'll take advantage of event delegation. You'll attach only one listener to the root of the document, the `html` element, as follows:

```
$('html').on(
    'contextmenu.jqiaContextMenu click.jqiaContextMenu',
    function() {
        $('#' + options.idMenu).hide();
    }
);
```

With this snippet in place, it may seem that you've completed your `init()` method, but this isn't true.

In its current state, your project has a serious bug. Once a click is performed on an initialized element, a custom menu is shown. Then, due to event bubbling, the event is propagated toward the root of the DOM tree. Once it reaches the `html` element, the callback you attached is executed, hiding the custom menu. The result is that the custom menu is shown for a few milliseconds (so fast you can't even see it). To fix this issue, you have to call `event.stopPropagation()` inside the callback of the initialized elements.

With this last consideration, you've completed the `init()` method. Let's now explore how to develop the `destroy()` method. (If you're curious about how the complete plugin will look, you can jump to listing 12.3.)

THE DESTROY() METHOD OF JQIA CONTEXT MENU

The `destroy()` method is responsible for cleaning up the resources used by your plugin, which consist of the data stored on the initialized elements and the listener attached, including those attached to the `html` element. Besides, you want to ensure the custom menu is hidden before `destroy()` is completed; otherwise it'll be displayed until the page is reloaded.

One of the possible versions of code that implements these needs is shown here:

```
this
    .each(function() {
        var options = $(this).data('jqiaContextMenu');
        if (options !== undefined) {
            $('#' + options.idMenu).hide();
        }
    })
    .removeData('jqiaContextMenu')
    .add('html')
    .off('.jqiaContextMenu');
```

As the first thing, you loop over each element in the set of matched elements to retrieve the custom menu attached and hide it (if this element was initialized by your plugin). Then you remove the data stored on each element, but this time you don't need to iterate over them to perform special checks, so you can let jQuery's `remove-Data()` method do it for you.

The last action to perform is to unbind all the handlers attached to events namespaced with `jqiaContextMenu`. To do that, you add the `html` element to the current jQuery set and call the `off()` method, passing the string `".jqiaContextMenu"` to it.

The most observant of you may have noted that you used the string `"jqiaContext-Menu"` a lot of times in the snippets shown previously, although for different purposes. To avoid these repetitions you can store it in a private variable (accessible only inside your plugin) and then change your code accordingly.

Assuming the following statement is added beneath the `defaults` variable,

```
var namespace = 'jqiaContextMenu';
```

the body of `destroy()` can be rewritten as follows:

```
this
   .each(function() {
      var options = $(this).data(namespace);
      if (options !== undefined) {
         $('#' + options.idMenu).hide();
      }
   })
   .removeData(namespace)
   .add('html')
   .off('.' + namespace);
```

At this point your plugin is working, but there are two additional gems of wisdom to discover.

12.3.6 *Maintaining chainability*

Throughout this book you've made a massive use of jQuery chaining to perform several operations in one statement. Your methods don't return a value, so `undefined` (which is the default) will be returned. Because of this, a developer using your extension can't call any other jQuery method after calling `jqiaContextMenu()`.

The change required to maintain chainability in a plugin is simple yet invaluable. What you need to do is to ensure that your methods always return the `this` keyword. Applying this change to the `destroy()` method, after the line

```
.off('. ' + namespace);
```

you'll have

```
return this;
```

You should make the same change for the `init()` method of the plugin as well. Thanks to this change you maintain chainability, allowing anyone using Jqia Context Menu to continue to operate on the same set in a single statement.

12.3.7 *Provide public access to default settings*

Your extension isn't very customizable, but as things become more complex, you may find yourself passing the same large subset of options over and over again to different elements.

An improvement you can make is to expose the default settings so that a developer using your plugin can override them. With this change, the developer only needs to pass an object with the different options at each call of the plugin.

To apply this improvement to your project you need to make two changes. The first change is made on the `defaults` variable. In order to expose it to the external world, you need to assign it to the `$.fn` property. To avoid going against the rule of

not claiming more than one namespace, you'll set the object containing the default values as a property of `jqiaContextMenu`. Therefore, you'll turn

```
var defaults = {
   // Options here...
};
```

into

```
$.fn.jqiaContextMenu.defaults = {
   // Options here...
};
```

You also need to move the default configuration after the statement where you claim the namespace (`$.fn.jqiaContextMenu = function(method) {`).

With this update, the default values can't be referenced anymore using the `defaults` variable. You have to replace each occurrence of `defaults` in your code with `$.fn.jqiaContextMenu.defaults`. In your project there's only one occurrence, which resides inside the `init()` method, when you merge the options with the default ones. Therefore you have to change that statement as shown here:

```
options = $.extend(true, {}, $.fn.jqiaContextMenu.defaults, options);
```

With these changes in place, let's look at how you can use the exposed default values.

Let's say that you want to always bind the left click when calling Jqia Context Menu. You can write

```
$.fn.jqiaContextMenu.defaults.bindLeftClick = true;
```

Then you can call the plugin by passing only the `idMenu` property inside the object.

With this last change you've completed your project. The final code is shown in the following listing.

Listing 12.3   The final version of Jqia Context Menu

```
(function($) {
   var namespace = 'jqiaContextMenu';

   var methods = {
      init: function(options) {
         if (!options.idMenu) {
            $.error('No menu specified');
         } else if ($('#' + options.idMenu).length === 0) {
            $.error('The menu specified does not exist');
         }

         options = $.extend(
            true,
            {},
            $.fn.jqiaContextMenu.defaults,
            options
         );

         if (
            this.filter(function() {
               return $(this).data(namespace);
```

```
        })).length !== 0
        ) {
        $.error('The plugin has already been initialized');
      }

      this.data(namespace, options);

      $('html').on(
        'contextmenu.' + namespace + ' click.' + namespace,
        function() {
          $('#' + options.idMenu).hide();
        }
      );

      this.on(
        'contextmenu.' + namespace +
          (options.bindLeftClick ? ' click.' + namespace : ''),
        function(event) {
          event.preventDefault();
          event.stopPropagation();

          $('#' + options.idMenu)
            .css({
              top: event.pageY,
              left: event.pageX
            })
            .show();
        }
      );

      return this;
    },
    destroy: function() {
      this
        .each(function() {
          var options = $(this).data(namespace);
          if (options !== undefined) {
            $('#' + options.idMenu).hide();
          }
        })
        .removeData(namespace)
        .add('html')
        .off('.' + namespace);

      return this;
    }
};

$.fn.jqiaContextMenu = function(method) {
    if (methods[method]) {
      return methods[method].apply(
        this,
        Array.prototype.slice.call(arguments, 1)
      );
    } else if ($.type(method) === 'object') {
      return methods.init.apply(this, arguments);
    } else {
      $.error('Method ' + method +
        ' does not exist on jQuery.jqiaContextMenu'
```

```
            );
        }
    };

    $.fn.jqiaContextMenu.defaults = {
        idMenu: null,
        bindLeftClick: false
    };
})(jQuery);
```

This code is available in the file js/jquery.jqia.contextMenu.js in the source code of this book. To have this plugin completely working, you also need to set up an accompanying style sheet that you can find in css/jquery.jqia.contextMenu.css. We also provided a demo that you can find in the file chapter-12/jqia.contextMenu.html so that you can play with your plugin. Figure 12.1 shows the result of clicking the right mouse button on an element of the demo page that has been initialized by the Jqia Context Menu plugin.

Figure 12.1   The effect of clicking the right mouse button on an element that has been initialized by the Jqia Context Menu plugin

What do you think of the result? Are you happy? We hope so and that you had fun while developing this small project. In the next section we'll develop a more complex jQuery plugin built according to the guidelines you just learned.

12.4   *Demo: creating a slideshow as a jQuery plugin*

For our more complex plugin example, you're going to develop a jQuery method that allows a web developer to whip up a quick slideshow page. You'll create a jQuery plugin, which you'll name Jqia Photomatic, and then you'll whip up a test page to put it through its paces. When complete, this test page will appear as shown in figure 12.2. This page sports the following components:

- A set of thumbnail images
- A full-size photo of one of the images available in the thumbnail list
- A set of buttons for moving through the slideshow manually and for starting and stopping the automatic slideshow

The behaviors of the page are as follows:

- Clicking any thumbnail displays the corresponding full-size image.
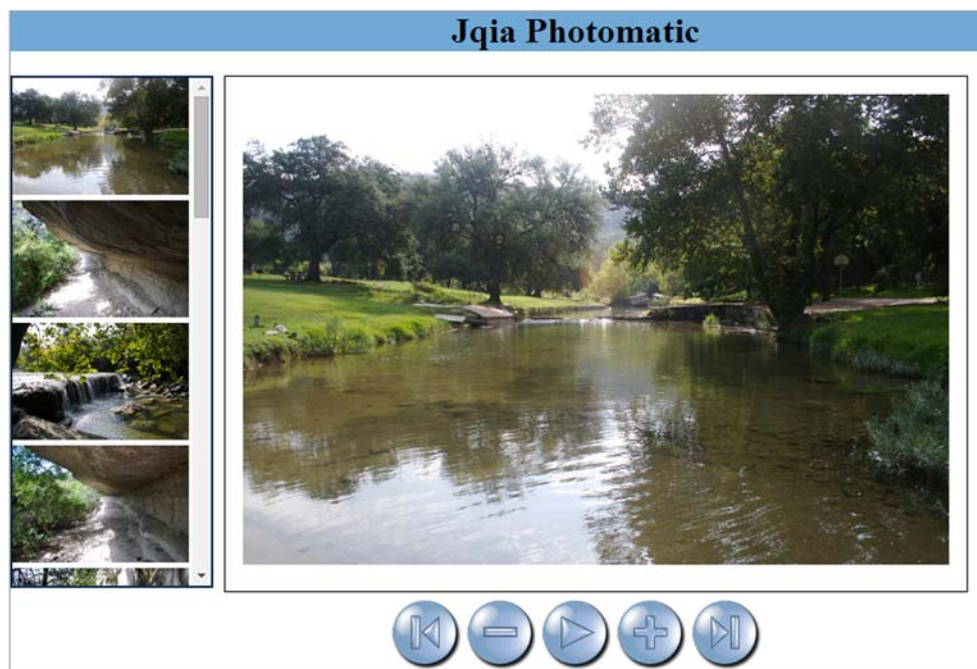
Figure 12.2   The test page that you'll use to put the Jqia Photomatic plugin through its paces

- Clicking the full-size 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
  - Play—Commences moving through the photos automatically until clicked again
- Any operation that moves past the end of the list of images wraps back to the beginning and any operation that moves past the beginning of the list wraps to the end. Clicking Next while on the last image displays the first image, and clicking Previous while on the first image displays the last.

You'll define your plugin so that developers can set up the elements in any manner they like and then tell you which page element should be used for each purpose. Furthermore, in order to give web developers as much leeway as possible, you'll define your plugin so that they can provide any jQuery collection containing images to serve as thumbnails as long as they're gathered together as in our test page.

To start, here's the syntax for the Jqia Photomatic plugin.

Method syntax: jqiaPhotomatic

`jqiaPhotomatic(options)`

Instruments the set of thumbnails, as well as page elements identified in the `options` object hash, to operate as Jqia Photomatic controls.

Parameters

`options`   (Object) An object hash that specifies the options for Jqia Photomatic. See table 12.1 for details.

Returns

The jQuery collection.

Because you have a nontrivial number of parameters for controlling the operation of Jqia Photomatic (some of which can be omitted), you utilize an object hash to pass them as discussed in section 12.3.3. The possible options that you can specify are shown in table 12.1.

Table 12.1   The options for the Jqia Photomatic custom plugin method

Option name	Description
`firstControl`	(Selector) A jQuery selector that identifies the DOM element(s) to serve as a First control. If omitted, no control is instrumented.
`lastControl`	(Selector) A jQuery selector that identifies the DOM element(s) to serve as a Last control. If omitted, no control is instrumented.
`nextControl`	(Selector) A jQuery selector that identifies the DOM element(s) to serve as a Next control. If omitted, no control is instrumented.
`photoElement`	(Selector) A jQuery selector that identifies the `img` element that's to serve as the full-size photo display. If omitted, defaults to the jQuery selector `img.photomatic-photo`.
`playControl`	(Selector) A jQuery selector that identifies the DOM element(s) to serve as a Play control. If omitted, no control is instrumented.
`previousControl`	(Selector) A jQuery selector that identifies the DOM element(s) to serve as a Previous control. If omitted, no control is instrumented.
`transformer`	(Function) A function used to transform the URL of a thumbnail image into the URL of its corresponding full-size photo image. If omitted, the default transformation substitutes all instances of `thumbnail` with `photo` in the URL.
`delay`	(Number) The interval between transitions for an automatic slideshow, in milliseconds. Defaults to 3000.

You'll now create the test page for this plugin before you dive into creating the Jqia Photomatic plugin itself.

12.4.1 Setting up the markup

The first step to perform to create your plugin is to build the page that will use it. The code for this page, available in the file chapter-12/jqia.photomatic.html, is shown in the listing that follows.

Listing 12.4   The test page that creates the Photomatic display in figure 12.2

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Jqia Photomatic - jQuery in Action, 3rd edition</title>
      <link rel="stylesheet" href="../css/main.css"/>
      <link rel="stylesheet" href="../css/jquery.jqia.photomatic.css"/>
   </head>
   <body>
      <h1 class="header">Jqia Photomatic</h1>

      <div id="thumbnails-pane">                              ①  Contains the
         <img src="../images/thumbnails/img_1.jpg" alt="" />     thumbnail
         <img src="../images/thumbnails/img_2.jpg" alt="" />     images
         <img src="../images/thumbnails/img_3.jpg" alt="" />
         <img src="../images/thumbnails/img_4.jpg" alt="" />
         <img src="../images/thumbnails/img_5.jpg" alt="" />
         <img src="../images/thumbnails/img_6.jpg" alt="" />
         <img src="../images/thumbnails/img_7.jpg" alt="" />
         <img src="../images/thumbnails/img_8.jpg" alt="" />
         <img src="../images/thumbnails/img_9.jpg" alt="" />
         <img src="../images/thumbnails/img_10.jpg" alt="" />  ②  Defines
         <img src="../images/thumbnails/img_11.jpg" alt="" />     the image
         <img src="../images/thumbnails/img_12.jpg" alt="" />     element
      </div>                                                      for full-size
                                                                  photos
      <div>
         <img id="photo-display" src="" title="Click for next photo" />
      </div>

      <div id="button-bar">
         <img src="../images/button.first.png" id="first-button"
            alt="First photo" />
         <img src="../images/button.previous.png" id="previous-button"
            alt="Previous photo" />
         <img src="../images/button.play.png" id="play-button"
            alt="Play or pause slideshow" />
         <img src="../images/button.next.png" id="next-button"
            alt="Next photo" />
         <img src="../images/button.last.png" id="last-button"
            alt="Last photo" />
      </div>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script src="../js/jquery.jqia.photomatic.js"></script>
      <script>
         $('#thumbnails-pane img').jqiaPhotomatic({
            photoElement: '#photo-display',
```

Contains the elements to serve as controls ③

④ Invokes the Photomatic plugin

```
                previousControl: '#previous-button',
                nextControl: '#next-button',
                firstControl: '#first-button',
                lastControl: '#last-button',
                playControl: '#play-button',
                delay: 1000
            });
        </script>
    </body>
</html>
```

By applying the principles of unobtrusive JavaScript and keeping all style information in an external style sheet, your markup is tidy and simple.

The HTML markup consists of a container that holds the thumbnail images ❶, an image element (initially sourceless) to hold the full-size photo ❷, and a collection of images ❸ that will control the slideshow. Everything else is handled by your new plugin. The on-page script has a tiny footprint, consisting of a single statement that invokes your plugin, passing a few options ❹.

With this markup in place, it's time to dive into the plugin itself.

12.4.2 *Developing Jqia Photomatic*

To start the development of this plugin you'll use the same skeleton that you used for Jqia Context Menu but with a different namespace:

```
(function($){
    var methods = {
        init: function() {
        }
    };

    $.fn.jqiaPhotomatic = function(method) {
        if (methods[method]) {
            return methods[method].apply(
                this,
                Array.prototype.slice.call(arguments, 1)
            );
        } else if ($.type(method) === 'object') {
            return methods.init.apply(this, arguments);
        } else {
            $.error('Method ' + method +
                ' does not exist on jQuery.jqiaPhotomatic'
            );
        }
    };
})(jQuery);
```

This plugin only needs the initialization function, but to adopt a future-proof approach in case you want to extend it, you'll use a methods variable as you did in the previous project.

Inside the `init()` function, you merge the caller settings with the default ones described in table 12.1. The result is stored in a single `options` object that you can refer to throughout the remainder of the function.

A caller of your plugin may have interest in overriding some of the default values (for example, the `delay` property), so you'll expose them to the external world as shown here:

```
$.fn.jqiaPhotomatic.defaults = {
    photoElement: 'img.photomatic-photo',
    transformer: function(name) {
        return name.replace('thumbnail', 'photo');
    },
    nextControl: null,
    previousControl: null,
    firstControl: null,
    lastControl: null,
    playControl: null,
    delay: 3000
};
```

In the same way you did for the Jqia Context Menu plugin, the merge is performed using jQuery's `$.extend()` method:

```
options = $.extend(true, {}, $.fn.jqiaPhotomatic.defaults, options);
```

After the execution of this statement, the `options` variable will contain the defaults supplied by the inline hash object overridden with any values supplied by the caller.

You also need to keep track of a few other things. In order for your plugin to know what concepts like *next* relative image and *previous* relative image mean, you 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 jQuery collection that this method is operating on—or, at least, it should be. You don't know what the developers collected in the jQuery collection, so you want to filter it down to only image elements. This operation can be done with a selector and jQuery's `filter()` method. But where should you store these two pieces of information?

You could easily create another variable to hold it, but to keep settings together, you'll store them as additional properties of `options`. To do that, you have to slightly modify the call to `extend()` as follows:

```
options = $.extend(
    true,
    {},
    $.fn.jqiaPhotomatic.defaults,
    options,
    {
        current: 0,
        $thumbnails: this.filter('img')
    }
);
```

Note how you place the object containing the current image shown and the list of the thumbnails as the last argument of the extend() method because of how the latter prioritizes the properties to merge. You name the property containing the list as $thumbnails because its value is a jQuery collection.

Now that your initial state is set up, you're ready to move on to the meat of the plugin—adding appropriate features to the controls, thumbnails, and photo display.

> **NOTE**   You're able to keep the state of things because of *closure.* You'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 Jqia Photomatic plugin but also when creating anything but the most trivial of plugins.

Now you need to attach a number of event listeners to the controls and elements that you've identified up to this point. Because the options variable is in scope when you declare the functions that represent those listeners, each listener will be part of a closure that includes the options variable. You can rest assured that even though the latter may appear to be temporary, the state that it represents will stick around and be available to all the listeners that you define.

Speaking of those listeners, here's a list of click event listeners that you'll need to attach to the various elements:

- Clicking a thumbnail photo will cause its full-size version to be displayed.
- Clicking the full-size 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. If the first image of the set was displayed, after clicking the Previous control the last image will be displayed.
- Clicking the Next control will cause the next image to be displayed. If the last image of the set was displayed, after clicking the Next control the first image will 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.
- Clicking the Play control will cause the slideshow to automatically proceed, progressing through the photos using a delay specified in the settings. A subsequent click on the control will stop the slideshow.

Looking over this list, you immediately note that all of these listeners have something in common: they all need to cause the full-size photo of one of the thumbnail images to be displayed. And being the good and clever coder that you are, you want to factor out that common processing into a function so that you don't need to repeat the same code over and over again.

You don't want to infringe on either the global namespace or the $ namespace for a function that should only be called internally from your plugin code. The power of

JavaScript as a functional language comes to your aid once again and allows you to define this new function within the plugin function. By doing so, you limit its scope to within the plugin function itself (one of your goals).

For this reason you define the function needed, named `showPhoto()`, inside the plugin but outside `init()`. The function will define two parameters. The first is the actual options for this call of the plugin, whereas the second is the index of the thumbnail that's to be shown full size. The code of the function is shown here:

```
function showPhoto(options, index) {
   $(options.photoElement).attr(
      'src',
      options.transformer(options.$thumbnails[index].src)
   );
   options.current = index;
}
```

This new function, when passed the index of the thumbnail whose full-size photo has to be displayed, uses the values in the `options` 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 thumb-nail URL to a full-size photo URL.
3 Assign the result of the transformation to the `src` attribute of the full-size image element.
4 Record the index of the displayed photo as the new current index.

With that handy function available, you're ready to define the listeners that we listed earlier. You'll start by adding functionality to the thumbnails themselves, which simply need to cause their corresponding full-size photo to be displayed, as follows:

```
options.$thumbnails.click(function() {
   showPhoto(options, options.$thumbnails.index(this));
});
```

In this handler, you obtain the value of the thumbnail's index by passing the clicked element (referenced by `this`) to jQuery's `index()` method.

Instrumenting the photo display element to show the next photo in the list is just as simple:

```
$(options.photoElement + ', ' + options.nextControl).click(function() {
   showPhoto(options, (options.current + 1) % options.$thumbnails.length);
});
```

You establish a click handler, in which you call the `showPhoto()` function with the `options` object and the next index value. Note how you use the JavaScript modulo operator (`%`) to wrap around to the front of the list when you fall off the end.

The most observant of you may have spotted that you actually include another selector in the statement. The reason is that the behavior of the Next button is exactly the same. You optimize the instruction by using the comma to create a single selector.

The handlers for the First, Previous, and Last controls all follow a similar pattern: figure out the appropriate index of the thumbnail whose full-size photo is to be shown and call `showPhoto()` with that index:

```
$(options.previousControl).click(function() {
   showPhoto(
      options,
      options.current === 0 ?
         options.$thumbnails.length - 1 :
         options.current - 1
   );
});

$(options.firstControl).click(function() {
   showPhoto(options, 0);
}).triggerHandler('click');

$(options.lastControl).click(function() {
   showPhoto(options, options.$thumbnails.length - 1);
});
```

The only line worth a mention in this code is the use of `triggerHandler()`. You call this method to load the initial photo into the image container when the plugin is executed.

The instrumentation of the Play control is somewhat more complicated. Rather than showing a particular photo, this control must start a progression through the entire photo set. Then it has to stop that progression on a subsequent click. Let's take a look at the code you use to accomplish that:

```
var tick;
$(options.playControl).click(function() {
   var $this = $(this);
   if ($this.attr('src').indexOf('play') !== -1) {
      tick = window.setInterval(
         function() {
            $(options.nextControl).triggerHandler('click');
         },
         options.delay
      );
      $this.attr(
         'src',
         $this.attr('src').replace('play', 'pause')
      );
   } else {
      window.clearInterval(tick);
      $this.attr(
         'src',
         $this.attr('src').replace('pause', 'play')
      );
   }
});
```

First you use the image's `src` to decide what operation to perform. If `src` attribute has the string "play" in it, you need to start the slideshow; otherwise you need to stop it.

In the body of the `if`, you employ the JavaScript `setInterval()` method to cause a function to continually fire off using the `delay` value. You store the handle of that interval timer in a varible called `tick` for later reference. Inside the anonymous function passed to `setInterval()`, you emulate a click on the Next control to progress to the next photo; this happens each time the function in `setInterval()` is called.

The last statement inside the `if` updates the `src` attribute of the element to show the pause image. This change allows you to reach the `else` part of the code at the second interaction with the Play button.

In the `else` part you want to stop the slideshow. To do that, you clear the interval timeout using `clearInterval()`, passing `tick`, and restore the play image.

As the final task, and in order to fulfill the wisdom of maintaining chainability, you need to return the original set of matched elements. You achieve this with

```
return this;
```

Take a moment to do a short victory dance; you're finally finished! Bet you didn't think it would be that easy. The completed plugin code, which you'll find in the file js/jquery.jqia.photomatic.js, is shown in the following listing.

Listing 12.5   The complete implementation of the Jqia Photomatic plugin

```
(function($) {

   function showPhoto(options, index) {
      $(options.photoElement).attr(
         'src',
         options.transformer(options.$thumbnails[index].src)
      );
      options.current = index;
   }

   var methods = {
      init: function(options) {
         options = $.extend(
            true,
            {},
            $.fn.jqiaPhotomatic.defaults,
            options,
            {
               current: 0,
               $thumbnails: this.filter('img')
            }
         );

         options.$thumbnails.click(function() {
            showPhoto(options, options.$thumbnails.index(this));
         });

         $(options.photoElement + ', ' + options.nextControl).click(
            function() {
               showPhoto(
                  options,
                  (options.current + 1) % options.$thumbnails.length
```

```
       $.error(
          'Method ' + method +
          ' does not exist on jQuery.jqiaPhotomatic'
       );
    }
};

$.fn.jqiaPhotomatic.defaults = {
    photoElement: 'img.photomatic-photo',
    transformer: function(name) {
       return name.replace('thumbnail', 'photo');
    },
    nextControl: null,
    previousControl: null,
    firstControl: null,
    lastControl: null,
    playControl: null,
    delay: 3000
};
})(jQuery);
```

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.

Also note that because you took such care to not let state leak out of the plugin, you're free to add as many Jqia Photomatic widgets to a page as you like, without fear that they'll interfere with one another (taking care, of course, to make sure you don't use duplicate ID values in the markup).

But is it complete? You be the judge and consider the following exercises:

- The transition from photo to photo is instantaneous. Using your knowledge from the animation and effect chapter, change the plugin so that photos cross-fade into one another.
- Going one step further, how would you go about allowing the developer to use a custom animation of choice?
- For maximum flexibility, you coded this plugin to take advantage of HTML elements already created by the user. How would you create an analogous plugin, but with less display freedom, that generated all the required HTML elements on the fly?

Now that you know all about implementing a new jQuery method, it's time to learn more about creating custom utility functions.

12.5   *Writing custom utility functions*

In this book, we've used the term *utility function* to describe functions defined as properties of jQuery (and therefore $). These functions are usually intended to act on non-element JavaScript objects or to perform some other operation that doesn't

specifically operate on any objects. Some examples of utility functions you've seen are `$.each()` and `$.noConflict()`. In this section, you'll learn how to add your own custom utility functions.

Adding a function as a property to an object or a function is as easy as declaring the function and assigning it. 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 utility functions isn't without pitfalls. If some developer includes this function on a page that uses Prototype and has called `$.noConflict()`, rather than adding a jQuery extension the developer would create a method on Prototype's `$()` function. (Get thee to the appendix if the concept of a *method* of a function makes your head hurt.) As you can see, unlike plugins that operate on a set of matched elements, a utility function is assigned to `$` and not to `$.fn`.

We've already described this issue and its solution in section 12.3.2 (hint: create an IIFE). Discussing a utility function like the one just shown isn't a big deal, so let's implement and examine a nontrivial one.

12.5.1 *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` object 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 of the month as a 0-filled, 2-character field
`d`: the day of the month
`EEEE`: the full name of the day of the week
`EEE`: the abbreviated name of the day of the week
`a`: the meridian (AM or PM)
`HH`: the 24-hour clock hour in the day as a 2-character, 0-filled field

---

Function syntax: $.formatDate *(continued)*

`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 the following listing. We're not going to go into too much detail regarding the algorithm used to perform the formatting because that isn't the point of this chapter. We'll use this implementation to point out some interesting tactics that you can use when creating a somewhat complex utility function.

Listing 12.6 Implementation of the `$.formatDate()` utility function

❶ **Defines the regular expression
to match tokens in the pattern**

```
(function($) {
  var patternParts =
/^(yy(yy)?|M(M(M(M)?)?)?|d(d)?|EEE(E)?|a|H(H)?|h(h)?|m(m)?|s(s)?|S)/;
  var 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) {
      return toFixedWidth(date.getMonth() + 1, 2);
    },
    M: function(date) {
      return date.getMonth() + 1;
    },
    dd: function(date) {
      return toFixedWidth(date.getDate(), 2);
```

❷ **Defines an object
containing formatting
function to use once a
specific match is found**

```
       },
       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';          ❸ A function to format
       }                                                          the passed value as a
   };                                                             fixed-width field of
                                                                  the specified length
   function toFixedWidth(value, length, fill) {    ◄──────
Assigns ❹  var result = (value || '').toString();
default    fill = fill || '0';
value      var padding = length - result.length;    ◄───── ❺ Computes padding
           if (padding < 0) {
              result = result.substr(-padding);     ◄───── ❻ Truncates if necessary
           } else {
              for (var n = 0; n < padding; n++) {
                 result = fill + result;          ◄──────── ❼ Pads result
              }
           }
           return result;                         ◄──────── ❽ Returns final result
   }
```

```
$.formatDate = function(date, pattern) {              ◁──────  Implements
    var result = [];                                          main body of
    while (pattern.length > 0) {                      ❾        the function
        patternParts.lastIndex = 0;
        var matched = patternParts.exec(pattern);
        if (matched) {
            result.push(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('');
};                                                      ❿  Provides name of the months
$.formatDate.monthNames = [                         ◁──┘
    'January', 'February', 'March', 'April', 'May', 'June', 'July',
    'August', 'September', 'October', 'November', 'December'
];                                                      ⓫  Provides name of the days
$.formatDate.dayNames = [                           ◁──┘
    'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
    'Saturday'
];
})(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 anonymous function assigned to `$.formatDate` ❾ needs some ancillary arrays, objects and functions to do its job. In particular:

- A regular expression used to match tokens in the pattern ❶
- A list of the English names of the months ❿
- A list of the English names of the days ⓫
- A set of subfunctions designed to provide the value for each token type, given a source date ❷
- A private function that formats the passed value as a fixed-width field of the specified length ❸

In this utility function you define several variables and functions. Some of them are private (declared using the `var` keyword), whereas others are exposed to the external world (defined as property of `$.formatDate`). The `patternParts` variable is only needed by your function so it's nonsense to expose it, and it's kept private. On the other hand, `monthNames` ❿ and `dayNames` ⓫ may be overwritten to provide the names of the months and the days in another language, so you allow accessing them from outside the function. Remember, JavaScript functions are first-class objects, and they can have their own properties like any other JavaScript object.

Your utility function relies on a support function called `toFixedWidth()`. The passed value to this function is converted to its string equivalent, and the fill character

is determined either from the passed value or the default of 0 **4**. Then you compute the amount of padding needed **5**.

If you end up with negative padding (the result is longer than the passed field length), you truncate from the beginning of the result to end up with the specified length **6**; otherwise, you pad the beginning of the result with the appropriate number of fill characters **7** prior to returning it as the result of the function **8**.

And the formatting algorithm itself? In a nutshell, it operates as follows:

1 Creates an array to hold portions of the result.

2 Iterates over the pattern, supplied as the second argument to the utility function, consuming identified token and non-token characters until the string has been completely inspected.

3 Resets the regular expression (stored in `patternParts`) on each iteration by setting its `lastIndex` property to 0.

4 Tests the regular expression for a token match against the current beginning of the pattern.

5 Calls the function in the `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.

6 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.

7 Joins the results array into a string and returns it as the value of the function when the entire pattern has been consumed.

You'll find this extension in the file js/jquery.jqia.formatDate.js and a simple page to test it at chapter-12/jqia.formatDate.html.

12.6   *Summary*

This chapter introduced you to writing reusable code that extends jQuery. Writing your own code as extensions to jQuery has a number of advantages. Not only does it keep your code consistent across your web application regardless of whether it's employing jQuery APIs or your own, but it also makes all of the power of jQuery available to your own code.

Following a few naming rules helps avoid naming collisions between filenames and with other plugins' code, as well as problems that might be encountered when the $ name is reassigned by a page that will use your plugin. In addition, you've seen how you can build plugins that don't break jQuery's chainability.

Creating new utility functions is as easy as creating new function properties on $, and new jQuery methods are 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 plugins, and you'll even learn new techniques.

Having yet more jQuery knowledge at your disposal, let's move on to learning how you can use jQuery to better manage asynchronous functions.

13

*Avoiding the callback
hell with Deferred*

This chapter covers

- What promises are and why they're important
- The `Deferred` object
- How to manage multiple asynchronous operations
- Resolving and rejecting a promise

For a long time, JavaScript developers have used callback functions to perform several tasks such as running operations after a given amount of time (using `set-Timeout()`), or at regular intervals (using `setInterval()`), or to react to a given event (`click`, `keypress`, `load`, and so on). We've discussed and employed callbacks extensively to perform asynchronous operations; for example, in chapter 6 where we focused on events, in chapter 8 when we talked about animations, and in chapter 10 where we covered Ajax. Callback functions are simple and get the job done, but they become unmanageable as soon as you need to execute many asynchronous operations, either in parallel or in sequence. The situation where you have a lot of nested callbacks, or independent callbacks that have to be synchronized, is often referred to as the "callback hell."

Today, websites and web applications are often powered by JavaScript code more than backend code only (this is the era of API-driven services, isn't it?). For this reason, developers need a better way to manage and synchronize asynchronous operations to keep their code readable and maintainable.

In this chapter we'll discuss promises, what they are, and how jQuery allows you to use them. Our lovely library implements promises (the concept) through the use of two objects: `Deferred` and `Promise`. How jQuery implements promises has been the subject of discussions, criticisms, and a lot of changes, as you'll learn in the next pages.

While progressing through this chapter, you'll note that the terminology might be confusing and that the concept of promises doesn't map one to one with the `Promise` object, which admittedly is weird. You'll also need to learn several new terms. Don't be scared, though. The text is filled with several examples and extensive explanations that will help you with the learning process.

13.1 Introduction to promises

In real life, the one away from computers (yes, there is something more out there), we often talk about promises. You ask people to make you a promise or you're asked by others to promise something. Intuitively, a promise is a commitment derived from a request you make to ask a person to perform a given action at some point in the future. Sometimes the action can be executed very soon; sometimes you have to wait for a while. In an ideal world, a promise would always be kept. Unfortunately, our world is far from being ideal, so it happens that a promise is sometimes unfulfilled, no matter the reason. Regardless of the time it'll take and the final result, a promise doesn't block you from doing something else in the meantime like reading, cooking, or working.

In JavaScript, a *promise* is exactly this. You request a resource from another website, the result of a long calculation either from your server or from a JavaScript function, or the response of a REST service (these are examples of promises), and you perform other tasks while you await the result. When the latter becomes available (the promise is *resolved/fulfilled*) or the request has failed (the promise is *failed/rejected*), you act accordingly.

The previous description has hopefully given you a good understanding of what promises are. But they have a much more formal definition. Promises have been widely discussed and such discussions have resulted in two proposals: Promises/A and Promises/A+. It's important that we discuss these two proposals before delving into the jQuery way of dealing with promises so that you can have a better understanding of promises as a concept.

The specifications of Promises/A+ can be found at http://promisesaplus.com/, but in summary, they define a promise in this way:

> A promise represents the eventual result of an asynchronous operation. The primary way of interacting with a promise is through its `then` method, which registers callbacks to receive either a promise's eventual value or the reason why the promise cannot be fulfilled.

PROMISES/A+ SPECIFICATIONS

The then() method described by the Promises/A+ proposal is the core of promises. The then() method accepts two functions: one to execute in the event that the promise is fulfilled and the other if the promise is rejected. When a promise is in one of these states, regardless of which one, it's *settled*. If a promise is neither fulfilled nor rejected (for example, if you're still waiting for the response of a server calculation), it's *pending*.

Even if the formal definition mentions asynchronous operations, it's worth noting that a promise can also be resolved or rejected by a synchronous operation, so you can use promises for more than Ajax requests, as we'll discuss in detail later in this chapter.

Sometimes the operation, either synchronous or asynchronous, may already be completed and thus the value returned or the reason why the promise was rejected is already available. In this case, if you register a function using the then() method, this function will be executed immediately. This is another important difference between how promises act compared to callback functions in response to an event. Remember that if you add a handler for an event that has already been fired, the callback won't be executed.

Now that you've read about promises and the then() method, it's crucial that you understand why promises are so good and why they can help you in your work. To prove it, we'll discuss and solve a simple scenario, similar to others you might encounter in your job. We'll first approach the problem using the knowledge you've acquired so far, such as callback functions, and then we'll iterate on the code to help you reach a better solution in terms of readability and maintainability by using promises.

Let's say that you're building a web page that uses a third-party web service called Randomizer. This service provides an API that returns a random number every time it's called. You want your web page to retrieve two numbers and sum them. Once finished, you want to show the result in an element having content as its ID. To accomplish this goal you need to perform two Ajax requests, synchronize the callback functions in some way (you'll use a support function), and finally show the result. The most complex part of the code is the second: the synchronization of the two independent Ajax requests.

The code of the web page we're discussing is shown in the following listing. Please note that executing this page won't give you any result because you're performing requests to a nonexistent Randomizer service, but the code developed will help you in understanding the importance of promises.

Listing 13.1   Implementing multiple asynchronous operations with callbacks

```
var number1, number2;

function handler() {                              Defines the function that both
   var $content = $('#content');                  the Ajax callbacks will invoke

   if (number1 === null || number2 === null) {
      $content.text('An error has occurred, try again later.');      Prints the
   } else if (number1 !== undefined && number2 !== undefined) {      result on
      $content.text(number1 + number2);                              the page
```

```
   }
}
$.ajax(http://www.randomizer.com/number', {          ◄─────────┐  Performs the first
   success: function(data, status) {                             Ajax request
      number1 = (status === 'success') ? parseInt(data, 10) : null;
      handler();                                       ◄────────┐
   },                                                            Calls the support
   error: function() {                                           function that
      number1 = null;                                            synchronizes
      handler();                                     ◄──────────┘ the callbacks
   }
});
$.ajax('http://www.randomizer.com/number', {        ◄─────────┐  Performs the second
   success: function(data, status) {                             Ajax request
      number2 = (status === 'success') ? parseInt(data, 10) : null;
      handler();                                      ◄────────┐
   },                                                            Calls the support
   error: function() {                                          function that
      number2 = null;                                           synchronizes
      handler();                                     ◄─────────┘ the callbacks
   }
});
```

The previous listing is pretty simple but it has a big issue: you need to introduce a variable for every callback function. In this case you need only two, but as the project grows, the situation could become unmanageable. In addition, let's assume that the previous code was the body of a function invoked by another function called foo. How could you return the result of the two Ajax requests to the foo function? With the current approach, you can't without introducing some global variables. Finally, what if you had to make two Ajax requests, with the second starting after the first is completed? In this case, you'd need to have a callback inside a callback. With more and more callbacks coming into play, the code would become a complete mess, and that's why this situation is known as the "callback hell."

The example discussed has given you an idea of what the problem is with callback functions. You can improve this code using promises, resulting in several advantages. ECMA International (http://www.ecma-international.org/), the group behind the JavaScript specifications, has decided to introduce promises in the next version of JavaScript (ECMAScript 2015, also known as ECMAScript 6) and to adhere to the Promises/A+ proposal. Some modern browsers already support them, but others don't and older browsers never will. Therefore, if you want to write clean, readable, and maintainable code by using the promises approach, you need to rely on a polyfill or a library in case you want more functions than those offered by the standard.

jQuery helps you avoid all these browser issues but, depending on the version of the library you're using, it might do it in a slightly different manner. jQuery provides you with two objects, Deferred and Promise, that you can reliably use in your projects. But before we introduce them, we need to split the discussion in order to help you to easily follow the concepts of this chapter.

jQuery's 1.x and 2.x implementation adheres to the CommonJS Promises/A proposal (http://wiki.commonjs.org/wiki/Promises/A) that was used as a base for Promises/A+. Therefore, there are differences in how you can use promises in pure JavaScript and in jQuery 1.x and 2.x. Moreover, because jQuery follows a different proposal in these branches, the library is incompatible with other ones. The Promises/A proposal defines a promise in this way:

> A promise represents the eventual value returned from the single completion of an operation. A promise may be in one of the three states, unfulfilled, fulfilled, and failed. The promise may only move from unfulfilled to fulfilled, or unfulfilled to failed.

PROMISES/A SPECIFICATIONS

As you can see, the terminology for the `Promise` object's definition, illustrated in figure 13.1, is a bit different. The Promises/A proposal defines the unfulfilled, fulfilled, and failed states, whereas the Promises/A+ proposal uses the pending, fulfilled, and rejected states.
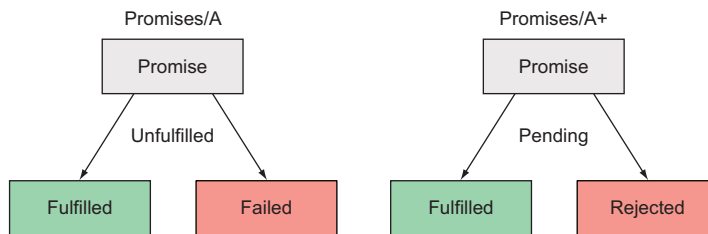
Figure 13.1   The terminology differences of the Promises/A and the Promises/A+ proposals

These proposals outline the behavior of promises and not the implementation, so libraries implementing promises have a `then()` method in common but may differ for other methods exposed.

In jQuery 3.x the interoperability with a standard promise (as implemented in ECMAScript 2015, which follows the Promises/A+ proposal) has been improved. The signature of the `then()` method is still a bit different for backward compatibility but the behavior is more in line with the standard. Don't worry if this is confusing at first; we'll highlight all the differences and also provide you with many examples.

In the next section, you'll learn more about the `Deferred` and `Promise` objects and, where necessary, we'll also highlight other differences between the Promises/A and the Promises/A+ proposals.

13.2   *The Deferred and Promise objects*

The `Deferred` object was introduced in jQuery 1.5 as a chainable utility used to register multiple callbacks into callback queues, invoke callback queues, and relay the success or failure state of any synchronous or asynchronous function. Since then, it has

been subject to discussions, some criticisms, and a lot of changes along the way.[1] This object can be used for many asynchronous operations, like Ajax requests and animations, but also with JavaScript timing functions. Together with the `Promise` object, it represents the jQuery implementation of promises.

The `Promise` object is created starting from a `Deferred` object or a jQuery object and possesses a subset of the methods of the `Deferred` object (`always()`, `done()`, `fail()`, `state()`, and `then()`). `Deferred` objects are typically used if you write your own function that deals with asynchronous callbacks. So, your function is the *producer* of the value and you want to prevent users from changing the state of the `Deferred`.

In chapter 10 we covered the utility functions that jQuery provides to work with Ajax requests, but for your convenience, we omitted to say that the returned value of those functions, which is a `jqHXR` object as you might remember, implements the `Promise` interface. For this reason they're sometimes referred to as `Promise`-compatible objects, and you can call all the methods of regular `Promise` objects on them. This will allow you to write cleaner and more readable code.

Let's now discuss how to work with `Deferred`s and `Promise`s.

13.3 *The Deferred methods*

In jQuery, a `Deferred` object is created by calling the `$.Deferred()` constructor. The syntax of this function is as follows.

Method syntax: $.Deferred

`$.Deferred([beforeStart])`

A constructor function that returns a chainable utility object with methods to register multiple callbacks into callback queues, invoke callback queues, and relay the success or failure state of any synchronous or asynchronous function. It accepts an optional function to execute before the constructor returns.

Parameters

beforeStart   (Function) A function that's called before the constructor returns. The function accepts a `Deferred` object used as the context (`this`) of the function.

Returns

The `Deferred` object.

The `Deferred` object allows for chaining, like many of the jQuery methods we've discussed so far, but it has its own methods. You'll never find yourself writing a statement like this:

```
$.Deferred().html('Promises are great!');
```

---

[1] http://bugs.jquery.com/ticket/11010.
"You're Missing the Point of Promises" by Domenic Denicola: http://domenic.me/2012/10/14/youre-missing-the-point-of-promises/.
"Javascript promises and why jQuery implementation is broken" by Valerio Gheri: https://thewayofcode.wordpress.com/tag/jquery-deferred-broken/.
https://github.com/jquery/jquery/issues/1722.

Having created a new `Deferred` object isn't of any utility if you don't know how to use it. Starting from the next section, we'll cover the methods exposed by this object.

13.3.1 Resolving or rejecting a Deferred

One of the first concepts we discussed in this chapter is the state in which a promise can be. In jQuery, a promise can be resolved (the promise is successful), rejected (an error occurred), or pending (the promise is neither resolved nor rejected). These states can be reached in two ways. The first is determined by code that you or another developer has written and with an explicit call to methods like `deferred.resolve()`, `deferred.resolveWith()`, `deferred.reject()`, or `deferred.rejectWith()`. These methods, as we'll discuss in a few moments, allow you to either resolve or reject a promise. The second is determined by the success or the failure of a jQuery function—for example, `$.ajax()`—so you don't have to call any of the previously mentioned methods yourself.

Before you can write any code that uses the `Deferred` object, you have to get acquainted with these methods, so let's discover their syntax. The syntax of the `deferred.resolve()` method is shown here.

Method syntax: deferred.resolve

`deferred.resolve([argument, ..., argument])`

Resolve a `Deferred` triggering the execution of any successful callback defined, passing any given `argument`. This method accepts an arbitrary number of arguments.

Parameters

argument   (Any) An optional argument of any type that is passed to the success callback
            functions defined.

Returns

The `Deferred` object.

The syntax of the `deferred.resolveWith()` method is shown here.

Method syntax: deferred.resolveWith

`deferred.resolveWith(context[, argument, ..., argument])`

Resolve a `Deferred` triggering the execution of any successful callback defined, passing any given `argument`, and setting `context` as their context.

Parameters

context   (Object) The object to set as the context of the successful callbacks.

argument   (Any) An optional argument of any type that is passed to the success callback
            function defined.

Returns

The `Deferred` object.

Sometimes it happens that a promise needs to be rejected. For such circumstances you can employ the `deferred.reject()` method. Its syntax is shown here.

Method syntax: deferred.reject

`deferred.reject([argument, ..., argument])`

Reject a `Deferred` triggering the execution of any failure callback defined, passing any given `argument`. This method accepts an arbitrary number of arguments.

Parameters

argument   (Any) An optional argument of any type that is passed to the failure callback function defined.

Returns

The `Deferred` object.

In the same way that jQuery defines a method to set the context for the successful callbacks, you can set it for failure callbacks by using the `deferred.rejectWith()` method. The syntax of this method is reported here.

Method syntax: deferred.rejectWith

`deferred.rejectWith(context[, argument, ..., argument])`

Reject a `Deferred`, triggering the execution of any failure callback defined, passing any given `argument`, and setting `context` as their context.

Parameters

context   (Object) The object to set as the context of the failure callbacks.
argument   (Any) An optional argument of any type that is passed to the failure callback function defined.

Returns

The `Deferred` object.

Up to this point you've learned how to create a `Deferred` and how to resolve or reject it, but the fun comes when you write code to react to the change in a `Deferred`'s state. Let's see how.

13.3.2 *Execute functions upon resolution or rejection*

Usually you want to know when a `Deferred` is resolved to perform some actions. To do that, you can employ the `deferred.done()` method. It accepts one or more arguments, all of which can be either a single function or an array of functions. These callback functions are executed in the order in which they were added. The syntax of this method is shown here.

Method syntax: deferred.done

`deferred.done(callbacks[, callbacks, ..., callbacks])`

Add handlers that are called when the `Deferred` object is resolved. This method accepts an arbitrary number of callbacks with a minimum of one.

Parameters

callbacks   (Function|Array) A function or array of functions that is called when the
                `Deferred` is resolved.

Returns

The `Deferred` object.

In the same way that you can execute operations when a `Deferred` object is resolved, you can run functions when it's rejected. To do that, you can use the `deferred .fail()` method. Its syntax is as follows.

Method syntax: deferred.fail

`deferred.fail(callbacks[, callbacks, ..., callbacks])`

Add handlers that are called when the `Deferred` object is rejected. This method accepts an arbitrary number of callbacks with a minimum of one.

Parameters

callbacks   (Function|Array) A function or array of functions that is called when the
                `Deferred` is rejected.

Returns

The `Deferred` object.

In this case, too, when the `Deferred` is rejected, the callbacks are executed in the order in which they were added.

Now that we've introduced you to these methods, we're able to show you some code to demonstrate what this fuss is all about. To start simply, we'll modify the example we discussed at the beginning of this chapter and rewrite it to use `Deferred`s. This time we want to provide you with a working demo, so we'll add in a simple PHP page, called "integer.php," to simulate the Randomizer service. The resulting code is shown in the next listing and is also available in the file chapter-13/randomizer.1.html.

Listing 13.2   Using `Promises` with Ajax requests, version 1

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Using Promises with Ajax requests version 1</title>
        <link rel="stylesheet" href="../css/main.css" />
    </head>
    <body>
        <p>
```

Then you attach the same success and failure callbacks that were developed in listing 13.1. The callbacks to execute if the Ajax request is successful are added by calling the done() method on the variables promise1 ❷ and promise2 ❺. The same thing happens to the failure callbacks that are added by calling the method fail() on promise1 ❸ and promise2 ❻.

While we're talking about done() and fail(), we want to highlight that if you add a success or failure callback *after* the state of a Deferred is changed to either resolved or rejected, the callback will be executed immediately. Consider the following code:

```
var promise1 = $.ajax('integer.php');
setTimeout(function() {
   promise1.done(function(data, status, jqXHR) {
      // Code here
   })
   .fail(function() {
      // Code here
   });
}, 5000);
```

In this case you delay the statement that adds the callbacks by five seconds to simulate a long enough time for the "integer.php" page to be executed and its response returned (this isn't guaranteed, but it's enough for the sake of the example). Based on this assumption, at the time done() and fail() are invoked to add the callbacks, the state of promise1 is already defined. What you might expect, because of your experience with listeners added to events, is that none of them will be executed. The reason is that the "event" of changing the state of the promise has already happened. But one of the two functions will still run, which is an important difference.

Another interesting difference is that you can add as many callbacks as you like with one statement. Let's say that if an Ajax request is successful, you want to execute two functions, foo() and bar(). In a traditional approach, you'd write code like the following:

```
$.ajax('integer.php', {
   success: function(data, status, jqXHR) {
      foo(data, status, jqXHR);
      bar(data, status, jqXHR);
   }
);
```

Using the done() method, you can rewrite it in a single line of code:

```
$.ajax('integer.php').done(foo, bar);
```

Or equivalently (note that here you pass an array of functions):

```
$.ajax('integer.php').done([foo, bar]);
```

Much better, isn't it?

Listing 13.2 employs some of the new methods you've learned, but it still suffers from the awkward synchronization approach used. Let's see how to do better.

13.3.3 *The when() method*

In order to edit listing 13.2 to its final version, we need to introduce you to another utility function: `$.when()`. It provides a simple way to execute callback functions based on one or more objects, usually `Deferred` or `Promise`-compatible objects representing asynchronous events. This is exactly what you need in your code because you have two `Promise`-compatible objects returned by the two `$.ajax()` calls. But before using it, let's discover its syntax and the parameters it accepts.

Method syntax: $.when

`$.when(object[, object, ..., object])`

Provides a way to execute callback functions based on one or more objects, usually `Deferred` or `Promise`-compatible objects representing asynchronous events. This function accepts an arbitrary number of objects with a minimum of one.

Parameters

`object`   (Deferred|Promise|Object) A `Deferred`, `Promise`, `Promise`-compatible, or JavaScript object. If a JavaScript object is passed, it's treated as a resolved `Deferred`.

Returns

A `Promise` object.

The `$.when()` utility function has an interesting point to highlight: it doesn't return a `Deferred` object but a `Promise` object. What's returned by this method can't be resolved or rejected; you can only call `done()`, `fail()`, and a few other methods on it. It's worth noting that in ECMAScript 2015 there's a similar method called `Promise.all()`.

If you pass a single `Deferred` object to `$.when()`, its `Promise` object is returned; otherwise a new `Promise` is created starting from a "master" `Deferred` that will keep track of the state of all the objects (`Promise`, `Promise`-compatible, `Deferred` objects, and so on) passed to `$.when()`.

`$.when()` causes the execution of the success callbacks (the functions to run in the event of success) when and if all the objects passed to this utility function are resolved (in the case of `Deferred`s, `Promise`s, and `Promise`-compatible objects) or can be considered resolved (any other type of objects). Conversely, it causes the execution of the failure callbacks as soon as one of the `Deferred`s is rejected or one of the `Promise` or `Promise`-compatible objects is in a rejected state. The arguments passed to the callback functions, whether for success or failure, are the ones passed to either `resolve()` or `reject()`, depending on the case.

Before we lose you in this sea of information, let's see a concrete example. As we mentioned earlier, we'll rewrite the code of listing 13.2 and try to improve it by using `Deferred`s. The final result is shown in the next listing and is also available in the file chapter-13/randomizer.2.html.

Listing 13.3   Using `Promises` with Ajax requests, version 2

```html
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Using Promises with Ajax requests version 2</title>
      <link rel="stylesheet" href="../css/main.css" />
   </head>
   <body>
      <p>
         The generated random numbers are <span id="number1"></span>
         and <span id="number2"></span>.
         Their sum is <span id="sum"></span>
      <p>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         function success(params1, params2) {
            var number1 = (params1[1] === 'success') ?
                  parseInt(params1[0], 10) : null;

            var number2 = (params2[1] === 'success') ?
                  parseInt(params2[0], 10) : null;

            if (number1 === null || number2 === null) {
               fail();
               return;
            }

            $('#number1').text(number1);
            $('#number2').text(number2);
            $('#sum').text(number1 + number2);
         }

         function fail() {
            $('#sum').text('An error has occurred, try again later.');
         }

         $.when($.ajax('integer.php'), $.ajax('integer.php'))
            .done(success)
            .fail(fail);
      </script>
   </body>
</html>
```

① Defines a success callback

② Defines a failure callback

③ Creates a new Promise starting from a "master" Deferred created internally by $.when(), which is based on the $.ajax() calls

④ Sets the success() callback using the deferred.done() method

⑤ Sets the fail() callback using the deferred.fail() method

Looking at this code should make you feel happy and your eyes should shine. Thanks to `$.when()`, we've highly simplified the code of listing 13.2 and made it more readable and manageable. Let's analyze the code.

The key point is the use of `$.when()`, which you employ to solve the issue you had with the synchronization of the results of the Ajax requests ③. By doing so, you don't have to set the success and failure functions for each of them. In fact, you'll only set

them on the Promise object returned by $.when() through the use of done() ❹ and fail() ❺. Once again, we want you to remember that a Promise object is created starting from a Deferred object or a jQuery object (in this case it's created internally by $.when()) and possesses a subset of the methods of the Deferred (always(), done(), fail(), state(), and then()).

The success() function is executed when both of the Promise-compatible objects are fulfilled. Its behavior isn't changed compared to the previous version, but there is an interesting detail to discuss. You define two parameters for this function, params1 and params2, because this is the number of the Promise-compatible objects you're using ❶. Each of these parameters is an array containing the usual parameters passed to the success callback of a $.ajax(), $.get(), or $.post() function: data, statusText, and jqXHR. It's worth noting that if the value passed was a single object, it wouldn't be wrapped.

The last function defined in the listing is fail() ❷. It's extracted from the previous listing's handler() function and it's executed as soon as one of the Ajax requests fails.

In addition to resolving or rejecting a Deferred, you can also give a notification about the progress of the process.

13.3.4 *Notifying about the progress of a Deferred*

Sometimes you may have some code that needs to know the state of a Deferred. For example, if you're retrieving data asynchronously, you want to know the percentage of completion of the process. If this process is promise-based, you may have a function waiting for either the resolution or the rejection of that promise, which you want to keep informed about its state. For such occasions, you can employ deferred.notify(). The syntax of this method is as follows.

Method syntax: deferred.notify

`deferred.notify([argument, ..., argument])`

Triggers the execution of any progress callback defined, passing any given argument. This method accepts an arbitrary number of arguments.

Parameters

argument   (Any) An optional argument of any type that is passed to the progress callback functions defined.

Returns

The Deferred object.

In case you want to force the context of the callback functions executed, you can use deferred.notifyWith().

Method syntax: deferred.notifyWith

`deferred.notifyWith(context[, argument, ..., argument])`

Triggers the execution of any progress callback defined, passing any given `argument` and setting `context` as their context

Parameters

`context`   (Object) The object to set as the context of the progress callbacks

`argument`   (Any) An optional argument of any type that is passed to the progress callback
                functions defined

Returns

The `Deferred` object

Thanks to these methods, you're now ready to see how you can perform some actions while an operation is in progress.

13.3.5 *Follow the progress*

With the methods discussed in the previous section, you can be notified about the progress of an asynchronous operation. But this is completely useless if you can't "listen" for these updates. Enter the `deferred.progress()` method.

Method syntax: deferred.progress

`deferred.progress(callbacks[, callbacks, ..., callbacks])`

Add handlers that are called when the `Deferred` object generates progress notifications. This method accepts an arbitrary number of callbacks with a minimum of one.

Parameters

`callbacks`   (Function|Array) A function or array of functions that is called when the
                `Deferred` object generates progress notifications.

Returns

The `Deferred` object.

Now that you know how this method works, let's fix the idea with an example. Let's say that you want to create an animation for a progress bar and you want to be able to keep track of the progress of the animation to display the percentage of completion. To do that you can use the `deferred.progress()` and the `deferred.notify()` methods.

> **NOTE**   The example we're going to develop isn't ideal. A better way would be to return the `Promise` object of the `Deferred` used and let the caller of the animation function update the percentage. We'll modify it in the next section, but for the moment we want to keep things as simple as possible and move in little steps, so bear with us.

The code that implements the previous requirements is reported in the following listing and available as a JS Bin (http://jsbin.com/yohiho/edit?html,css,js,output). You can find it in the file chapter-13/deferred.progress.1.html.

Listing 13.4 Using `deferred.progress()`, version 1

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Using deferred.progress() version 1</title>
        <link rel="stylesheet" href="../css/main.css" />
        <style>
            .bar,
            .inner-bar
            {
                height: 50px;
                border-radius: 3px;
            }

            .bar
            {
                position: relative;
                border: 1px solid #000000;
                background-color: #CCCCCC;
            }

            .inner-bar
            {
                width: 0%;
                background-color: #F72F39;
            }

            .progress
            {
                position: absolute;
                font-size: 30px;
                width: 100%;
                text-align: center;
                top: 10px;
            }

            button
            {
                margin-top: 15px;
            }
        </style>
    </head>
    <body>
        <div class="bar">
            <div class="inner-bar"></div>
            <span class="progress">0%</span>
        </div>

        <button id="run-button">Run animation</button>

        <script src="../js/jquery-1.11.3.min.js"></script>
        <script>
            function animate(milliseconds) {
                var $innerBar = $('.inner-bar').width(0);
                var $barWidth = $('.bar').width();
                var step = $barWidth / 100;
```

❶ Defines a simple progress bar

❷ Defines the function that animates the progress bar

Creates a new Deferred, adding a progress callback ❸

```
var deferred = $.Deferred().progress(function (value) {
   $('.progress').text(Math.floor(value) + '%');
});

var idInterval = setInterval(
   function () {
      // Can't use width() to retrieve the width because
      // of issue #1724
      // (https://github.com/jquery/jquery/issues/1724)
      var nextWidth =
         parseFloat($innerBar.get(0).style.width) + step;
      deferred.notify(nextWidth * 100 / $barWidth);
      $innerBar.width(nextWidth);
      if (nextWidth >= $barWidth) {
         deferred.resolve();
         clearInterval(idInterval);
      }
   },
   milliseconds / 100
);
}

$('#run-button').click(function() {
   animate(1000);
});
</script>
</body>
</html>
```

Notifies the Deferred at each step of the animation, passing the percentage of completion ❹

Resolves the Deferred when the animation is completed ❺

In this example you create a simple progress bar that shows the percentage of completion ❶. Inside the script element, you define a function called `animate()` that animates the progress bar ❷. It accepts the number of milliseconds the animation has to last. Inside it, you instantiate a new `Deferred` object and add a progress callback that updates the percentage of completion ❸.

Using JavaScript's `setInterval()` function, you set up the core of the `animate()` function where you calculate the next step of the animation, which will always be to add another `$barWidth/100` to the bar width, and notify the `Deferred` ❹. Finally, when the animation is completed you resolve the `Deferred` using the `deferred.resolve()` method ❺.

This example has shown you how to use the `deferred.progress()` method, but you might struggle to understand why you don't place the statement to update the percentage inside the function passed to `setInterval()` and get rid of the `Deferred` object altogether. You could indeed do this, but the previous listing has given us the chance to lead you gradually toward a crucial concept: when to use the `Deferred` object or the `Promise` object and why.

13.3.6 *Using the Promise object*

In order to master `Deferred`s and `Promise`s you have to understand when to use one and when the other. To help you understand this topic, let's say that you want to implement a promise-based timeout function. You are the *producer* of the function. In

this case, the *consumer* of your function doesn't need to care about resolving or rejecting it. The consumer only needs to be able to add handlers to execute operations upon the fulfillment, the failure, or the progress of the Deferred. Even more, you want to ensure that the consumer isn't able to resolve or reject the Deferred at their discretion.

To achieve this goal, you need to return the Promise object of the Deferred you've created in your timeout function, not the Deferred itself. To be able to do that, we need to introduce you to another method called deferred.promise().

Method syntax: deferred.promise

deferred.promise([target])
Return the Deferred's Promise object.

Parameters

target   (Object) An object to which the promise methods will be attached. If this parameter is provided, the method attaches the Promise behavior to it and returns this object, instead of creating a new one.

Returns
The Promise object.

Now that you know about the existence of this method, let's write some code to use it. The code is shown in the next listing, available as a JS Bin (http://jsbin.com/kefaza/edit?js,output) and in the file chapter-13/promise.timer.html.

Listing 13.5   A promise-based timer

```
function timeout(milliseconds) {
   var deferred = $.Deferred();                          ①  Creates a new Deferred
   setTimeout(deferred.resolve, milliseconds);
                                                             Resolves the
   return deferred.promise();                              Deferred after
}                                                           a given amount
                                                            of time
timeout(1000).done(function() {                          ② (milliseconds)
   alert('I waited for 1 second!');
});
```

① Creates a new Deferred
② Resolves the Deferred after a given amount of time (milliseconds)
③ Returns the Deferred's Promise
④ Adds a success callback to the returned Promise

In this listing you define a function called timeout() that wraps the JavaScript setTimeout() function. Inside timeout() you create a new Deferred object (you're the producer) ① and arrange that setTimeout() resolves this Deferred after a given number of milliseconds ②. Once finished, you return the Deferred's Promise object ③. Doing so, you ensure that the caller of your function (the consumer) ④ can't resolve or reject the Deferred object but can only add callbacks to execute using methods like deferred.done() and deferred.fail().

The previous example is pretty simple and may not have entirely clarified when to use a `Deferred` and when to use a `Promise` object. For those of you who still have doubts, let's discuss another example. We'll revisit our progress bar demo so that the `animate()` function will focus on animating the progress bar only, freeing it from the responsibility of updating the text showing the percentage of completion. Hence, it's the caller of the function that has to update the text percentage and, optionally, perform other tasks when the `Deferred` is resolved or rejected.

The new version of this code is shown in the following listing and is also available as a JS Bin (http://jsbin.com/cefece/edit?html,css,js,output) and in the file chapter-13/deferred.progress.2.html. Note that in this listing we've omitted the style of the page so that you can focus on the code. In addition, we've bolded the parts that are changed in comparison with the previous version.

Listing 13.6 Using `deferred.progress()`, version 2

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Using deferred.progress() version 2</title>
      <link rel="stylesheet" href="../css/main.css" />
   </head>
   <body>
      <div class="bar">
         <div class="inner-bar"></div>
         <span class="progress">0%</span>
      </div>

      <button id="run-button">Run animation</button>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         function animate(milliseconds) {
            var $innerBar = $('.inner-bar').width(0);
            var $barWidth = $('.bar').width();
            var step = $barWidth / 100;

            var deferred = $.Deferred();                    ← ❶ Creates a new
            var idInterval = setInterval(                        Deferred
               function () {
                  // Can't use width() to retrieve the width because
                  // of issue #1724
                  // (https://github.com/jquery/jquery/issues/1724)
                  var nextWidth =
                     parseFloat($innerBar.get(0).style.width) + step;
                  deferred.notify(nextWidth * 100 / $barWidth);
                  $innerBar.width(nextWidth);
                  if (nextWidth >= $barWidth) {
                     deferred.resolve();
                     clearInterval(idInterval);
                  }
```

```
                },
                milliseconds / 100
            );

            return deferred.promise();                ❷ Returns the
        }                                                Deferred's Promise

        $('#run-button').click(function() {
            animate(1000)
                .progress(function(value) {
                    $('.progress').text(Math.floor(value) + '%');
                })
                .done(function() {
                    alert('The process is completed');
                });
        });
    </script>
</body>
</html>
```

Adds a handler to execute while the Promise object returned is in progress ❸

Adds a handler to execute when the Promise object returned is resolved ❹

As you can see, in this version of the `animate()` function you only create the `Deferred` ❶. Once the code that runs the animation has been set, you return its `Promise` so that the caller of this method can add callbacks ❷. Once again, you're returning the `Promise` because you want to enable the caller to add callbacks but not to be able to change the state of the `Deferred`. In this case, it's the responsibility of the function that created the `Deferred` to either resolve or reject it.

Finally, inside the handler that's attached to the `click` event of the button you define a callback to run during the progress of the animation using `deferred.progress()` ❸, and then a callback to execute when the `Deferred` is resolved, using `deferred.done()` (this was absent in the previous version) ❹.

With this example we've hopefully reinforced the concepts and the methods described so far. The most observant of you may have noted that we haven't yet covered the only method mentioned by both the Promises/A and the Promises/A+ specifications, the `then()` method. Let's fix that.

13.3.7 *Take it short with then()*

In section 13.1 we summarized the definitions of promise taken by the Promises/A and the Promises/A+ proposals. Both of these definitions mention a `then()` method, but whereas the Promises/A proposal specifies that the method must possess a third argument that will be used as a handler for a progress event, the Promises/A+ proposal doesn't have such an argument. The jQuery implementation of this method prior to version 3 (branches 1.x and 2.x) differs from both these proposals because it defines the first argument, the success callback, as mandatory and the other two, the failure and progress callbacks, as optional. In contrast, the Promises/A and Promises/A+ proposals specify that all their arguments are optional.

The `deferred.then()` method can be used as a shortcut for calling `deferred.done()`, `deferred.fail()`, and `deferred.progress()` when you only need

to execute one handler for each or some of these methods. In case you need more handlers of the same type, you can chain calls to `then()`, `done()`, `fail()`, and `progress()`. Moreover, if you don't need a handler of a given type, you can simply pass `null`. For example, you can write a code statement like the following:

```
$.Deferred()
   .then(success1)
   .then(success2, null, progress1)
   .done(success3);
```

Now that you know what this method can do for you, it's time to learn its syntax.

Method syntax: deferred.then

deferred.then(resolvedCallback[, rejectedCallback [, progressCallback]])

Defines handlers executed when the `Deferred` object is resolved, rejected, or in progress. In case one of these parameters isn't needed, you can pass `null`.

Parameters

resolvedCallback   (Function) A function executed when the `Deferred` is resolved.
rejectedCallback   (Function) A function executed when the `Deferred` is rejected.
progressCallback   (Function) A function executed when the `Deferred` is in progress.

Returns

A `Promise` object.

The `deferred.then()` method returns a `Promise` object instead of a `Deferred`. After you invoke it, you won't be able to resolve or reject the `Deferred` you used unless you keep a reference to it. Recalling the handler of the `click` event of the button defined in listing 13.6, using the `deferred.then()` method you could rewrite it as follows, obtaining the same result:

```
animate(1000).then(
   function() {
      alert('The process is completed');
   },
   null,
   function (value) {
      $('.progress').text(Math.floor(value) + '%');
   }
);
```

What makes this method even more interesting is that it has the power to forward the value received as a parameter to other `deferred.then()`, `deferred.done()`, `deferred.fail()`, or `deferred.progress()` calls defined after it.

Before you start to cry in despair, let's look at a simple example:

```
var deferred = $.Deferred();
deferred
   .then(function(value) { return value + 1; })
   .then(function(value) { return value + 2; })
```

```
   .done(function(value) { alert(value); });
deferred.resolve(0);
```

This code creates a new `Deferred` and then adds three functions to execute when it's resolved, two using the `deferred.then()` method and one using `deferred.done()`. The last line resolves the `Deferred` with a value of `0` (zero), causing the execution of the three functions defined (in the order in which they were added).

Inside the functions added using `deferred.then()` you return a new value, created starting from the one received. The first function receives the value `0` because this is the value passed to `deferred.resolve()`, sums it to 1, and returns the result. The result of the sum is passed to the next function added using `deferred .then()`. The second function receives `1` instead of `0` as an argument. This value (`1`) is summed to 2 and the result returned (3) is used by the next handler. This time, the handler is added using `deferred.done()`, which doesn't have this power, so you alert the final value, `3`.

If you add yet another function using `deferred.done()` to the chain in the previous example and return a modified value from the third in the chain (the one added using `deferred.done()`), the new handler will receive the same value as the third. The following code will alert the value `3` twice:

```
var deferred = $.Deferred();
deferred
   .then(function(value) { return value + 1; })
   .then(function(value) { return value + 2; })
   .done(function(value) { alert(value); return value + 5; })
   .done(function(value) { alert(value); });
deferred.resolve(0);
```

In Promises/A and Promises/A+ compliant libraries (for example, jQuery 3.x), a thrown exception is translated into a rejection and the failure callback is called with the exception. In jQuery 1.x and 2.x an uncaught exception will halt the program's execution. Consider the following code:

```
var deferred = $.Deferred()
deferred
   .then(function(value) {
      throw new Error('An error message');
   })
   .fail(function(value) {
      alert('Error');
   });
deferred.resolve();
```

In jQuery 3.x, you'll see an alert with the message "Error." In contrast, jQuery 1.x and 2.x will allow the thrown exception to bubble up, usually reaching `window.onerror`. If a function for this isn't defined, you'll see on the console the message "Uncaught Error: An error message."

You can investigate this issue further to better understand this different behavior. Take a look at the following code:

```
var deferred = $.Deferred();
deferred
  .then(function() {
     throw new Error('An error message');
  })
  .then(
     function() {
        console.log('First success function');
     },
     function() {
        console.log('First failure function');
     }
  )
  .then(
     function() {
        console.log('Second success function');
     },
     function() {
        console.log('Second failure function');
     }
  );
deferred.resolve();
```

In jQuery 3.x, this code would write on the console the messages "First failure function" and "Second success function." The reason is that, as we mentioned, the specification states that a thrown exception should be translated into a rejection and the failure callback has to be called with the exception. In addition, once the exception has been managed (in our example by the failure callback passed to the second `then()`), the following success functions should be executed (in this case the success callback passed to the third `then()`).

In jQuery 1.x and 2.x, none but the first function (the one throwing the error) is executed. You'll only see on the console the message "Uncaught Error: An error message."

jQuery 3: Method added

jQuery 3 adds a new method to the `Deferred` and the `Promise` objects called `catch()`. It's a method to define a handler executed when the `Deferred` object is rejected or its `Promise` object is in a rejected state. Its signature is as follows:

```
deferred.catch(rejectedCallback)
```

This method is nothing but a shortcut for `then(null, rejectedCallback)` and it has been added to align jQuery 3 even more to the ECMAScript 2015 specifications that include a namesake method.

Despite these differences of the jQuery library from the specifications, `Deferred` remains an incredibly powerful tool to have under your belt. As a professional developer and with the increasing difficulty of your projects, you'll find yourself using it a lot.

Sometimes, regardless of the state of a `Deferred`, you'll want to perform one or more actions. jQuery has a method for such circumstances, too.

13.3.8 *Always execute a handler*

There may be times when you want to execute one or more handlers regardless of the state of the `Deferred`. To do that you can use the `deferred.always()` method.

Method syntax: deferred.always

`deferred.always(callbacks[, callbacks, ..., callbacks])`

Add handlers that are called when the `Deferred` object is either resolved or rejected. This method accepts an arbitrary number of callbacks with a minimum of one.

Parameters

callbacks	(Function|Array) A function or array of functions that is called when the `Deferred` is either resolved or rejected.

Returns

The `Deferred` object.

Consider the following code:

```
var deferred = $.Deferred();
deferred
   .then(
      function(value) {
         console.log('success: ' + value);
      },
      function(value) {
         console.log('fail: ' + value);
      }
   )
   .always(function() {
      console.log('I am always logged');
   });
deferred.reject('An error');
```

When executing this code, you'll see two messages on the console. The first is "fail: An error" because the `Deferred` has been rejected. The second is "I am always logged" because the callbacks added using `deferred.always()` are executed regardless of the resolution or rejection of the `Deferred`.

There's one last method to discuss.

13.3.9 *Determine the state of a Deferred*

When writing code that uses a `Deferred`, you may need to verify its current state. To do that, you can employ the `deferred.state()` method. It does exactly what you'd expect: it returns a string that specifies the current state of the `Deferred`. Its syntax is the following.

Method syntax: deferred.state

`deferred.state()`

Determines the current state of a `Deferred` object. Returns a string that can be `"resolved"`, `"rejected"`, or `"pending"`.

Returns

A string representing the state of the `Deferred`.

This method is particularly useful when you want to unit-test your code. For example, you could write a statement like this:

```
assert.equal(deferred.state(), 'resolved');
```

The method used in the previous statement comes from the QUnit unit-testing framework that we'll discuss in the next chapter. It simply verifies that the first parameter is equal to the second, in which case the test passes.

13.4 *Promisifying all the things*

The previous sections focused on the `Deferred` and `Promise` objects and their methods, but we've reserved another small surprise for you: the `promise()` method. The latter is different from the `deferred.promise()` method that you learned a few pages back. `promise()` allows you to transform a jQuery object into a `Promise` object, enabling you to add handlers using the methods discussed in this chapter. The syntax of this method is shown here.

Method syntax: promise

`promise([type][, target])`

Returns a dynamically generated `Promise` object that's resolved once all actions of a certain type bound to the collection, queued or not, have finished. By default, type is `fx`, which means the returned `Promise` is resolved when all animations of the selected elements have completed.

Parameters

type	(String) The type of queue that has to be observed. The default value is `fx`, which represents the default queue for the effects.
target	(Object) The object onto which the `promise` methods have to be attached.

Returns

A `Promise` object.

Based on the description of this method, if the jQuery object in use doesn't have animations running, it's resolved right away. Hence, any handler attached is executed immediately. Consider the following code:

```
$('p')
   .promise()
   .then(function(value) { console.log(value); });
```

If the paragraphs of the page aren't running any animations, the function added using then() is executed immediately and the value passed to it is the jQuery object itself.

Let's now consider a slightly more complex example that involves animations created using the animate() method that you learned about in chapter 8. In this example you'll create two squares and move them from the left to the right of the page at different speeds, so that the animations will finish at different times. Once both the animations are completed, you'll alert a success message.

The code to achieve this goal is shown in the next listing and is also available as a JS Bin (http://jsbin.com/wuhiqa/edit?js,output) and in the file chapter-13/ promisify.html.

Listing 13.7   Promisifying a jQuery collection

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Promisifying a jQuery object</title>
        <link rel="stylesheet" href="../css/main.css" />
        <style>
            .square
            {
                position: relative;
                width: 25px;
                height: 25px;
                background-color: #1E39BC;
                margin: 10px 0;
            }
        </style>
    </head>
    <body>
        <div id="square1" class="square"></div>
        <div id="square2" class="square"></div>

        <script src="../js/jquery-1.11.3.min.js"></script>
        <script>
            $('#square1').animate({left: 500}, 1500);
            $('#square2').animate({left: 500}, 3000);
            $('.square')
                    .promise()
                    .done(function() {
                        alert('The animations are completed');
                    });
        </script>
    </body>
</html>
```

Animates the first square ❶

Animates the second square ❷

❸ Creates a Promise based on the jQuery object that includes both squares

❹ Defines a function to execute once both animations are completed

In the code of this listing you animate both squares, but the first animation will last 1500 milliseconds ❶, whereas the second will last 3000 milliseconds ❷. After these animations have been defined, you select both squares using the class name that they

share, `square`, and create a `Promise` object using the `promise()` method that we're covering in this section ❸. Finally, you add a function to execute once both animations are completed ❹. How cool is that?

13.5 *Summary*

In this chapter we introduced you to promises, a better pattern for dealing with asynchronous code. Promises enable you to avoid having to use nasty tricks to synchronize parallel asynchronous functions and the need to nest callbacks inside callbacks inside callbacks.

We outlined the Promises/A and the Promises/A+ proposals and explained how they compare with jQuery's implementation of promises. We also illustrated the differences that the two proposals have, focusing our attention on the different terminology adopted and the number of parameters specified for the `then()` method, the core of promises.

The jQuery implementation of promises involves two objects: `Deferred` and `Promise`. As we described, the latter is created starting from a `Deferred` or a jQuery object and possesses a subset of the methods of the `Deferred`.

We delved into the many methods exposed by `Deferred`. The latter can be resolved using the `deferred.resolve()` method or rejected using `deferred.reject()`. You can also be notified about the progress of the asynchronous operation using `deferred.notify()`. All these methods have a related method that accepts an additional parameter, which allows you to set the context of the functions triggered: `deferred.resolveWith()`, `deferred.rejectWith()`, and `deferred.notifyWith()`.

`deferred.done()`, `deferred.fail()`, and `deferred.progress()` are the methods that enable you to add handlers to run when a `Deferred` is resolved, rejected, or still in progress.

Another interesting concept is `$.when()`, a utility function that allows you to easily synchronize multiple asynchronous and even synchronous functions.

We introduced you to jQuery's implementation of the `then()` method, highlighting the differences that jQuery has with other libraries that adhere to either the Promises/A or the Promises/A+ proposal.

This chapter also covered other methods exposed by the `Deferred` object, like `deferred.always()` and `deferred.state()`.

Finally, we discussed `promise()`, a method to transform any jQuery object into a `Promise` object. This method is very useful because it enables you to use the power of `Deferred`s with simple jQuery objects.

With this chapter we've completed our overview of the jQuery library. Thanks to the chapters you've read so far, you've hopefully learned the ins and the outs of this library and you can now define yourself as a jQuery ninja! Congratulations!

In the next chapter we'll try to move a step forward by discussing a crucial concept that every professional developer has to master: unit testing. Turn the page and continue your learning process.

Unit testing with QUnit

This chapter covers

- Why testing is important
- What unit testing means
- Installing QUnit
- Testing your JavaScript code with QUnit
- How to create a complete test suite with QUnit

The previous chapter was the last discussing concepts strictly related to the jQuery core. `Deferred` and `Promise` might have been tough arguments to learn but, hopefully, with our help the process has gone smoothly.

It's now time for you to enhance your skills even further. In this chapter you'll learn other tools and techniques that every pro must know. You'll apply them to code written using jQuery, but you can reuse this knowledge with any code written in JavaScript. Testing is an essential concept to master if you're working in a team or on anything that isn't for your personal use only.

Important topics that you'll explore in this chapter are tests and how to make assertions. An assertion is a way to verify that your code works correctly and your assumptions are respected. Stated another way, an assertion verifies that a function returns the value you expect given a set of inputs, or that the value of a variable or property of an object is what you expect after running some operations on it.

Finally, after a brief introduction about what unit testing is and why you should consider it, we'll discuss QUnit. It's one of the libraries available in the JavaScript community to unit test code written in JavaScript. The reason why we'll talk about QUnit and not another library is that QUnit is the framework employed by all the jQuery projects (jQuery, jQuery UI, and jQuery Mobile) to unit test the code.

Without further ado, let's discover more about testing, in particular unit testing, and QUnit.

14.1 *Why is testing important?*

To explain why software testing is important, we'll start with an example. How would you answer if we asked you to drive a car that nobody has ever tested in a Formula 1 race? Would you risk your life with something nobody has ever checked to see if it's robust enough to not get destroyed at the first curve? Or that the brakes actually brake the car? Of course, the answer to all these questions is a big "No!" The same principle applies to software. Nobody wants to use software that crashes every two minutes or that doesn't work as expected, or even worse, that doesn't match the requirements or the expectations of the user. That's exactly where software testing comes into play.

Software testing is the process of evaluating a piece of software to detect differences between the actual and expected output, with a given input. Testing helps build safe, reliable, and stable software. It provides valuable information and insight into the state of the system or software. Software testing also serves as a metric of how your product differs from the specifications, customer expectations, past versions of the same product, and many other criteria. Another primary purpose is to detect defects of the code, usually referred to as bugs.

When talking about bug detection, we don't mean that testing allows you to verify all the possible conditions and find every bug in your system, which is impossible in every real-world system. The scope is to verify your code, product, or system under certain conditions to see if it works as expected. Even employing testing, bugs will always exist. This isn't something you can eradicate with practice or knowledge because it's intrinsic in every complex system. You have to accept that bugs in software exist not because you or other developers working on it aren't good enough, but because any real-world software is so complex that you can't predict, and thus fix, every source of failure.

Testing is an essential part of the life of every developer. At least, it should be. Unfortunately, many developers are scared of testing. It's often seen as an extra activity, something that forces you to waste time—a lot of time. Of course, you don't have to take this wisdom to the other extreme. If you're developing a really small piece of code for yourself to automate a process you only have to perform once in your life, testing it is probably not worth the effort. But as more experienced developers will confirm, if you're developing even a small project or library that you plan to use in your daily work and share with your team, other developers, or the entire JavaScript community, then you'd better test it.

Testing is an incredibly wide discipline. You can test your projects for a bunch of different aspects (for example, compatibility testing and regression testing), using different methods (for example, visual testing and black-box testing), and at different levels (for example, unit testing and integration testing). By no means is the purpose of this section to teach you the ins and outs of software testing. The topic is so broad that it would require us to write another book just for this topic. What we want to communicate here is the importance of testing and why you should test your software in case you haven't been doing it yet.

In the next section, we'll give you an overview of one of the types we mentioned: unit testing.

14.1.1 Why unit testing?

Unit testing is a software testing method that promotes the practice of thinking of software as a set of isolated pieces of code, referred to as *units*, that can be tested individually to verify that they work as expected. When unit testing, each set of tests targeting a single unit should be independent from the others. Usually a unit is identified by a function or a method, depending on the type of programming language adopted.

In short, the main benefits of unit testing code are these:

- Attest that the code returns expected results given specific inputs.
- Discover the highest number of defects in an early stage (related to the previous point).
- Improve the design of the code.
- Identify units that are too complex.

Adhering to the principles of unit testing, given a function of your software and a set of inputs, you can determine if the function is returning the expected outputs. This process is usually automated and involves a unit-testing framework. Unit testing consists of writing functions that, when executed, pass a set of inputs you've defined to the targeted function (the one you're testing). Then these functions verify that for each set of inputs the returned result is the one expected (that you've defined in the test function). Employing this method, you can also verify that if you pass invalid inputs to the targeted function, the latter is able to deal with them gracefully (this can also mean raising an expected error or exception). When one or more tests fail, you know there's an error in the code of the unit, so you need to fix it. This process is iterated until all the tests are passed (the results returned match those expected).

The goal of unit testing is to find the largest percentage possible of software defects in an early stage of the development process. Another advantage is that once you have all the tests for a given unit in place, you can improve the code of the unit (function or method) confidently. If you make a mistake while updating the code, one or more tests will (usually) fail and you'll know that something went wrong. Therefore, you can change your code more confidently knowing that you're not breaking a feature that used to work properly.

Another benefit always associated with unit testing is that it helps in understanding and improving how to design the code. Instead of writing code to do something, you start by outlining all the conditions your code has to meet and what outputs you expect. This concept is usually related to a development methodology called test-driven development (TDD).

Test-driven development

Test-driven development is a software development process that relies on writing tests before writing the code (unit) to test. The first step in TDD is to write an initially failing test case. Then the developer has to produce the code that implements the feature until all the tests are passed. Finally, the code is refactored until it matches an acceptable quality standard.

The final advantage we want to highlight is that unit testing helps in identifying units that are too complex. For example, you can recognize that a method is doing more than its primary goal—remember the single responsibility principle (SRP). What usually happens is that when writing the test code you feel that it's too complex or it's becoming too long. This is a good indication that the method needs to be divided up or refactored.

If a project is developed with JavaScript, there's an additional reason to embrace testing: browser incompatibilities. Although the major browsers, including Internet Explorer, are adhering to web standards more and more every day, they still have different behaviors in many circumstances. Having solid tests is one way to avoid the issue of deploying code that works in certain browsers and breaks your web pages in others. You can run the same tests in different browsers to verify that all of them pass.

Now that you have an idea of what testing and unit testing are, and you have a good grasp of why you should employ them, it's time to take a look at the unit-testing frameworks available for your JavaScript code.

14.1.2 *Frameworks for unit testing JavaScript*

Do you know that joke, pretty famous among JavaScript developers, that says that you should think of a word, search Google for "<word>.js," and if a library with that name exists, have a drink? If you didn't, you know it now. The point of this joke isn't to get you drunk but to highlight the huge number of JavaScript libraries, frameworks, and plugins out there. The same point could be made for unit testing frameworks.

The JavaScript community offers a lot of frameworks that you can use to unit test your projects. But like software, testing frameworks come and go. Before sticking with one you should check that it's still maintained. In this section we'll give you a brief overview of some of the most popular JavaScript unit-testing frameworks.

QUnit (http://qunitjs.com/) is the first unit-testing framework we want to introduce. It was developed to test jQuery, but then it turned into a standalone unit-testing

framework. It has been adopted by all the other projects managed by the jQuery team, but it can be used with any JavaScript-based code. QUnit supports the same browsers as jQuery 1.x. One of the advantages of this framework is that it provides an easy-to-use set of methods that you can employ to test your project. In addition to the usual assert methods, QUnit allows you to test asynchronous functions.

Mocha (http://mochajs.org/) is a feature-rich JavaScript test framework running on Node.js and the browser. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases.

Jasmine (http://jasmine.github.io/) is an open source, behavior-driven development (BDD) framework for JavaScript. It has a clean and easy-to-read syntax.

Behavior-driven development

Behavior-driven development is a software development process that evolved from test-driven development. When using BDD you not only test the code at the granular level with unit tests but also test the application using acceptance tests. BDD specifies that tests of any unit of software should be specified in terms of the desired behavior of the unit.

Other frameworks that you may read about on the web and that we want to mention are YUI Test (http://yuilibrary.com/yui/docs/test/) and Selenium (http://docs .seleniumhq.org/). Which framework to use depends on your preference, on your and your team's skills, and on the kind of approach you want to adopt (TDD or BDD). In the remainder of this chapter we'll discuss QUnit because, as we mentioned, it's the framework maintained and used by the jQuery team. If they trust QUnit, why shouldn't you? Let's look at how you can start using it.

14.2   *Getting started with QUnit*

One of the best features of QUnit is its ease of use. Getting started with this framework is a matter of performing three simple steps.

The first thing to do is to download the framework. QUnit can be downloaded in several different ways. The first method is to access its website and download the JavaScript and the CSS file in the latest version available.

> **NOTE**   At the time of this writing the latest version is 1.18.0, but all the examples provided in this chapter were developed to work seamlessly in QUnit 2.0.

The JavaScript file contains the test runner (the code responsible for executing the tests) and the actual testing framework (the set of methods used to test the code); the CSS file styles the test suite page used to display the test results. You can find the links to these files on the right-hand side of the homepage, as shown in figure 14.1.

The second step is to move them into a folder where you'll also create an HTML file. This file must contain a reference to the CSS file and the JavaScript file, as well as

Figure 14.1   The homepage of the QUnit framework

a mandatory element (usually a `<div>`) having an ID of `qunit`. Inside it, the framework will create the elements that make up the user interface used to group the tests and show the results. The resulting HTML code should resemble that shown in the following listing.

Listing 14.1   A minimal setup of the QUnit framework

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>QUnit Tests</title>
      <link rel="stylesheet" href="qunit-1.18.0.css" />        Includes the
   </head>                                                      framework
   <body>                                                       style sheet
      <div id="qunit"></div>               Wraps QUnit's
      <script src="qunit-1.18.0.js"></script>      user interface
   </body>                                                   Includes QUnit's
</html>                                                       JavaScript file
```

The last step to perform is to open the HTML file in the browser of your choice. Once it's open you'll be presented with a page similar to the one illustrated in figure 14.2.
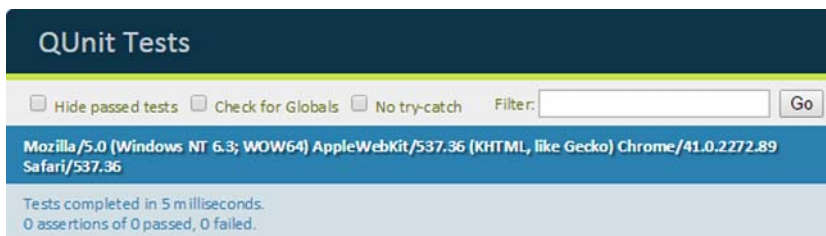
Figure 14.2   The user interface of the QUnit framework

Pretty easy, isn't it? Let's now analyze the components of this user interface.

The top part of the interface is nothing but the content of the `title` element placed in the `head` of the page ("QUnit Tests"). Below this title, you can see a green bar. It's green because all the tests passed (no tests at all means everything is working fine). This bar will turn red if one or more tests you defined fail.

In the second section you can see three check boxes: Hide passed tests, Check for Globals (sometimes referred as noglobals), and No try-catch. When the first option is checked, QUnit will hide the passed tests (those returning the expected result). The second check box allows you to verify if a property has been added to the `window` object by comparing it before and after each test, in which case the test will fail, listing the differences. The last flag can be used to verify if your code throws an exception when you instruct QUnit to run your test outside of its surrounding `try-catch` block. In addition to the check boxes, there's an `input` element that you can use to filter the tests run to search a specific test.

In the third section you can read the value of the `window.navigator.userAgent` property. This property returns the user agent string for the browser that's accessing the page.

The bottom section shows the time QUnit spent processing the defined tests. Underneath that you can read the number of *assertions* defined, the number of those that passed, and the number that failed.

> ### What's an assertion?
> An assertion verifies that a statement is equal to `true`. This is useful for testing that your code returns the expected result. It's important to note that an assertion should only be used to test *meaningful* code. We mean that you should verify only your code base and nothing more. For example, if your code uses a native JavaScript function, it's completely useless to test it. You have to assume that a JavaScript function (for example, `getElementById()`) doesn't have issues, even if sometimes it does.

As you can see, these numbers are all zero. The reason is that we haven't defined any test yet. There are no assertions at all, either passed or failed. When you write some tests and assertions, QUnit will list here the assertions grouped by test.

If you looked at the official documentation while downloading the files, you may have noticed that it specifies another element as mandatory in the minimal setup of the framework. This is an element (usually a `<div>`) having an ID of `qunit-fixture`. Its aim is to prevent some tests failing or succeeding as a side effect of previously executed tests, such as deleting or inserting elements in the DOM. This element is useful because the framework resets the elements inside it after each test. Thus, it isn't mandatory, but you should include it in any real project.

Other ways to obtain QUnit

QUnit can be obtained in different ways. A first possible method is to include the files needed using the jQuery CDN. To do that using version 1.18.0, you should include in your page the following code:

```
<link rel="stylesheet" href="//code.jquery.com/qunit/qunit-1.18.0.css"/>
<script src="//code.jquery.com/qunit/qunit-1.18.0.js"></script>
```

Another method is to download QUnit via Bower by running the following command:

```
bower install --save-dev qunit
```

Finally, you can obtain the framework via npm:

```
npm install --save-dev qunitjs
```

With the framework set up, you're ready to test your code. But before learning how to do that, you need to make a few changes. The first is to include in your page the JavaScript code or file that you want to test (for example, code.js). You can place it wherever you want as long as it comes before the code you'll write to test it. The second change is to include the code or file containing the tests. They're usually placed in a different JavaScript file, often called tests.js. This file must be included after the `script` element you used to include the QUnit's JavaScript file. You should have code that looks like the following:

```
<script src="code.js"></script>
<script src="qunit-1.18.0.js"></script>
<script src="tests.js"></script>
```

The content of these files can also be inlined (putting the content inside the `script` element). In the examples of this chapter we'll always inline the test code and sometimes even the code to test because of its simplicity, but we strongly suggest adopting an external file when you employ QUnit in a real project.

With this last note in mind, you're ready to discover what this framework has to offer.

14.3   *Creating tests for synchronous code*

QUnit allows you to test synchronous and asynchronous code. For the moment we'll focus on testing synchronous code because it's the easiest way to delve into the world of QUnit.

To create a test in QUnit you have to use a method called `test()`. Its syntax is shown here.

Method syntax: QUnit.test

`QUnit.test(name, test)`

Add a test to run.

Parameters

`name`	(String) The name to identify the test created.
`test`	(Function) The function containing the assertions to run. The framework passes an argument called `assert` to this function. It provides all of QUnit's assertion methods.

Returns

`undefined`

To create a new test, write the following code:

```
QUnit.test('My first test', function(assert) {
   // Code here...
});
```

If you put this test in the previously mentioned tests.js file or inline it and then open the HTML file in a browser, you'll see an error. The framework is complaining that you've defined a test without any assertion. What's the sense of defining a test if you don't test any code at all?

When creating a test, it's a best practice to set the number of assertions you expect to be executed. This can be done by using the `expect()` method of the `assert` parameter described when discussing `QUnit.test()`. If you deal with synchronous code only, the use of `expect()` might seem useless because you might think that the only way an assertion could not be executed is if an error, caught by the framework, occurs. This objection is valid until you take into account asynchronous code. For the moment, trust us and use `expect()`.

The syntax of `expect()` is as follows.

Method syntax: expect

`expect(total)`

Set the number of assertions that are expected to run within a test. If the number of assertions actually executed doesn't match the `total` parameter, the test will fail.

Parameters

`total`	(Number) The number of assertion expected to run within the test.

Returns

`undefined`

With the knowledge of the `expect()` method in hand, you can now modify the previous test to set your expectation of running zero tests:

```
QUnit.test('My first test', function(assert) {
   assert.expect(0);
});
```

If you reopen the HTML page, you'll find that the previous error message has disappeared. The reason is that with this change you've explicitly set the number of assertions to be executed to zero, so QUnit is sure that this is exactly what you wanted.

By using `expect()` you've fixed the issue in your HTML page but you still don't have any assertions in place. Let's examine the various types of assertions QUnit provides.

14.4 *Testing your code using assertions*

Assertions are the core of software testing because they allow you to verify that your code is working as expected. QUnit provides numerous methods to test your expectations that can all be accessed within a test through the `assert` parameter passed to the function passed to `QUnit.test()`.

We'll start our overview of the assertion methods by covering four of them.

14.4.1 *equal(), strictEqual(), notEqual(), and notStrictEqual()*

In this section we'll cover four of the methods provided by QUnit. The first method we want to introduce is `equal()`.

Method syntax: equal

`equal(value, expected[, message])`
Verify that the `value` parameter is equal to the `expected` parameter using a nonstrict comparison (==).

Parameters

`value`	(Any) The value returned by a function or a method, or stored in a variable to verify.
`expected`	(Any) The value to test against.
`message`	(String) An optional description of the assertion. If omitted, the message shown is "okay" in case of success and "failed" in case of failure.

Returns
`undefined`

The description of the assertion, the `message` parameter, is optional but we suggest that you always set it.

To give you an idea of how to use this method, let's look at a simple example. Imagine you created a function that sums two numbers passed as arguments. The definition of such a function in JavaScript would be like the following:

```
function sum(a, b) {
   return a + b;
}
```

With the `sum()` function in place, you want to verify that it works correctly. To do that employing the `equal()` method, you can write a test like this:

```
QUnit.test('My first test', function(assert) {
   assert.expect(3);
   assert.equal(sum(2, 2), 4, 'Sum of two positive numbers');
   assert.equal(sum(-2, -2), -4, 'Sum of two negative numbers');
   assert.equal(sum(2, 0), 2, 'Sum of a positive number and the neutral
      element');
});
```

You can run the test opening the file chapter-14/test.1.html or accessing the relative JS Bin (http://jsbin.com/towoxa/edit?html,output).

Running the previous test gives you confidence that your code works properly and your function is robust. But is this the case? As it turns out, it isn't. What if you add the following assertion to your test? (Remember to update the argument passed to `assert.expect()` accordingly.)

```
assert.equal(sum(-1, true), 0, 'Sum of a negative number and true');
```

This assertion succeeds because JavaScript is a weakly typed language, so `true` is equal to `1`. Therefore, summing `-1` and `true` gives `0` as a result.

In addition to the issue of passing a Boolean, what will happen if you pass a number and a string like `"foo"` to `sum()`? In this situation your function will treat both parameters as strings and will concatenate them. Sometimes this may even be the expected result, but usually you want to explicitly deal with such cases. For example, you may want to raise an exception if one or both of the parameters aren't of type `Number` or convert them before performing the sum. Before we start discussing how to deal with exceptions and taking into account complex cases, let's introduce another assertion method QUnit has to offer: `strictEqual()`.

The `strictEqual()` method is similar to `equal()` with the exception that it performs a strict comparison between the actual and the expected values. Its syntax is shown here.

Method syntax: strictEqual

`strictEqual(value, expected[, message])`

Verify the `value` parameter is equal to the `expected` parameter using a strict comparison (===).

Parameters

`value`	(Any) The value returned by a function, a method, or stored in a variable to verify.
`expected`	(Any) The value to test against.
`message`	(String) An optional description of the assertion. If omitted, the message shown is "okay" in case of success and "failed" in case of failure.

Returns

`undefined`

For the sake of using `strictEqual()` with the simple `sum()` function you created, let's replace the previous assertion with one where you compare the sum of two numbers

with the Boolean value `false` (you can also update all the other assertions to use `strictEqual()`):

```
assert.strictEqual(sum(-2, 2), false, 'Sum of a negative and a positive
    number is equal to false');
```

Refreshing the HTML page will present you with an error. The test is able to detect that the actual and expected values aren't equal. But using this kind of assertion (that isn't very useful to test the function), your test is failing, which isn't what you want. The idea should be to verify that the sum of two numbers is not equal to a Boolean (`false` in this case), and if this happens your assertion has to succeed.

For such situations where you want to assert that a value is not equal or strictly equal to another, you can employ the counterpart of the methods you've seen: `notEqual()` and `notStrictEqual()`.

Update the previous assertion in order to see if your test succeeds:

```
assert.notStrictEqual(sum(-2, 2), false, 'Sum of a negative and a positive
    number is not equal to false');
```

This time refreshing the HTML page will correctly execute all four assertions and your test will succeed.

The complete and updated version of the page is shown in the following listing. It's available in the file chapter-14/test.2.html of the source provided with this book and as a JS Bin (http://jsbin.com/suroya/edit?html,output).

Listing 14.2 The use of strictEqual() and notStrictEqual()

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>QUnit - Test 2</title>
      <link rel="stylesheet" href="../css/qunit-1.18.0.css" />        ⟵  The CSS
   </head>                                                                style sheet
   <body>                                                                 of QUnit
      <div id="qunit"></div>
      <div id="qunit-fixture"></div>
      <script>
         function sum(a, b) {              The
            return a + b;                  function
         }                                 to test
      </script>
      <script src="../js/qunit-1.18.0.js"></script>       ⟵           The
      <script>                                                        JavaScript file
         QUnit.test('My first test', function(assert) {    ⟵         of QUnit
            assert.expect(4);
            assert.strictEqual(                                       The code to test
               sum(2, 2),                                             the sum()
               4,                                                     function
               'Sum of two positive numbers'
            );
            assert.strictEqual(
```

```
            sum(-2, -2),
            -4,
            'Sum of two negative numbers'
        );
        assert.strictEqual(
            sum(2, 0),
            2,
            'Sum of a positive number and the neutral element'
        );
        assert.notStrictEqual(
            sum(-2, 2),
            false,
            'Sum of a negative and a positive number is not false'
        );
    });
    </script>
  </body>
</html>
```

Up to this point, we've only discussed how to test numbers in JavaScript. But the methods you've learned can be used to test other data types such as `Array`, `Object`, `String`, and so on. In addition, although QUnit is useful for testing any JavaScript code, this book is still about jQuery. Therefore, let's see some examples using all the methods described so far applied to code written using jQuery methods and utility functions:

```
assert.equal($.trim('        '), '',
    'Trimming a spaces-only string returns an empty string');
assert.strictEqual($('input:checked').length,
    $('input').filter(':checked').length,
    'Filtering elements in advance or later produce the same number of
     elements');
assert.notEqual($('input:checked'), $('input').filter(':checked'),
    'Two jQuery objects are different unless they point to the same memory
     address');
assert.notStrictEqual(new Array(1, 2, 3), [1, 2, 3],
    'Two arrays are different unless they point to the same memory address');
```

As you can see, testing different data types is really no different than testing `Numbers`. Let's now move on to the other assertion methods.

14.4.2 *The other assertion methods*

Other assertion methods provided by QUnit are similar in scope and the parameters they accept; therefore we decided to compact them into table 14.1.

Table 14.1   An overview of other assertion methods provided by QUnit

Method	Description
`deepEqual(value, expected[, message])`	A recursive, strict comparison that works on all JavaScript types. The assertion passes if `value` and `expected` are identical in terms of properties and values and `value` and `expected` also have the same prototype.

Table 14.1   An overview of other assertion methods provided by QUnit *(continued)*

Method	Description
`notDeepEqual(value, expected[, message])`	Same as `deepEqual()` but tests for inequality of at least one property or value.
`propEqual(value, expected[, message])`	A strict comparison of the properties and values of an object. The assertion passes if all the properties and the values are equal in a strict comparison.
`notPropEqual(value, expected[, message])`	Same as `propEqual()` but tests for inequality of at least one property or value.
`ok(value[, message])`	An assertion that passes if the first argument is truthy.

To see these new methods in action, let's build a function that tests if a number is even:

```
function isEven(number) {
   return number % 2 === 0;
}
```

To test the `isEven()` function you may write

```
assert.strictEqual(isEven(4), true, '4 is an even number');
```

But by using the `ok()` method you can simplify the assertion:

```
assert.ok(isEven(4), '4 is an even number');
```

Much better, isn't it?

The difference between `deepEqual()` and `propEqual()` is subtle but important. To understand it, let's define an object literal called `human` with a property named `full-Name` that's initialized to `null`. In addition, let's define a function called `Person` that has to be used as a constructor that accepts as its only parameter a string and defines a property on the created object named `fullName`. The code to create the function and the object literal is the following:

```
function Person(fullName) {
   this.fullName = fullName;
}

var human = {
   fullName: null
};
```

Now you can instantiate an object of type `Person` setting `"John Doe"` as the value of `fullName` and also set the `fullName` property of the `human` object literal. Then you can test for their equality using `deepEqual()` and `propEqual()` to highlight their difference. The relevant code is shown here, but it's also available in the file chapter-14/test.3.html and as a JS Bin (http://jsbin.com/tecihe/edit?html,output):

```
QUnit.test('Testing propEqual() and deepEqual()', function(assert) {
   assert.expect(2);

   var person = new Person('John Doe');
   human.fullName = 'John Doe';

   assert.propEqual(person, human, 'Passes. Same properties and values');
   assert.deepEqual(person, human, 'Fails. Same properties and values, but
      different prototype');
});
```

If you run this test, you'll see that the first assertion passes but the second fails. The reason is that the objects `person` and `human` have the same properties and values but different *prototypes* (`person` has `Person` as its prototype whereas `human` has `Object` as its prototype). This condition is sufficient to pass an assertion using `propEqual()` but not using `deepEqual()`.

There's one last assertion method to discuss.

14.4.3 *The throws() assertion method*

We left the `throws()` assertion method to the end because it's a bit different from the others. Its syntax is as follows.

Method syntax: throws

throws(function[, expected][, message])

Verify that a callback throws an exception, and optionally compare the thrown error.

Parameters

function	(Function) The function to execute.
expected	(Object|Function|RegExp) An Error object, an Error function (constructor), a RegExp that matches (or partially matches) the string representation, or a callback function that must return `true` to pass the assertion check.
message	(String) An optional description of the assertion. If omitted, the message shown is "okay" in case of success and "failed" in case of failure.

Returns

undefined

This method is different from the others in that it doesn't accept as its first argument a value to test but rather a function that's expected to throw an error because the expected value is optional. To see it in action, let's modify the `isEven()` function to throw an error if the parameter passed isn't of type `Number`:

```
function isEven(number) {
   if (typeof number !== 'number') {
      throw new Error('The passed argument is not a number');
   }

   return number % 2 === 0;
}
```

To test that the error is thrown, you can write

```
assert.throws(
   function() {
      isEven('test');
   },
   new Error('The passed argument is not a number'),
   'Passing a string throws an error'
);
```

In this case you pass the expected value in the form of the same `Error` instance you expect to be thrown. Alternatively, you can verify that the error message is what you expect by changing the previous assertion to use a regular expression:

```
assert.throws(
   function() {
      isEven('test');
   },
   /The argument passed is not a number/,
   'Passing a string throws an error'
);
```

With the `throws()` method we've completed the overview of the assertion methods provided by QUnit to test synchronous code. To test asynchronous code like callbacks passed to jQuery's Ajax functions, you need to learn an additional method. Let's discover more.

14.5 *How to test asynchronous tasks*

Sometimes you need to perform a given action or repeat it over and over again after a given amount of time. Other times you want to retrieve information from a server without reloading the page. These are situations where you need to execute one or more functions asynchronously.

To test asynchronous functions you can use the same method to create the test and the same assertion methods you learned in the previous section. But you also need a mechanism to inform the test runner that you're waiting for an asynchronous method to be completed. Let's look at the `async()` method. It belongs to the same `assert` parameter we've mentioned many times in this chapter, and its syntax is the following.

Method syntax: async

`async()`
Instruct QUnit to wait for an asynchronous operation.

Parameters

Returns

A unique resolution callback function.

As you can see from the description, this method doesn't accept any parameters and returns a function. This function is unique and must be used only once, and it must be invoked inside the asynchronous function you want to test.

To better understand how this methods works, let's analyze the code shown in the next listing, which is also available in the file chapter-14/asynchronous.test.html.

Listing 14.3 Asynchronous test with QUnit

```html
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>QUnit - Asynchronous test</title>
      <link rel="stylesheet" href="../css/qunit-1.18.0.css" />
   </head>
   <body>
      <div id="qunit"></div>
      <div id="qunit-fixture"></div>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         function isEven(number) {
            return number % 2 === 0;
         }
      </script>
      <script src="../js/qunit-1.18.0.js"></script>
      <script>
         QUnit.test('Testing asynchronous code', function(assert) {
            var $fixtures = $('#qunit-fixture');
            assert.expect(4);

            assert.strictEqual(
               $fixtures.children().length,
               0,
               'The children of qunit-fixtures are 0'
            );

            var firstCallback = assert.async();
            window.setTimeout(function() {
               assert.ok(isEven(4), '4 is even');
               firstCallback();
            }, 500);

            var secondCallback = assert.async();
            $fixtures.load('test.1.html #qunit', function() {
               assert.ok(
                  true,
                  'File test.1.html has been successfully loaded'
               );
               assert.strictEqual(
                  $fixtures.children().length,
                  1,
                  'The elements appended to qunit-fixtures are 1'
```

1 Defines an isEven() function

2 Retrieves the element with ID qunit-feature

3 Calls async() to create a unique callback stored in firstCallback

4 Calls window.setTimeout() to execute a function asynchronously

5 Executes the callback stored in firstCallback

6 Calls async() to create a second unique callback stored in secondCallback

7 Retrieves a page asynchronously using load()

```
                );
                secondCallback();
            });
        });
    </script>
  </body>
</html>
```

> **Executes the callback stored in ❽ secondCallback**

In this code you set up the markup to use QUnit and include jQuery. You also define the isEven() function to test if a number is even or not ❶. Then, after including the QUnit test runner, you create an asynchronous test using the usual QUnit.test() method.

Inside the test you retrieve the element with the ID of qunit-feature ❷ that you'll use to inject some elements later in the code. Then you set the number of asserts you expect to run and create a first assertion.

Next, you call the assert.async() method to create a first unique callback that's stored in the variable named firstCallback ❸. The first asynchronous operation you perform is to run a function with 500 milliseconds of delay by using the window.setTimeout() method ❹. Inside the callback defined, you test if 4 is even and execute the callback stored in firstCallback ❺. Executing this function is crucial because if you omit it, the test runner will wait indefinitely for the invocation of the function, blocking the execution of any other test.

In the last part of the code, you execute async() again to create a second unique callback ❻. The second callback is stored in the variable secondCallback. The other asynchronous operation of the code uses jQuery's load() method. You attempt to retrieve the file chapter-14/test.1.html and then inject only the element having qunit as its ID ❼. Inside the callback passed to load(), you verify that the file is correctly loaded using the assert.ok() method and that only one element has been injected into the element having as its ID qunit-fixture. Once this is finished, you execute the callback stored in secondCallback ❽.

Running asynchronous tests is easy, as you've seen in this example. The only aspect to remember is to invoke the callback functions generated using the async() method.

Sometimes when using jQuery's Ajax functions you want to load resources from a server. It may happen that you're developing the JavaScript code before the backend code has been written or that you don't want to rely on the correctness of the backend code to ensure that your frontend code works as expected. For such situations, you can mock the Ajax requests using jQuery Mockjax (https://github.com/jakerella/jquery-mockjax) or Sinon.js (http://sinonjs.org/).

Earlier in this chapter we mentioned three check boxes on the user interface provided by QUnit when running tests. Let's talk a bit more about the two of them that modify the behavior of QUnit.

14.6   *noglobals and notrycatch*

QUnit offers you two check boxes, `noglobals` (labelled as "Check for Globals") and `notrycatch` (labelled as "No try-catch"), that you can check and uncheck as needed in order to change the behavior of all tests performed on the page.

The `noglobals` flag will fail a test if a new global variable (which is the same as adding a property to the `window` object) is introduced by the code you're executing. The following example shows a test that will fail if the `noglobals` flag is checked:

```
QUnit.test('Testing the noglobals option', function(assert) {
   assert.expect(1);
   window.bookName = 'jQuery in Action';
   assert.strictEqual(bookName, 'jQuery in Action', 'Strings are equal');
});
```

The reason the test fails is because the code added the `bookName` property to the `window` object. The test wouldn't have failed if the code had only modified an existing property of the `window` object such as `name`.

The `notrycatch` flag allows you to run QUnit without a surrounding `try-catch` block. This will prevent QUnit from catching any error and listing it, but often it allows for deeper debugging of an issue. The following example, which you can find in the file chapter-14/notrycatch.html, shows this option in action:

```
QUnit.test('Testing the notrycatch option', function(assert) {
   assert.expect(1);
   assert.strictEqual(add(2, 2), 4, 'The sum of 2 plus 2 is equal to 4');
});
```

In this code the `add()` function isn't defined, so invoking it will cause an error (specifically a `ReferenceError`). If you run the code activating the `notrycatch` flag, the framework won't catch this error for you and it'll be logged in the console. The difference is shown in figures 14.3a and 14.3b.
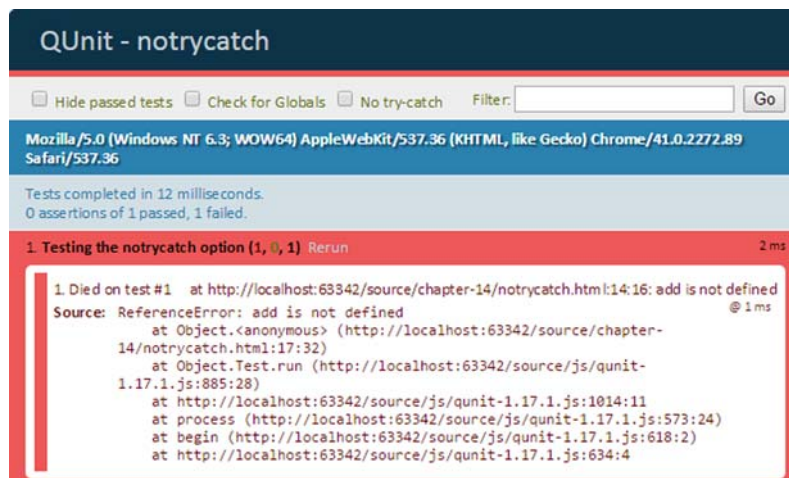
Figure 14.3a   Executing the test without the `notrycatch` flag
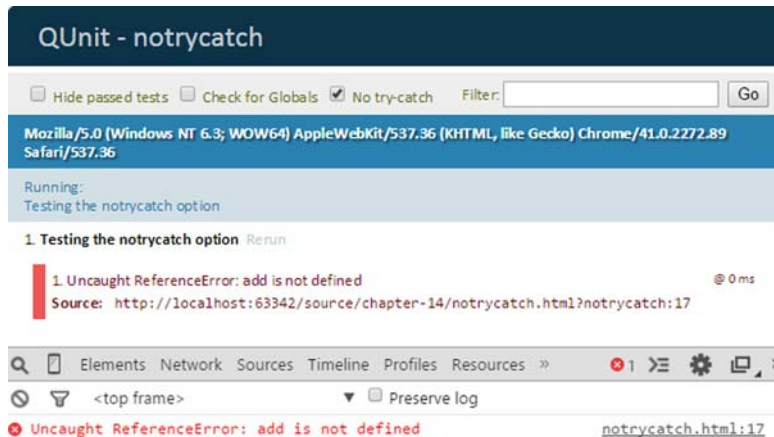
Figure 14.3b   Executing the test with the `notrycatch` flag activated

When dealing with large applications, you need to keep your tests logically organized so that you're able to run a specific group of tests without running the complete test suite. This is where modules come into play.

14.7   *Group your tests in modules*

As a plugin or library increases in size, you may want to split the source into modules to enhance its maintainability. You already saw this pattern when we discussed the structure of jQuery, whose code is made up of more than 10,000 lines. The same principle applies to the code you write to test your project.

  QUnit has a simple method to group tests into modules called `QUnit.module()`. Its syntax is as follows.

Method syntax: QUnit.module

`QUnit.module(name[, lifecycle])`
Group a set of related tests under a single module.

Parameters

  name   (String) The name to identify the module.
  lifecycle   (Object) An object containing two optional functions to run before (`beforeEach`)
                and after (`afterEach`) each test, respectively. Each of these functions receives
                an argument called `assert` that provides all of QUnit's assertion methods.

Returns

  `undefined`

Looking at the signature of this method, you may guess how you can specify which tests belong to a given module. The answer is that the tests belonging to a module are those defined after `QUnit.module()` is invoked but before another call to `QUnit.module()` (if any) is found. The following code should clarify this statement:

```
QUnit.module('Core');
QUnit.test('First test', function(assert) {
   assert.expect(1);
   assert.ok(true);
});


QUnit.module('Ajax');
QUnit.test('Second test', function(assert) {
   assert.expect(1);
   assert.ok(true);
});
```

In this snippet we've highlighted in boldface the invocations of `QUnit.module()`. In this case the test labeled as "First test" belongs to the Core module, and the test labeled "Second test" belongs to the Ajax module.

If you put the previous code into action, you'll see that the test names are preceded by the module name in the test results. In addition, you can select a specific module name to select the tests to run from the drop-down menu displayed in the top-right corner of the page. These details are illustrated in figure 14.4.
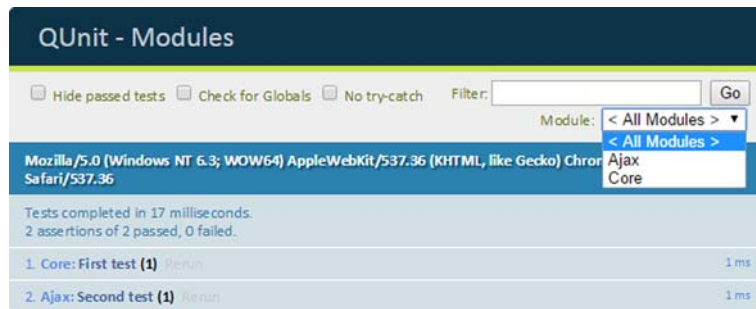
Figure 14.4   Execution of a test suite organized into modules

Now that you know how to organize a test suite in a proper and maintainable way, it's time to learn some specific configuration properties of the QUnit framework.

14.8   Configuring QUnit

In the same way that jQuery has a lot of reasonable default values for many of its methods, QUnit is released with a preset configuration. Sometimes you may need to tweak this configuration a bit to satisfy your project-specific needs.

The framework allows you to override these default values, exposing them through a property called `config`. Table 14.2 shows all the properties exposed via the `QUnit.config` object.

Table 14.2 The configuration properties of `QUnit.config`

Name	Description
altertitle	(Boolean) QUnit changes the `document.title` to add a check mark or an x to specify that a test suite passed or failed. Setting this property to `false` (the default value is `true`) disables this behavior. This is useful if your code works with `document.title`.
autostart	(Boolean) QUnit runs tests when the `load` event is triggered on `window`. If you load tests asynchronously, you can set the value to `false` (by default it's `true`) and then call `QUnit.start()` when everything is loaded.
hidepassed	(Boolean) QUnit shows all the tests, including the ones passed. Setting this property to `true`, you'll see only those that failed.
moduleFilter	(String) Specify a single module to run by specifying its name. The default value is `undefined`, so QUnit will run all the loaded modules.
reorder	(Boolean) The framework first runs tests that failed on a previous execution. If you want to change this behavior, set the value to `false`.
requireExpects	(Boolean) Set this property to `true` if you want to force the use of the `assert.expect()` method.
testId	(Array) This property allows QUnit to run specific test blocks by a hashed string combining their module name and test name. The default value is `undefined`.
testTimeout	(Number) Set a maximum time execution after which all tests will fail. The default value is `undefined`, which means there's no limit.
scrolltop	(Boolean) Set this property to `false` if you want to prevent QUnit from going back to the top of the page when all the tests have executed.
urlConfig	(Array) Controls the form controls to place into the QUnit toolbar (the bar where you can find the `noglobals` and `notrycatch` flags). By extending this array, you can add your own check boxes and select lists.

The custom configuration of the test suite must be placed after the JavaScript file of QUnit. You can define the configuration in an external file, as in the following code, or inline it.

```
<script src="qunit-1.18.0.js"></script>
<script src="qunit-config.js"></script>
```

If you have a huge test suite, you may want to hide the passed tests by default to focus on those that have failed. To do that you use the `hidepassed` property described in table 14.2 as shown here:

```
QUnit.config.hidepassed = true;
```

If you want to force yourself or your team to specify the number of assertions, you can use the `requireExpects` property:

```
QUnit.config.requireExpects = true;
```

QUnit defines other methods that we haven't and won't cover in this chapter, but they're of secondary importance. The topics we've covered up to this point allow you to create a complete test suite that's sufficient for most cases.

In the next section, you'll create a complete test suite to see all the QUnit knowledge you've acquired in action. It'll be fun!

14.9 An example test suite

In this last section you'll build a complete test suite for a project, and what's better than testing something you've developed in this very book? That's why you'll create a test suite for Jqia Context Menu, the jQuery plugin you built in chapter 12 to show a custom context menu on one or more specified elements of a page.

The first step needed to create the suite is to construct a new page with the essential setup for QUnit, where you'll also need to include the jQuery library and the files relative to Jqia Context Menu (jquery.jqia.contextMenu.css and jquery.jqia.contextMenu.js). Once you've done so, you'll place an unordered list that will act as the custom menu inside the element having `qunit-fixture` as its ID. You'll also use this element as the one where you'll show the custom menu.

You want to force yourself and all the people involved in writing the test suite to use the `assert.expect()` method. To do so you have to modify the default value of the QUnit configuration's property `requireExpects` so that QUnit raises an error if it isn't invoked.

The Jqia Context Menu plugin consists of one JavaScript file, so ideally you won't need to group your tests into modules. But you must remove all the data attached to the element having the ID of `qunit-fixture` each time a test is completed. Therefore, you need to use `afterEach` as follows:

```
QUnit.module('Core', {
   afterEach: function() {
      $('#qunit-fixture').removeData();
   }
});
```

After the declaration of the module you can define your tests. You'll create five tests in total, divided as described here:

- *Basic requirements*—Contains the assertions that verify whether jQuery and the Jqia Context Menu plugin have been correctly loaded. In addition it checks that your plugin correctly exposes the default values to configure it.
- *Wrong parameters*—Defines the assertions to check that the plugin is able to deal with unexpected parameter types or missing mandatory properties (that is, `idMenu`).
- *Initializations*—Specifies the assertions to verify if the plugin breaks chainability and that it doesn't raise errors when the right parameters are passed. In addition, it tests that the plugin can't be initialized two or more times on the same element.

- *Callbacks*—Contains the assertions to test if the menu is displayed or hidden based on the events fired and what element caused the event.
- *Destroy*—Tests that the chainability is kept and that the method correctly removes all the data attached to the element. In addition, it verifies that the menu is hidden when the effect of the plugin is cancelled.

Coding the description of these tests results in the following listing.

Listing 14.4   The complete test suite for Jqia Context Menu

```
QUnit.test('Basic requirements', function(assert) {
    assert.expect(4);

    assert.ok($, 'jQuery is loaded');
    assert.ok($.fn.jqiaContextMenu, 'The plugin is loaded correctly');
    assert.ok($.fn.jqiaContextMenu.defaults, 'The defaults are exposed');
    assert.propEqual(
        $.fn.jqiaContextMenu.defaults,
        {
            idMenu: null,
            bindLeftClick: false
        },
        'The defaults exposed are correct'
    );
});

QUnit.test('Wrong parameters', function(assert) {
    assert.expect(6);
    var $fixture = $('#qunit-fixture');

    assert.throws(
        function() {
            $fixture.jqiaContextMenu('no method');
        },
        /Method .*? does not exist/,
        'Call an undefined method'
    );

    assert.throws(
        function() {
            $fixture.jqiaContextMenu(100);
        },
        /Method .*? does not exist/,
        'Wrong argument type: number'
    );

    assert.throws(
        function() {
            $fixture.jqiaContextMenu(null);
        },
        /Method .*? does not exist/,
        'Wrong argument type: null'
    );

    assert.throws(
        function() {
```

Creates a test to verify basic requirements are met

Tests that the plugin can deal with wrong parameters passed to it

```
            $fixture.jqiaContextMenu([]);
        },
        /Method .*? does not exist/,
        'Wrong argument type: array'
    );

    assert.throws(
        function() {
            $fixture.jqiaContextMenu({});
        },
        /No menu specified/,
        'Unspecified menu'
    );

    assert.throws(
        function() {
            $fixture.jqiaContextMenu({idMenu: 'unknown id'});
        },
        /The menu specified does not exist/,
        'Unknown menu'
    );
});

QUnit.test('Initialization', function(assert) {
    assert.expect(5);
    var $fixture = $('#qunit-fixture');
    var $fixtureInitialized = $fixture.jqiaContextMenu({
        idMenu: 'context-menu'
    });

    assert.ok($fixtureInitialized, 'Menu initialized');
    assert.notEqual(
        $fixtureInitialized.data('jqiaContextMenu'),
        {},
        'Correct namespace used'
    );
    assert.strictEqual(
        $fixture.length,
        $fixtureInitialized.length,
        'Keep chainability'
    );
    assert.strictEqual(
        $fixture,
        $fixtureInitialized,
        'Return the same object'
    );
    assert.throws(
        function() {
            $fixture.jqiaContextMenu({idMenu: 'context-menu'});
        },
        /The plugin has already been initialized/,
        'Plugin already initialized on the element'
    );
});

QUnit.test('Callbacks', function(assert) {
    assert.expect(3);
```

Verifies that jqiaContextMenu() keeps chainability, sets data-* attributes, and so on when passing a correct parameter

Tests that the menu is hidden/shown after the event of interest is fired (clicks outside/inside the target elements)

```
      var $fixture = $('#qunit-fixture').jqiaContextMenu({
         idMenu: 'context-menu'
      });
      var $menu = $('#context-menu');

      assert.strictEqual(
         $menu.css('display'),
         'none',
         'The menu is hidden'
      );
      $fixture.trigger('contextmenu');
      assert.strictEqual(
         $menu.css('display'),
         'block',
         'The menu is displayed after the click'
      );
      $('html').click();
      assert.strictEqual(
         $menu.css('display'),
         'none',
         'The menu is hidden after clicking other elements'
      );
   });

   QUnit.test('Destroy', function(assert) {        ◁─────
      assert.expect(5);
      var $fixture = $('#qunit-fixture').jqiaContextMenu({
         idMenu: 'context-menu'
      });
      var $fixtureDestroyed = $fixture.jqiaContextMenu('destroy');
      var $menu = $('#context-menu');

      assert.strictEqual(
         $fixture.length,
         $fixtureDestroyed.length,
         'Keep chainability'
      );
      assert.strictEqual(
         $fixture,
         $fixtureDestroyed,
         'Return the same object'
      );
      assert.strictEqual(
         $menu.css('display'),
         'none',
         'The menu is hidden'
      );
      assert.strictEqual(
         $fixture.data('jqiaContextMenu'),
         undefined,
         'Namespaced data cleared'
      );
      $fixture.trigger('contextmenu');
      assert.strictEqual(
         $menu.css('display'),
         'none',
```

> Verifies that destroy() keeps chainability, removes data-* attributes, and so on

```
      'The menu is still hidden after the click'
   );
});
```

If you want to run this test suite and play with it a bit (maybe introducing other meaningful tests?), you'll find it in the file chapter-14/test.suite.html.

This test suite has been created to pass all the tests, but to have a better grasp of the code written to test the plugin, you might want to see some tests failing. If you want to see some tests fail, try to change the code of the plugin in an unexpected way. If you need a suggestion, try to remove the `return this;` statement in the `init()` and the `destroy()` methods. Doing so will break the chainability of the plugin, and thus the relative assertions will fail.

This last demo showed you not only another example of most of the methods we covered in this chapter but also what a complete test suite looks like. We hope that you'll take our advice to heart and start testing your code more frequently if you don't do so already.

14.10 Summary

In this chapter we described the fundamental concepts of software testing and why unit-testing your code is so important. Testing gives you more confidence that your code is working properly and that it (virtually) doesn't have bugs.

We provided an overview of the frameworks available for unit testing JavaScript projects, focusing our attention on QUnit. This framework, maintained by the same team that offers you the lovely jQuery library, provides an easy-to-use set of methods to test your code.

After describing how to create tests with `QUnit.test()`, we introduced you to several assertion methods used to verify that the returned values of your functions and methods are what you expect. You also learned the importance of setting the number of assertions you expect to run through the use of the `assert.expect()` method.

Then you discovered how to test functions that run asynchronously—for example, those passed to JavaScript's native function `setTimeout()` or to jQuery's Ajax functions—with the help of `assert.async()`.

Finally, you learned how to organize your test suite into modules and how to set a project-specific configuration. With all this power in your hands, you developed a complete and working test suite. Our hope is that starting from tomorrow, or even today, you'll begin testing your code so that you can use and refactor it with more confidence.

In the next and last chapter of this book you'll discover some useful tools that can help you employ jQuery in large projects.

How jQuery fits
into large projects

This chapter covers

- Improving selectors for better performance
- Organizing your code in modules
- Loading modules with RequireJS
- Managing dependencies with Bower
- Creating SPAs with Backbone.js

If you've read all the previous chapters, you've hopefully learned how to write beautiful and concise code using jQuery, how to extend its features, and how to unit-test your code. Now that you know jQuery, you're ready to learn when it isn't enough and the use of another library or even a framework is required.

In this chapter, the last of this book, we'll broaden our focus to several tools, frameworks, and patterns not strictly related to jQuery but that can be used to craft fast, solid, and beautiful code.

The main purpose of jQuery is to help you manipulate the DOM. DOM manipulation is usually slow, so you need to understand how you can tweak the performance of your jQuery code to perform operations as quickly as possible. You also

have to understand how to easily integrate jQuery into large projects and how to correctly structure your code in modules to improve the maintainability of your code base.

One of the most important challenges developers deal with is the creation of performant code. Many people often underestimate this task, but optimizing your JavaScript-based source is always rewarding. It may happen that improving the performance is hard due to poorly written code that needs a deep refactoring, but other times it's as easy as selecting elements properly. In the first section of this chapter, we'll discuss extensively how you can improve the performance of code written with jQuery by selecting elements the right way.

When you work on large projects there's a strong need for a better code organization. If you don't manage it properly, when you add new features or refactor your code, you'll find yourself dealing with a complete mess. One way to address this issue is to apply some common and reliable patterns. The practice of organizing your code in modules, which is the second topic covered in this chapter, allows you to have a better overview of the whole project. It also allows you to easily organize the project so that different developers can work on different modules.

Splitting a project into modules is a good way to keep it organized but it introduces a problem. A given module may depend on others in order to work, so you need to be careful about the order in which you include them in your web pages. If you deal with a couple of modules, this issue is easy to manage, but when developing large projects, it isn't so simple. In situations where many modules, plugins, libraries, or frameworks come into play, each of them with its own dependencies, you need a professional and reliable method to include them in the right order. One of the possible approaches is to adopt RequireJS, a library that we'll cover in section 15.3.

In the previous paragraph we mentioned plugins, libraries, and frameworks. When developing a web project, it takes time to write everything from scratch, so you usually rely on third-party software. Examples of such software are jQuery and Modernizr. To include these components in your project, you visit the respective websites, download the files needed, and put them into a folder. Although this process works, it's slow and boring. In addition, it leaves you with the burden of manually checking for new versions. To automate this activity you can employ Bower, which we'll introduce in section 15.4.

Finally, in the last section of this chapter, we'll take a look at Backbone.js. The section isn't meant as a complete guide to this framework, but we want to give you an idea of what's next in your learning path and how jQuery integrates with frameworks such as Backbone.js to create complex applications.

15.1 *Improving the performance of your selectors*

Achieving high performance is a huge concern, today more than ever. Every web project should take care of this aspect from the start in order to avoid publishing pages that take up to 10 seconds to load.

Performant code isn't only something to brag about with friends; it can improve the satisfaction of your users. In this section, you'll learn some tips and tricks to speed up your code by selecting elements with jQuery the right way.

15.1.1 Avoiding the Universal selector

The first and simplest advice we can give you to improve the performance of a selection is to avoid the Universal selector unless it's absolutely needed. If in your code you have a selection like

```
$('form :checkbox');
```

it's equivalent to

```
$('form *:checkbox');
```

As you might recall, when a selector is omitted in front of a filter, the Universal selector is implicitly assumed. To improve the performance of the previous selection, you should turn it into

```
$('form input:checkbox');
```

or even better, recalling what you've learned about the `context` parameter, into

```
$('input:checkbox', 'form');
```

This last selection is in most cases faster than the one you saw at the beginning of this section.

Another case where you should avoid the use of the Universal selector is when you're retrieving all the direct children of a given element. An easy solution to achieve this task that uses the Universal selector is

```
$('form > *');
```

But you can do better than that! A better approach is to use the tag name selector and then employ jQuery's `children()` function:

```
$('form').children();
```

This solution is better because it allows jQuery to call the native JavaScript `get-ElementsByTagName()` function, which is very fast.

The principle shown in the last example can be applied to other cases. Remember that the best performances are achieved when jQuery can call the JavaScript native functions like `getElementById()` (the fastest function among similar ones), `get-ElementsByTagName()`, and so on.

15.1.2 Improving the Class selector

Throughout the book you've discovered that, when possible, jQuery uses the JavaScript native functions to speed up the operations performed.

To select elements based on their class name, the library uses the `getElementsBy-ClassName()` function behind the scenes in browsers that support it: IE9+, Firefox 3+,

Chrome, Safari, Opera 9.5+, and many others. In versions of Internet Explorer prior to 9, jQuery is still able to give you the expected result but it has to rely on an implementation of its own. Because of this, if the page contains a huge number of elements, the selection process can be slow.

If you want to improve the performance for Internet Explorer 6–8—for example, if they're the only browsers you're targeting (some organizations are really stuck in the past)—you can optimize the search by combining the Class selector with the Element selector. Specifically, you can prepend the latter to the class name of interest.

For example, if you want to select all of the p elements having class `description` and store them in a variable, you can write

```
var $elements = $('p.description');
```

This is a good start to improving the performance of your code, but there's more that you can do.

15.1.3 *Don't abuse the context parameter*

Back in chapter 2 we introduced the second parameter of jQuery() called `context`. We stated that when using this parameter it's usually possible to improve the performance of a selection by restricting the latter to one or more subtrees of the DOM, depending on the selector used.

There are cases where the use of `context` doesn't improve the performance, though. For example, if you're selecting an element by its ID, you won't reap any benefit in specifying `context`. Even more, in this specific case, you're worsening the performance. Therefore, avoid writing statements like

```
var $element = $('#test', 'div');
```

because it'll worsen the performance compared to

```
var $element = $('#test');
```

The first solution is slower because the library has to retrieve a potentially large number of <div>s first and then test their descendants, instead of immediately taking advantage of the native `getElementById()` function. In case you wonder how slow it could be, take a look at the astonishing results of the test at http://jsperf.com/jquery-context-parameter that are displayed in the chart of figure 15.1.

This chart was created using jsPerf (http://jsperf.com), a service that allows you to create and share test cases. It's a good alternative if you don't want to run a test on your machine or if you want to share the results.

Thanks to this example, we can extract another good point. In order to allow jQuery to use `getElementById()`, you should never prepend a tag name to an ID, so avoid writing a selector like `$('p#test')`.

A case where the use of `context` can often speed up the performance is when you provide an ID. However, this rule isn't set in stone. In fact, when dealing with performance there isn't a rule that's always true or false, and you need to test case by case.
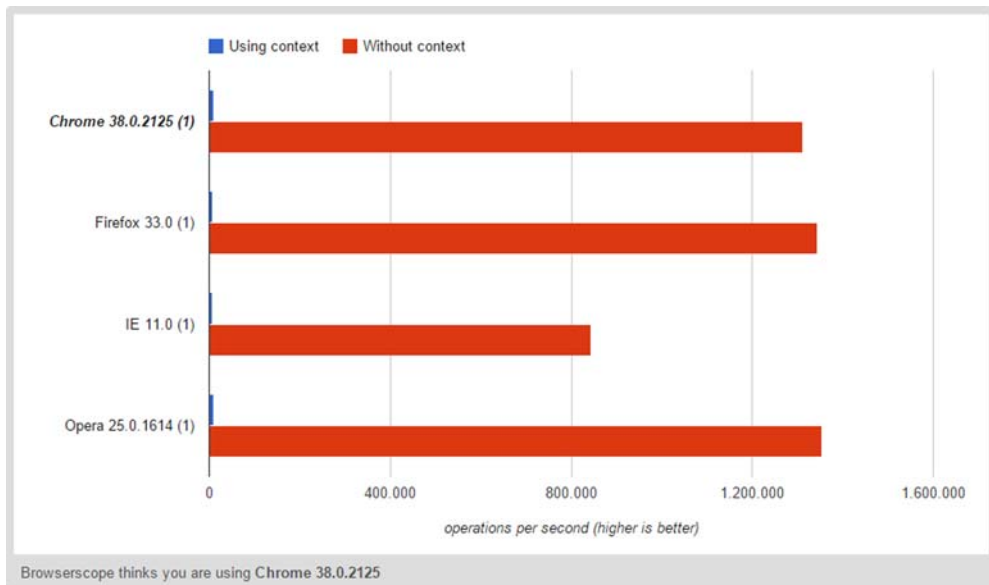
Figure 15.1   A performance test of a selection of an element using its ID with and without the use of the `context` parameter (higher is better)

Performance depends on a lot of factors such as the number and type of the elements in your page and the browser. The take-away lesson is test, test, and once again, test your selectors to verify what works best in that specific case.

The possible optimizations aren't over yet. Let's see what you can do with filters.

15.1.4 *Optimizing filters*

Many of the filters supported by jQuery aren't part of the CSS specification, so they can't take advantage of the performance provided by the use of native methods such as `querySelectorAll()`. For some of them, such as `:input`, `:visible`, and others, it's better to first select using a pure CSS selector and then filter using the `filter()` method. For example, instead of writing

```
$('p:visible');
```

you can write

```
$('p').filter(':visible');
```

For other filters, such as `:image`, `:password`, `:reset`, and others, you can take advantage of the attribute selector instead. Imagine you want to retrieve all the reset buttons in a page. You could use the `:reset` filter writing

```
$(':reset');
```

but you can optimize this selection by turning it into

```
$('[type="reset"]');
```

In this selection you're implicitly using the Universal selector that we said you should avoid. To further improve this selection, you can prepend an Element selector as shown here:

```
$('input[type="reset"]');
```

During our exploration, you've met the position filters and in particular :eq(), :lt(), and :gt(). Like other filters discussed in this book, they're jQuery extensions and aren't supported by CSS. To improve the performance in cases where you need to use them, you can select the elements and then employ the eq() method as a replacement for the :eq() filter. By doing so, you allow jQuery to take advantage of the native JavaScript methods. As a replacement for :lt() and :gt(), you can use the slice() method.

Based on these suggestions, if you want to select the first two list items in a list, you can write

```
$('#my-list li').slice(0, 2);
```

instead of

```
$('#my-list li:lt(2)');
```

The last optimization we want to mention here concerns the :not() and the :has() filters. In browsers that support querySelectorAll(), the former can be replaced by jQuery's not() function, and the second can be replaced by jQuery's has() method.

For example, you can turn

```
$('input[placeholder!="Name"]');
```

into

```
$('input').not('[placeholder="Name"]');
```

With this last piece of advice, we've completed the optimizations applicable to filters. But there's a last pearl of wisdom we want to share with you.

15.1.5 *Don't overspecify selectors*

jQuery relies on a selector engine, called *Sizzle*, that parses selectors from right to left. This means that in order to speed up a selection, you must be more specific on the right side and less specific on the left. To give you a concrete idea, imagine you want to select all the <span>s having the class value within a <table> with the class revenue. When performing a selection it's usually better to write

```
var $values = $('.revenue span.value');
```

instead of

```
var $values = $('table.revenue .value');
```

Because of the previous statement and the example we showed you, you might be tempted to overspecify selectors, especially on the right side. Don't do it! If the same

set of elements can be retrieved with fewer selectors, get rid of the useless parts. Therefore, avoid writing statements like

```
var $values = $('table.revenue tr td span.value');
```

if you can select the same elements with

```
var $values = $('.revenue span.value');
```

The former statement is harder to read and won't boost performance.

The advice we've given in this section should help you to optimize your selections. Let's now discuss how to improve the structure of a project by organizing it into modules.

15.2 *Organizing your code into modules*

When working on large applications you have to pay attention to organize your code properly. Limiting global namespace pollution and providing logical module organization are priorities, and code written using jQuery is no exception. You should take care of its structure in the same way you would do for any other code written without it.

The simple approach those starting with JavaScript adopt is to define several functions and objects in a file, as shown here:

```
function foo() {};
function bar() {};
function baz() {};
var obj = {};
var anotherObj = {};
```

The problem with this code is that all these functions and objects become global (available as properties of the `window` object), so sooner or later a library will come in and overwrite one of them. In addition, it doesn't consider that you may want to keep some data private, a problem we already discussed when talking about jQuery plugins and how the jQuery library is structured.

Another problem is that if these functions and objects serve different roles and are used in different parts of the project, the only way you have to recognize those roles is by reading the names you assigned. Let's say that you have JavaScript code for an ecommerce website, and that the `obj` object and the `bar()` function are needed for the payments part of the application, and the `baz()` function and the `anotherObj` object are required for the basket. How you can distinguish them?

Using more technical terminology, you can say that your application has two *modules*: payment and basket. Modules play an important role in the robust architecture of an application and, as you'll see, help you keep the units of code for a project separate and organized. In JavaScript, there are several options for implementing modules: AMD (asynchronous module definition, discussed in section 15.3), ECMAScript 2015 (also known as ECMAScript 6), object literals, CommonJS, and others.

In the following sections you'll learn some patterns to use to organize your code into modules.

15.2.1 *The object literals pattern*

One of the simplest techniques you can adopt to organize code into modules is to use object literals. To simplify the explanation of this approach we'll break the discussion into simple steps.

The first step, which lets you avoid polluting the global namespace, is to "namespace" the functions, objects, and other variables you defined using an object literal:

```
var myService = {
   foo: function() {},
   bar: function() {},
   baz: function() {},
   obj: {},
   anotherObj: {}
}
```

With this small change you can access the same functions and objects as before but through one entry point. Because of that, the likelihood that a library would override your code is decreased, but you haven't solved the issue of separating the functions and objects based on their role.

The previous approach can be extended further to solve this issue:

```
var myService = {
   foo: function() {},          ⟵————— Defines the only function that belongs to the core
   payment: {                   ⟵
      obj: {},                       Defines the payment module
      bar: function() {}
   },
   basket: {                    ⟵————— Defines the basket module
      anotherObj: {},
      baz: function() {}
   }
};
```

With this structure, you could call the `baz()` function of the `basket` module as shown here:

```
myService.basket.baz();
```

Once you have your code logically separated, you can even place each module in a different file. You might have a basket.js file, containing the `basket` module, defined as such:

```
myService.basket = {
   anotherObj: {},
   baz: function() {}
};
```

Thanks to the division into multiple files, different developers can easily work on a single module, reducing the chance of stepping on each other's toes.

Although this approach solves some issues, it isn't well suited for keeping data private that's globally available inside the module. In other words, you need a way to

create functions and objects accessible only inside the module but not outside it. (Inside a function you can create data available to that function only.) In the next section we'll present a better methodology that's also able to deal with this issue.

15.2.2 *The Module pattern*

The Module pattern has been adopted in JavaScript to *emulate* the concept of private methods and variables inside an object like in OO languages such as Java and C#. We used the term *emulate* because technically speaking there aren't access modifiers like `private` and `public` in JavaScript.

The Module pattern consists of two main components: an IIFE (more on this concept in the appendix of this book) and an object to return or augment. To give you a quick grasp of the subject, here's a simple example of code implementing this pattern:

```
var myFirstModule = (function() {
   return {
      foo: function() {},
      bar: function () {},
      obj: {}
   }
})();
```

This code creates an IIFE and inside it returns an object literal containing the functions and the objects you want to expose publicly.

At first glance, this pattern doesn't seem very different from the previous one, but wait until the end of this section before judging. Thanks to this approach you're able to create variables and functions accessible within the module that from the outside are inaccessible. Let's say that you want to add to the previous code a "private" variable, called `count`, that keeps track of the number of times the `foo()` function is invoked. You also want to define a "private" function `doSomethingPrivate()`, which is invoked whenever the `bar()` function is executed. The code to achieve this goal is shown here:

```
var myFirstModule = (function() {
   var count = 0;                           ⟵——— Declares a "private" variable
   function doSomethingPrivate() {};        ⟵——— Declares a "private" function

   return {
      obj: {},
      foo: function() { count++; },         ⟵——— Increments the value of count
      bar: function () { doSomethingPrivate(); } ⟵┐ Invokes
   }                                             │ doSomethingPrivate()
})();
```

Now that you have a better understanding of the Module pattern, we can introduce you to one of its variations. This variation lets you augment the `basket` module discussed in the previous section:

```
window.myService = (function(oldMyService) {
    oldMyService.basket = {
        baz: function() {},
        anotherObj: {}
    };

    return oldMyService;
})(window.myService || {});
```

❶ Creates an IIFE and assigns the result of its execution to window.myService

❷ Creates a basket property on oldMyService and assigns an object literal to it

❸ Returns the updated oldMyService object

❹ Passes a previously defined window.myService as an argument or an empty object if it isn't defined

The previous code is small but employs a couple of interesting techniques. First, you assign whatever is returned by the IIFE to a property called `myService` defined on the `window` object ❶. The IIFE has only one parameter defined, called `oldMyService`, that will receive the value of a previously defined `window.myService` property or an empty object in case it isn't defined (that is, in case its value is *falsy*) ❹. This way, you can augment the `myService` property with another module (if it already has at least one) or create a first module.

Inside the IIFE you define a property called `basket` of this object that represents your module, where you define the functions and the properties you want to expose publicly ❷. Finally, you return the augmented object ❸. This statement is necessary if the value of `window.myService` was falsy and an empty object literal was passed to the IIFE.

With this last example we've concluded our brief overview of how to organize a project into modules. There are other patterns that we haven't covered in this section, but now you should be ready to keep your code clean and more manageable. In the next section we'll discuss another pattern for creating modules, called AMD, and will introduce you to RequireJS, a library created to load modules while respecting their dependencies.

15.3 *Loading modules with RequireJS*

In the previous section we discussed two simple approaches to organize your code into modules. But they have a major drawback: you have to manually manage the dependencies of each module. For example, a method of one module may need to use a property of another object. To avoid this issue you have to pay attention to the order in which you include them in your web pages, but for a project that has tens or hundreds of modules this becomes difficult and error-prone. What makes this solution even harder to employ is that some modules may depend on third-party plugins, libraries, or frameworks.

One method for solving this issue would be to define the dependencies of the modules and then have "something" that organizes the inclusions in the right order for you. This is where asynchronous module definition and RequireJS (http://requirejs.org/) come into play.

The asynchronous module definition (AMD) is a JavaScript API that specifies a mechanism for defining modules so that the module and its dependencies can be asynchronously loaded.

RequireJS is a JavaScript file and module loader that's optimized for in-browser use but that can be used in other JavaScript environments, like Rhino and Node.js. This library is highly configurable, allowing for a high level of flexibility, but it's also possible to start simple to accommodate basic needs. In this section we won't cover the library in detail, but we'll describe it enough to get you started.

RequireJS loads each dependency as a script tag inside the `head` element of the page. Then the library waits for all dependencies to load and calculates the right order in which to call the functions that define the modules. Finally, it calls the module definition functions in the appropriate order.

Now that you know what RequireJS is and what it does, let's start using it.

15.3.1 Getting started with RequireJS

The first step you need to perform is to download the library. Access the download page at http://requirejs.org/docs/download.html and download the latest version available.

Before you can use RequireJS in your web pages, we need to discuss some of its main concepts. The first topic we want to cover is the `define()` function defined by AMD.

Method syntax: define

`define([[id, ] dependencies,] factory)`

Define a new module with optional dependencies and identifier.

Parameters

id	(String) The identifier of the module.
dependencies	(Array) An array containing the name of the modules the new module depends on.
factory	(Object|Function) An object literal or function that defines the new module. When a function is provided, it receives as parameters the dependencies in the order in which they are defined.

Returns

undefined

To fix the idea, let's say that you have an object called `Person`, defined in a file called Person.js. Its only property is `name` and it has no dependencies. Using the `define()` function you can create it as such:

```
define({
   name: 'John Doe'
});
```

As you can see, you declare neither an ID for this module nor dependencies.

In addition to `Person`, you have an object named `Car` that's stored in a file called Car.js. This object has a method called `getOwner()` that internally uses the property name of the `Person` object (literal); thus it has the `Person` module as a dependency. The `Car` module can be defined as shown here:

```
define(['Person'], function(Person) {
   function Car() {
      this.getOwner = function() {
         return 'The owner is ' + Person.name;
      };
   }

   return Car;
});
```

In the code you include `Person` as a dependency and then create a function named `Car` that acts as a constructor. This function has a `getOwner()` method that returns a simple message that uses the `name` property of `Person`. Finally, `Car` is returned to be available as a module. So far, you've defined two modules, but you still haven't used them. The `require()` function exists for such a purpose.

The `require()` function is similar to `define()` in that both define a module, but the former also executes it. This means that it loads and executes the dependent modules before executing the function provided. Usually an application has one `require()` function as a main entry and other modules defined via `define()`.

To conclude our example, imagine you have a file called main.js acting as the entry point of your application, where you want to alert the name of the car's owner. To do so, you can use `require()` defining `Car` as its dependency:

```
require(['Car'], function(Car) {
   var car = new Car();
   alert(car.getOwner());
});
```

If you've followed this section carefully, you should have a question blinking in your head: how does RequireJS know from a simple string (for example, `"Car"`) what module to load? The answer is that the library creates a module having the same name as the file containing the definition. In the example, even though you didn't define the ID for your modules, you have three module names: main, `Car`, and `Person`. The reason is that you have three JavaScript files: main.js, Car.js, and Person.js.

Let's take this convention even further. If you have a file named Basket.js that's stored in a folder called cart, the module will be named `cart/Basket`.

The final step to let RequireJS do its job is to include it in an HTML page. You can do so using a `script` element with the addition of a `data-main` attribute. This attribute is used to define the entry point of your application (the file using `require()`). Assuming that you've placed the RequireJS library and all the previously created modules in a folder called scripts, you could start the demo by writing

```
<script data-main="scripts/main" src="scripts/require.min.js"></script>
```

A working example that employs all the snippets we've developed in this section can be found in the folder chapter-15/requirejs.

Now that we've covered some basic concepts, let's see how you can apply them and use RequireJS with jQuery.

15.3.2 Using RequireJS with jQuery

Back in chapter 12, we taught you what jQuery plugins are and how you can develop your own to extend jQuery's powers. Being plugins for jQuery, they depend on, well, jQuery. And the code you write using your jQuery plugins' methods depends on jQuery and these plugins. This is a perfect situation in which to employ RequireJS in your project.

Your plugins haven't been developed to use AMD from the start. Therefore, it would seem that the only solution would be to go through every plugin and adapt them to use `define()`, declaring jQuery as a dependency. Fortunately, there's a better approach, and that's the subject of the next section.

JQUERY PLUGINS USING AMD

If you're developing a plugin from scratch and you want to use RequireJS, you can wrap the definition of the plugin with a call to `define()`, declaring jQuery as a dependency, as follows:

```
define(['jquery'], function($) {
   $.fn.jqia = function() {
      // Plugin code here...
   };
});
```

What this snippet reveals is that you don't need to return the module because you're augmenting the original jQuery object. Furthermore, you don't need to wrap the definition of your plugin inside an IIFE because you already have a function that wraps it, and the jQuery object will be provided by RequireJS. As it is, the previous code doesn't work because RequireJS isn't able to resolve the "jquery" string specified as a dependency into the jQuery library. An easy way to solve this issue, recalling the conventions RequireJS employs, is to rename the jQuery file as jquery.js and place it in the same directory as the main entry. Once you've done that, you'll be ready to use your plugin.

Let's now assume that the plugin you wrapped was stored in a file called jquery.jqia.js. We'll also assume the entry point of your project is stored in a file called main.js and that it depends on jQuery and your plugin. With this in mind, your main.js file should look like the following:

```
require(['jquery', 'jquery.jqia'], function($) {
   // Code that uses jQuery and the plugin
});
```

In this example there are two details to highlight. The first is that because the code relies on the plugin, the latter is defined as a dependency. The second is that you don't need to add a second parameter to use the plugin, because once the plugin is loaded it augments the original jQuery object.

ADAPTING EXISTENT JQUERY PLUGINS

When starting a new project, planning to structure it in modules that employ AMD is easy. But often you have to maintain older libraries or use third-party software that wasn't created with AMD in mind. For such situations, you can create a configuration file for RequireJS that allows you to avoid changing these files to use `define()`:

```
requirejs.config({
    shim: {
        'jquery.jqia': ['jquery']
    }
});
```

This configuration employs the `shim` property and must be placed before the `require()` call. The `shim` property has an object as its value and enables you to specify dependencies for jQuery plugins that don't call `define()`. The object literal assigned to `shim` must define the name of the modules as properties and their array of dependencies as values.

Based on this description, the final code of main.js is as follows:

```
requirejs.config({
    shim: {
        'jquery.jqia': ['jquery']
    }
});

require(['jquery', 'jquery.jqia'], function($) {
    // Code that uses jQuery and our plugin
});
```

With this last example you've seen how to adopt RequireJS in projects that take advantage of jQuery and jQuery plugins. The concepts described are far from being a complete guide to RequireJS, and there's a lot more to discuss, like the optimizer and the many configuration properties provided. But this is a good start to employing this library to organize the dependency of your projects.

In the same way that you need a better way to manage the dependencies of a module and the order in which you should include them in your projects, you need a better and faster method to install, update, and even delete third-party software that your project uses. That's exactly what we'll discuss next.

15.4 *Managing dependencies with Bower*

The development of a web project usually involves the use of third-party components to speed up the process. A project that employs one or two third-party components can be easily managed manually, as we used to do until a few years ago. As things get more complicated, developers needed a reliable way to install and manage those project dependencies.

In the past few years a lot of tools have been released to address this problem. One of these tools is Bower (http://bower.io/). In this section, we'll look at it and its main features, focusing our attention on how jQuery can be integrated into a project using Bower.

15.4.1 *Getting started with Bower*

Bower was created at Twitter and released in 2012. Since then, a lot of developers have contributed to this project, and now it's one of the most well-known front-end tools. Bower is defined as "a package manager for the web," which means that it's a dependency manager for JavaScript, CSS, and much more, such as web fonts. A package can be a JavaScript library (such as jQuery, jQuery UI, or QUnit), a CSS file (such as Reset.css or Normalize.css), an iconic web font (such as FontAwesome), a framework (such as Bootstrap), a jQuery plugin (such as jQuery Easing or jQuery File Upload), or anything else a developer wants to expose as a third-party component for a web project.

Funny enough, Bower has some dependencies itself, so you need to satisfy those dependencies before you can use it. These dependencies are Node.js (http://nodejs.org/), the platform that enables you to run JavaScript as a server-side language and that we've mentioned a few times in this book; npm (https://www.npmjs.com/), which is installed with Node.js; and a Git client (http://git-scm.com/downloads). Once you've installed them, you're ready to enter the Bower world.

Bower defines a *manifest file* called bower.json, written in JSON format, which provides information about the project such as its name, the author(s), the current version, and the packages used. This manifest file comes in handy if you're working in a team because it lets you share the information with the other members. This is useful because they can install all the dependencies of the project by typing a single command (we'll discuss it in a few moments).

Once you've installed all the Bower dependencies, you can install Bower by running on the command-line interface (CLI) the command

```
npm install -g bower
```

This process can take up to a few minutes, but once it's completed you're ready to employ this tool in your projects.

Let's now say that you're developing a new project and you want to use Bower to manage the dependencies. To start, you need to move to the project's folder and create the bower.json file inside it. This file can be created either manually or with the help of Bower. In this example we'll discuss the second option. Open up the CLI, move to the project's folder, and run the command

```
bower init
```

The tool will ask you some information about the project, as shown in figure 15.2.

After you've filled in all the fields and confirmed the information, the manifest file (bower.json) will be created inside the folder. An example of a bower.json file is shown in the following listing.

Figure 15.2   Using Bower to create the manifest file of a project

---

Listing 15.1   An example of a bower.json file

```
{
  "name": "jqia-context-menu",
  "version": "1.0.0",
  "authors": [
    "jqia-team <test@test.com>"
  ],
  "description": "A jQuery plugin to show a custom context menu on one or
    more specified elements of a page.",
  "main": "src/jqia-context-menu.js",
  "keywords": [
    "jQuery",
    "plugin",
    "context-menu"
  ],
  "license": "MIT"
}
```

Your project is now set up to use Bower, but so far you haven't done anything really exciting or very useful. In the next two sections we'll revamp your interest.

15.4.2   *Searching a package*

A package in Bower is nothing but a component that you want to use in a project. Not all the libraries and frameworks available on the web can be managed with Bower, but because Bower comprises more than 34,000 packages, you can be pretty sure that everything you may need is already available.

   If you want to know if a package is available, you can search it using Bower. To do that, open the CLI and run the command

```
bower search <package_name>
```

where <package_name> stands for the name of the package.

   And what better example could we propose than searching for jQuery? To search for jQuery using Bower you have to run the command

```
bower search jquery
```

The execution of this command will give you as a result not only the jQuery library itself but also all the other packages that have the string "jquery" in their name, description, or keywords.

Once you've identified the exact name of the package to be used, you're ready to install it.

15.4.3   *Installing, updating, and deleting packages*

Before installing a package, there's an important decision to make. You have to decide if the dependency you're going to install is needed in production or is only necessary for you as a developer.

To give you a concrete idea, jQuery is a package you need in production because your whole JavaScript code, or part of it, needs jQuery to work. The same reason is valid for other components like jQuery UI or Backbone.js. Other packages, such as a testing framework (like QUnit or Mocha), are only needed while developing the project to ensure the code quality and robustness. No other parts of the software—at least not those that will be deployed—need it. This is an important difference that will slightly change the way you install a dependency.

To install a package with Bower you have to run the command

```
bower install <package_name> <--production-or-development>
```

where `<package_name>` is the name of the package and `<--production-or-development>` is a flag to specify if the package is intended for development purposes only (`--save-dev`) or not (`--save`).

To install jQuery as a production dependency, open the CLI and move the project's folder to the same level as the bower.json file. Once that's done, execute the command

```
bower install jquery --save
```

Let's now say that you also want to install QUnit because you want to unit test your project. To install it as a development dependency, the command to use is

```
bower install qunit --save-dev
```

The first time the `install` command is executed, a folder called bower_components is created. Inside this folder the tool downloads the packages required. It also updates the bower.json file by adding the package in the `dependency` or the `devDependency` section, depending on the option specified.

Once a dependency—for example, jQuery—is downloaded, you need to include it in your project. Assuming you have an index.html file created at the same level of the bower_components folder, you have to write

```
<script src="bower_components/jquery/dist/jquery.min.js"></script>
```

The actual path varies from package to package, but the structure is usually similar.

Once you install all the dependencies needed, you can start developing the features of the project.

The development process usually requires a lot of time, and while writing the code it can happen that a new version of one or more of the packages you're using is released. New releases often fix important bugs, so it's important to keep your dependencies up to date.

UPDATING A PACKAGE

Updating a package is easy. All you need to do is to move to the root of the project and execute on the CLI the command

```
bower update <package_name>
```

If you want to update jQuery, you have to write

```
bower update jquery
```

Sometimes you might want to update all of your packages at once. Bower enables you to do that with the command

```
bower update
```

There may be situations where a dependency isn't needed anymore and you want to delete it. Let's see how.

REMOVING A PACKAGE

To delete a dependency using Bower you can run the command

```
bower uninstall <package_name> <--production-or-development>
```

where the meaning of the two parameters is the same as explained previously.

Imagine that you gave QUnit a try in your project but you didn't like it very much and decided to write your tests using Mocha. You need to delete QUnit, which was installed as a development dependency. To do that, you need to run on the CLI the command

```
bower uninstall qunit --save-dev
```

With this last example we've completed our overview of Bower. This tool has more commands than those we've discussed, but they're enough to get you up and running and to speed up your workflow.

Thanks to jQuery, Bower, RequireJS, and the other software we've introduced in this book, you're ready to develop websites and web applications in a solid and professional way. But this chapter would be incomplete without mentioning single-page applications and how to create them using an MVC framework.

15.5 *Creating single-page applications with Backbone.js*

As we discussed in the introduction, when working on large projects there's a real need to have good code organization. Imagine for a moment what would happen to software developed by companies like Google or Microsoft if they had bad code organization. With products where new features are added and old ones are updated frequently, poorly structured code leads to an incredible amount of time wasted. And as you know, time is money.

One of the most used patterns to structure software is the Model–View–Controller (MVC) pattern. It's a software architectural pattern that separates concerns into three main concepts: model, view, and controller. The model represents the application data, such as a registered user of a website. The view is the component that deals with displaying the data—that is, how the data will be shown in the web page if you're using this pattern on the web. The controller updates the model's state and sends the data to the view (for example, a change in the address of the user).

If you were a PHP developer, you'd be aware of Symphony, Laravel, or Zend Framework; if you were a Java developer, you'd probably have heard of Spring Web MVC or Struts. But because we're talking about JavaScript, we have other frameworks. One of these frameworks that implements the MVC pattern is Backbone.js.

Ideally, jQuery could be used to create single-page applications, but because that isn't its main purpose, more often than not you'll end up with complex, hard-to-write, and hard-to-maintain code. To create such applications, you need a framework specifically developed with this scope in mind, such as Backbone.js. Backbone.js has some libraries on which it relies: Underscore.js and jQuery. Therefore, the knowledge you've acquired so far will be useful even in this context.

Today many web applications rely on JavaScript and Ajax to create better user interactions and to create single-page applications (SPAs). These are applications that, once loaded in the browser, perform all the HTTP requests without requiring the whole page to reload. In terms of performance, this is a big advantage because the browser doesn't have to load all the assets (JavaScript files, CSS files, fonts, and so on) again. It loads only the small portion that has to be injected into the DOM, returned by the server.

The presented approach doesn't come without cost. SPAs usually have a longer loading time because they require loading more code than other applications. For this reason, you have to pay attention to balancing the code needed by your application and its weight. A page that takes too long to load is at risk of losing users.

In the following sections, we'll discuss the features of the pattern implemented by the framework and also give a brief overview of its main concepts such as models, views, and routers. In addition, we'll develop a simple yet useful application to organize to-do lists (from now on we'll refer to it as the Todos manager). We chose this application because it's often used as a project when learning a new framework. Its scope is to save the tasks you have to do so you can easily remember them. Please note that this section isn't intended to be an in-depth guide to Backbone.js, and we'll only skim the surface. If you want a complete resource, you can buy a book dedicated to this framework.

15.5.1   Why use an MV* framework?

Just like a lot of things in life, programming is a cycle. You have a problem that you solve using the tools you have at a given moment, but then new libraries and frameworks are released to improve the code and the solutions adopted. Then new issues arise and the cycle starts again.

When the first SPAs came out, many developers started to employ jQuery (or similar libraries) and a lot of callbacks and Ajax calls to synchronize the user input with the data stored on the server. Once these applications increased in complexity, the code became unmaintainable and unscalable. As a consequence, developers needed a framework to help them with the development of a well-structured and maintainable code. That's where frameworks like Backbone.js, AngularJS, Ember, and many others came into play. They're usually referred to as MV* frameworks because they don't quite fit either the MVC pattern, the MVP (Model-View-Presenter) pattern, or the MVVM (Model-View-ViewModel) pattern.

An illustration representing the MVC pattern and the Backbone.js implementation of this pattern is shown in figure 15.3.

As you can see from figure 15.3, the main difference between the classic MVC pattern and the Backbone.js implementation lies in the view component. The view also acts as a controller, handling the rendering of the UI that's represented by the template block, and has the responsibility of updating the model.

Figure 15.3 The MVC pattern compared to the Backbone.js implementation

Now that you're aware of the model implemented by Backbone.js, let's delve into each component, one at a time, to analyze its responsibilities and features.

15.5.2 Starting with Backbone.js

As you've seen, Backbone.js allows developers to break their code into small pieces. Let's take a brief look at its components.

MODEL

Models are the objects representing the data of your application. Models have as properties the attributes that feature the objects. Here you usually add the methods to validate the data, initialize the properties, and notify the server about changes in the model.

To explore the idea of what a model is, think for a moment about the Todos manager you're going to build. One of the models of your application (actually the only one you'll have) is the Todo entry, which is a single activity to perform. Each of these Todos will have a title, a position inside the list, and a property to indicate if it has been completed or not. These three attributes of the object are the properties of the model.

Models are completely agnostic about how the information they contain will be displayed, and each of them can be connected with one or more views that listen for changes. In this way, you ensure that what is displayed is in sync with the data described by the model. In Backbone.js, models are created by extending the `Backbone.Model` object. You can also group models into a unique entity, called a *collection*, which is the subject of the next section.

Here's an example of a Todo model having the properties just described:

```
var todo = Backbone.Model.extend({
   position: 1,
   title: '',
   done: false
});
```

As you can see from the code, in its basic form a model is just a set of properties declared into an object passed to the `Backbone.Model.extend` method.

COLLECTION

A collection is a set of models and it's used to organize and perform operations on the models included in it. When you define a collection, you need to set a property that specifies the type of the collection you're creating.

Collections help you avoid the need to manually observe single model instances. In the Todos manager application, you'll need a way to group the Todos because you'll want to represent them as a unique list. Therefore, you'll use a collection.

In Backbone.js, you create a collection by extending the `Backbone.Collection` object. A collection can have one or more views listening for changes.

The following example shows how you can create a collection of Todos using the model shown in the previous section:

```
var todoList = Backbone.Collection.extend({
   model: todo
});
```

As you can see, a collection can be as simple as an object containing as its only property the specification of the models it contains.

VIEW

The view is the component that responds to DOM events by executing one or more methods based on your needs. It's usually tied to a specific model. Views help you keep the DOM in sync with the data, and they're the components where you write the logic behind the presentation of the data.

This component doesn't contain the HTML code. The HTML code is written in *templates* managed with other JavaScript libraries like Mustache.js or Underscore.js. Because Backbone.js relies on Underscore.js, in our demo project we'll stick with the latter.

Recalling our example of the Todos manager, the view represents the object that will allow you to listen for DOM events and run one or more methods accordingly. To give you a concrete idea, you can think of a DOM event as the click on the Add Todo button or the addition of a new model in your list of Todos.

The most important property of a view is el. It's a reference to a DOM element that all views must have, and it ties the View object to a DOM element. Often you'll use jQuery and its methods on el, so Backbone.js defines a convenient $el property, which is nothing but a jQuery object that wraps the el property.

The el property can be associated with a DOM element in two ways. The first way is to create a new element for the view and then add it to the DOM. In this case, the new element is created by the framework and the reference is assigned automatically. In addition, you can use some other properties to set common attributes of a DOM element. The properties discussed are tagName, which is the name of the tag (as div, p, li), id, and className, which is the CSS class name to assign. The second way is to reference to an element that already exists in the page.

Another important although optional feature of a view is the render() method. In it you define the logic behind the render of a template—the statements that will actually render the HTML representing the model. In the render() method you'll usually have a JSON object of the model associated with the view that's passed to the template to populate it with this data and show the HTML to the user. To be precise, you'll compile a template into a function using Underscore's _.template() method and pass the JSON object to this function. For example, you'll have code like the following:

```
var TodoView = Backbone.View.extend({
   render: function() {
      this.$el.html(this.template(this.model.toJSON()));

      return this;
   }
});
```

The following example creates a `li` element using the first approach discussed to associate the `el` property to a DOM element and also defines a template:

```
var TodoView = Backbone.View.extend({
    tagName: 'li',
    className: 'todo',
    template: _.template($('#todo-template').html())
});
```

A template is a piece of HTML containing some template tags that will be replaced by the data store in a model. For example, let's say that you want to show the title of a Todo; you might create a template like the following:

```
<script type="text/template" id="todo-template">
    <span class="todo-title"><%- title %></span>
</script>
```

To sum up, users interact with the HTML markup contained in templates that are managed by views. Views are responsible for notifying the model and, eventually, for modifying part of the HTML.

Views are also responsible for passing the models to the templates. The templates contain placeholders where you can show the attributes' values. These placeholders are replaced on the fly with the actual values of the models. You can also employ other structures like conditional statements that use the data passed to decide if a given HTML element or attribute has to be shown or not.

As you've seen from the template snippet, templates are created by inserting their content into a `<script>`. They usually have a `type="text/template"` attribute and an ID so you can easily retrieve them using jQuery or similar libraries. An example of the template tags cited previously is `<%- title %>`, where `<%- %>` is used to interpolate the value of the variable and HTML-escape it, and `title` is the name of the property you expect to be sent from the view.

ROUTER

A router provides a way to tie parts of your project to a given URL and to keep track of states in the application. A router maps a path to a function. It usually works with one or more models and then updates a view. Figure 15.4 shows an abstraction of how the router component fits into the general Backbone.js architecture.

Routers translate a URL or hash fragments of the URL into an application state. This means that they're needed if you want to make a given state of your application

Figure 15.4   A schema of how a router interacts with other components of Backbone.js

sharable or bookmarkable. Once one of the paths defined in the router matches the current URL, the relative function is executed.

To illustrate the idea, let's recall our Todos manager and imagine that you want to enable the user to ask for a specific Todo in order to see its details. This is where a router comes into play. You can create a router and associate a URL—for example, todo/MY-TODO-ID—with a function to execute that updates the page, removing the list of Todos and showing just the details of the one required.

Routers are defined by extending the `Backbone.Router` object, and you usually will have only one router per application, although you can have as many routers as you need. An example router definition is shown in the following listing.

Listing 15.2 A simple router in Backbone.js

```
var TodoRouter = Backbone.Router.extend({
    routes: {                                      Defines routes
        "todo/:id": "getTodo",               ❶ and maps them
        "search/:string": "searchTodo"          to functions
    },

    getTodo: function(id) {
        // Your code here
    },                                          Declares functions
                                            ❷ to execute on
    searchTodo(string) {                        route match
        // Your code here
    }
});
```

In the router example shown, you define two routes ❶, `todo/:id` and `search/:string`, assigning an object to the `routes` property. The object has as keys the pattern of the routes and as values the function to execute when the route matches the URL (for instance, "todo/2" matches the first route defined). In this example, `todo/:id` and `search/:string` are mapped to the `getTodo` and `searchTodo` functions respectively, whose bodies are defined in the remainder of the object literal ❷.

As explained earlier, once one of the paths defined in the router matches the current URL, the relative function is executed, passing as its arguments the variable(s) defined in the path. Variables are the parts of the path you defined starting with the colon, like `:id`.

With this explanation, we've completed the analysis of the framework's components. It's time to get your hands dirty with the development of the Todos manager.

15.5.3 *Creating a Todos manager application using Backbone.js*

When learning a new framework, most people agree that one of the most effective ways to solidify the idea is to develop a small project. The goal of this section is to guide you through the creation of a simple Todos manager (figure 15.5) that allows you to perform the typical CRUD (Create Read Update Delete) operations. To keep the project as simple as possible, instead of using a web service to send and store

your data, you'll use the Web Storage API (http://www.w3.org/TR/webstorage/), relying on a Backbone.js adapter called Backbone .localStorage (https://github.com/jeromegn/ Backbone.localStorage). The complete code can be found in the source code provided with this book in the folder chapter-15/todos-manager.

Figure 15.5   Layout of the Todos manager application

Your Todos manager will have only one model that represents a Todo. Every Todo item will have a `title` property, where you'll save the task that has to be done, its position inside the list in a property called `position`, and a Boolean that specifies if the Todo has been completed or not in a property called `done`. You'll also employ one collection to help you keep the models sorted. Finally, differently from what you might expect, you'll have two views. You'll adopt the Element controller pattern that consists of two views: the first controls a collection of items, whereas instances of the second deal with each individual item.

The structure of the project, shown in figure 15.6, is quite straightforward. It has an index.html page that contains the HTML markup and the templates used by the application. It also has a css folder containing the basic CSS file that gives the application a better look and feel, a js folder containing all the libraries included (such as jQuery and Backbone.js) inside a subfolder named vendor, and a file called app.js containing the project's specific code. To keep things as easy as possible, you'll put all the code in the same file, but when dealing with large projects a better choice is to have a different file for every object stored in a subfolder named like the components you've seen so far: models, collections, and views.

Figure 15.6   Folder and file structure of the Todo application

Now that you've seen the features of the project, let's start developing it.

CREATING THE HTML

No web app that interacts with users can be developed without an interface, so the first step is to create the HTML markup. All of the HTML markup and the templates will reside in the index.html file. The interface is simple because you only need two components. The first is the place where the user can type the new activity to perform (the title of the Todo) and add it to the list, while the second is the list of Todos.

When developing a web application it's a good practice to provide feedback to the user in case of failure of any of the operations performed. For this reason, you'll also add a DOM element to show error messages in case they're needed.

The HTML implementing these needs is shown here:

```html
<div id="todo-sheet">
   <input id="new-todo" type="text" placeholder="Put your Todo here" />
   <button id="new-todo-save">Save</button>
   <span class="error-message"></span>
   <ul class="todos">
   </ul>
</div>
```

In this code you create the element (`ul`) where the Todos will be injected, but you haven't decided how you want to display them. You need to create the template for the `<li>`s containing the information of a Todo. How you do that depends on your design choices, but we'll still give you a suggestion. As we said in the introduction, the Todos have `title`, `position`, and `done` properties. The first two can be represented using a simple `span` element, whereas for the last one a check box would be a better choice because it enables users to check it and mark the Todo as completed.

> **TIP**   In the markup shown we omitted a `label` element associated with the `input` field to let you focus on the code related to the project. However, when dealing with form elements, it's always a good practice to provide such `label`s because they improve the accessibility of your elements.

The `<span>` showing the title of the Todo will be editable in place thanks to the `contenteditable` attribute. In addition, you'll allow users to delete the Todo using a `button` element having as its text a big *X*. To provide feedback to the user for completed Todos, you'll assign to the element a class that styles it as stroked.

Earlier, in the section titled "View," we pointed out that templates contain place-holders where you can show the model's values and other structures like conditional statements. The template for the Todo is shown below:

```html
<script type="text/template" id="todo-template">
   <span class="todo-position"><%- position %></span>.
   <input class="todo-done" type="checkbox"
          <%= done ? checked="checked" : '' %> title="Completed" />
   <span class="todo-title <%= done ? 'todo-stroked' : '' %>"
        contenteditable="true"><%- title %></span>
   <button class="todo-delete" title="Delete">X</button>
</script>
```

Apart from the placeholders, you use a condition to test if the Todo is completed.

With all the HTML code in place, you need to include the libraries that will allow you to kick off the application and your code.

INSTALLING BACKBONE.JS

Backbone.js has as its unique hard dependency a library called Underscore.js (version >= 1.7.0). This means that you must include the latter before Backbone.js; otherwise, the framework won't work. Including the framework and its dependency is as easy as including jQuery. All you have to do is add them to your page using the `<script>` tag.

The libraries your project will rely on are jQuery (this is still a book on jQuery, isn't it?), Backbone.js and its dependency Underscore.js, and the Backbone.localStorage adapter. To include them, you'll add the `<script>` tags having the reference to them at the end of the index.html page but before the closing `<body>` tag, as shown in the next listing.

Listing 15.3   Including libraries in a web page

```html
<!DOCTYPE html>
<html>
    <head>
        ...
    </head>
    <body>
        ...
        <script src="js/vendor/jquery-1.11.3.min.js"></script>
        <script src="js/vendor/underscore-min.js"></script>
        <script src="js/vendor/backbone-min.js"></script>
        <script src="js/vendor/backbone.localStorage-min.js"></script>
        <script src="js/app.js"></script>
    </body>
</html>
```

The code shows how simple it is to include the framework.

> **NOTE**   If you want to improve this demo project, you can download all the libraries with Bower and manage the order of inclusion with RequireJS.

For the sake of precision, the homepage of the framework specifies the following:

> [...] for RESTful persistence, history support via Backbone.Router and DOM manipulation with Backbone.View, include jQuery, and json2.js for older Internet Explorer support.

The markup looks good, but at the moment your application isn't able to do anything. Let's fix this by developing the model of your project.

THE TODO MODEL

The only model of the Todos manager is the one representing a single Todo, which is the activity to complete (figure 15.7). Each instance of this object has `title`, `position`, and `done` properties. Instead of adding them directly, as shown in the snippet in the section "Model," you'll wrap them with an object assigned to a property called `defaults`. By doing so, when an instance of the model is created, any unspecified property will be set to the respective default value. This approach isn't mandatory, but the advantage is that you ensure a default value for each property of the model.

In your model you'll also create two methods: `initialize()` and `validate()`. Both are optional

Figure 15.7   A representation of a Todo object, the only model of the Todos manager

but you'll find yourself using them often in your projects. The former, present also in collections and views, is executed whenever a new instance of a model is created. Here you'll add the listeners for one or more events so that a handler can be executed once an event is triggered.

The `validate()` method is, by default, called before storing the object. This method should return an error, which can be a string or an object, in case of failure and nothing on success. In case of failure, the model won't be updated on the storage used, whether it resides on a server or locally (as your project does). This method is important because when it returns an error, it also fires an event called `invalid` that you can listen to in order to perform one or more specific actions.

In your project, inside the `initialize()` method you'll listen for this and other events to log some information on the console. By doing so you can keep an eye on what's going on in your application.

Before delving into the code, you need to perform a simple preliminary step. Throughout the book you've learned how important it is to not pollute the global scope. All the models, views, and collections you'll develop will live inside a single namespace that you'll name `app`. Your first line of code will be this:

```
window.app = {};
```

With your namespace in place, take a look at the code of the Todo model shown in the following listing.

Listing 15.4   The Todo model

```
app.Todo = Backbone.Model.extend({          ⟵──────❶ Extends the Backbone.Model object
  defaults: {
    position: 1,
    title: '',            ❷ Defines the
    done: false              default values of
  },                         the properties

  initialize: function() {
    this
      .on('invalid', function(model, error) {     ⟵────── Attaches a handler to
        console.log(error);                    ❸ the invalid event
      })
      .on('add', function(model, error) {
        console.log(
          'Todo with title "' + model.get('title') + '" added.'
        );
      })
      .on('remove', function(model, error) {
        console.log(
          'Todo with title "' + model.get('title') + '" deleted.'
        );
      })
      .on('change', function(model, error) {
        console.log(
          'Todo with title "' + model.get('title') + '" updated.'
        );
```

```
            });
        },

        validate: function(attributes) {                    ←    ❹ Declares the
            if(!attributes.title) {                                 method to validate
                return 'Title is required and cannot be empty';     the properties
            }

            if(
                attributes.position === undefined ||
                parseInt(attributes.position, 10) < 1
            ) {
                return 'Position must be positive';
            }
        }
    });
```

At the beginning of the listing you create a new object by extending `Backbone`
`.Model` ❶. You assign the result to a property called `Todo`, which is stored as a prop-
erty of the `window.app` object you previously created (not shown in the listing). You
also define an object literal having as keys the names of the attributes you want to cre-
ate and as values the default values of the properties ❷.

   As mentioned before, you also create the `initialize()` method, where you add the
handlers for several events such as `invalid` ❸. Finally, you override the `validate()`
method ❹, where you check that the `title` isn't *falsy* (empty string, `null`, `undefined`,
and so on), make sure the `position` is greater than zero, and return an error message
if needed.

   In this section you built the model to represent the Todos for your application, but
you want to group them into a collection. Let's see how.

THE TODOS COLLECTION

You want to group the models together to represent them as a unique list. To do so
you'll employ a collection that's created by extending the `Backbone.Collection`
object and specify its type by setting the value for the `model` property. You also have to
force the list to be an ordered list because you want it to be sorted by the position spec-
ified in each Todo. Therefore, you need to define a comparator to sort the models by
setting the `comparator` property inside the collection. The latter can be a method
defined by the developer as well as a string that specifies the name of the attribute to
use to sort objects. As we said, you want to sort the Todos based on the `position` attri-
bute; hence you'll specify `position` as the value of the `comparator` property.

   When a new Todo is added or deleted by the user, you want to keep your list cor-
rectly sorted and with sequential position numbers. For this reason, you need to listen
for the `add` and `remove` events to execute a function that you'll call `collection-`
`Changed`, which is responsible for restoring the correct numbering sequence. Back-
bone.js passes the model added or deleted to the handler as an argument. This is
important because you'll test if it's valid or not, using the `isValid()` method, and only
if the test is passed will the other model's position be updated. The code that imple-
ments the Todo collection is shown in the next listing.

Listing 15.5   The collection of Todos

```
app.todoList = new (Backbone.Collection.extend({
    model: app.Todo,

    localStorage: new Backbone.LocalStorage('todo-list'),

    comparator: 'position',

    initialize: function() {
        this.on('add remove', this.collectionChanged);
    },

    collectionChanged: function(todo) {
        if (todo.isValid()) {
            this.each(function(element, index) {
                element.save({
                    position: index + 1
                });
            });
            this.sort();
        }
    }
}));
```

Specifies that this is a collection of Todo models

Defines the (local) storage where the Todos are stored

Elements are updated only if the model is valid

With this section we've covered the objects used to store the data and group the Todos, but nothing can be presented to the user yet. It's time to fill the gap by developing the views.

THE TODO VIEWS

The Todos manager has two views, one that deals with each individual Todo and another that deals with the collection of Todos. In this section we'll discuss the former; the latter is covered in the next section.

The Todos (as a group) are represented as a list, whereas a single Todo is created as an item of the list. In terms of HTML this means you'll have a ul element and many li elements. Inside the s you'll display the data associated with the model: title, position, and done. To create the li element, you'll set the tagName property of the view and, although not mandatory, the className property to easily associate a style to the element. To display the data of the model, you'll use the template described in the "Creating the HTML" section and override the render() method.

This view is also responsible for reacting to events of interest for a single Todo, like the deletion or addition of a single activity (a Todo). To achieve this goal, you'll use the Backbone events hash. It's nothing but an object, assigned to the events property of the view, made up of key-value pairs. A key is in the form of "eventName selector" and a value is the name of a callback function to execute.

The code implementing this view is shown in the listing that follows.

Listing 15.6   The Todo view

```
app.TodoView = Backbone.View.extend({
   tagName: 'li',
   className: 'todo',

   template: _.template($('#todo-template').html()),   ◁─── Caches the template

   events: {
      'blur .todo-position': 'updateTodo',
      'change .todo-done': 'updateTodo',            ❶ The events hash:
      'keypress .todo-title': 'updateOnEnter',         associate events
      'click .todo-delete': 'deleteTodo'               with callbacks
   },

   initialize: function() {
      this.listenTo(this.model, 'change', this.render);
      this.listenTo(this.model, 'destroy', this.remove);
   },

   deleteTodo: function() {                    ❷ Deletes the model
      this.model.destroy();                       from the storage
   },

   updateTodo: function() {
      this.model.save({                  ◁─── Saves the model
         title: $.trim(this.$title.text()),
         position: parseInt(this.$position.text(), 10),
         done: this.$done.is(':checked')
      });                                    Updates the model
   },                                        when the Enter key
                                             is pressed while
   updateOnEnter: function(event) {          editing a Todo
      if (event.which === 13) {       ◁─────┘
         this.updateTodo();
      }
   },                                           ❸ Renders the
                                                   compiled
   render: function() {                            template
      this.$el.html(this.template(this.model.toJSON()));  ◁──┘
      this.$title = this.$('.todo-title');
      this.$position = this.$('.todo-position');
      this.$done = this.$('.todo-done');

      return this;
   }
});
```

In the first lines of the listing you define this view's tag element, define a class name for the element, and cache its template. Then you create the events hash ❶ associating some events to a set of callbacks you'll create in the view. For example, you want to know when a user clicks the Delete button to delete the model from the list ❷. Finally, in the render() function you display the previously compiled template ❸ and return the HTML snippet that replaces the content of the view's element.

Let's now discuss the second view of the application.

THE APPLICATION VIEW

The application view, called `appView`, is responsible for the creation of new Todos and the display of the Todo list.

Unlike the Todo view, in your HTML you already have a DOM element this view can refer to, so you won't set the `tagName` and the `className`. The element is the `<div>` having `todo-sheet` as its ID. You'll set it as the value of the `el` property of the `appView`. When the view is initialized, you also want to fetch the list of Todos stored so that you can show them to the user. For this reason you'll call the `fetch()` method on the list in the `initialize()` method.

Based on its responsibilities, the only DOM event you're interested in for this view is the click on the Save button. Once it's fired, you can execute the function `create-Todo()` to create and store the new Todo written by the user. This situation is a perfect fit for the `events` hash. In addition to this DOM event, this view needs to listen for changes in the Todos list in order to update the HTML representing the list. The next listing shows the code that implements what we've discussed in this section.

> **Listing 15.7   The application view**

```
app.appView = Backbone.View.extend({
   el: '#todo-sheet',

   events: {
      'click #new-todo-save': 'createTodo'
   },

   initialize: function() {
      this.$input = this.$('#new-todo');          ❶ Caches the list and
      this.$list = this.$('ul.todos');               the input element

      this.listenTo(app.todoList, 'reset sort destroy', this.showTodos);
      this.listenTo(app.todoList, 'invalid', this.showError);

      app.todoList.fetch();            ◁———— ❷ Fetches models from storage
   },

   createTodo: function() {
      app.todoList.create(
         {
            title: this.$input.val().trim()      ❸ Creates the new Todo
         },
         {
            at: 0,                          ❹ Places the Todo at the top of the
            validate: true                     list and forces its validation
         }
      );

      this.$input.val('');
   },
                                               ❺ Displays the error
   showError: function(collection, error, model) {  ◁———    to the user
      this
         .$('.error-message')
         .finish()
```

```
        .html(error)
        .fadeIn('slow')
        .delay(2000)
        .fadeOut('slow');
    },

    showTodo: function(todo) {                              ⑥ Adds the item only if
        if (todo.isValid()) {            ⟵                    the model is valid
            var view = new app.TodoView({ model: todo });
            this.$list.prepend(view.render().el);
        }
    },

    showTodos: function() {
        this.$list.empty();
        var todos = app.todoList.sortBy(function(element) {
            return -1 * parseInt(element.get('position'), 10);
        });
        for(var i = 0; i < todos.length; i++) {
            this.showTodo(todos[i]);
        }
    }
});
```

In the `initialize()` method you cache the `ul` and the `input` element where the user can write the activity to perform (the title of the Todo) ❶. Then you attach a handler for the events of interest. Finally, you fetch the models from local storage ❷.

In the `createTodo()` method you create the instance of a Todo model, passing only the title and relying on the default values for the other properties ❸. Then you put it in your list of Todos, placing it at the beginning of the list using the `at` option and forcing its validation via the `validate` option ❹.

In case an error occurs, a message is shown inside the element having the class `error-message` by executing the `showError()` method ❺.

To render the list of Todos you create a `showTodos()` method and a support method, `showTodo()`, that's responsible for rendering a single Todo. Inside `show-Todos()`, you first ensure there's nothing inside the `ul` element that contains the Todos, using jQuery's `empty()` method. Then you sort the list in reverse order because you want the last stored element, the head of the list, to be shown as the first list item. The last Todo should be displayed at the top of the list, shouldn't it?

Finally, you loop over the reverse-sorted list, calling the `showTodo()` method and passing as an argument the current Todo. The `showTodo()` method tests if the given Todo is valid ❻ and, in case of success, a view associated with it is created and prepended to the `ul`.

At this point all the code is in place and you need to kick off your application. This is done by writing at the end of the file the following statement:

```
new app.appView();
```

With this last statement, the project is completed and you're allowed to celebrate this event by drinking champagne. The final and complete code can be found in the

source provided with this book in the folder chapter-15/todos-manager. To execute the application, open the index.html file in your browser.

This section dedicated to Backbone.js is just an introduction to the framework, and the application you built is very basic. But you should have noticed how closely jQuery was integrated within it and how jQuery was used extensively inside the functions that Backbone.js allowed you to construct. This is a true testament to the widespread usefulness and flexibility of jQuery. We hope that thanks to this introduction you're more aware of the potentiality of Backbone.js and intrigued enough to go further. As a final challenge to test your knowledge, we invite you to modify the project to employ Bower, RequireJS, and QUnit. Have fun!

15.6   *Summary*

In the first part of this chapter you saw how to improve the performance of code that uses jQuery by selecting elements the right way. We discussed when to take advantage of the `context` parameter of `jQuery()` and when to avoid its use. We also covered how to avoid using the Universal selector. Later you learned how to achieve better performance in older browsers by creating selectors that allow jQuery to call JavaScript native functions like `getElementById()` and `getElementsByClassName()`.

No library or framework is magic. Remember that when you use third-party software, even a powerful one like jQuery, it makes some optimizations on your behalf, but others are your responsibility.

In the second main section we introduced you to the importance of keeping your code base clean and organized. We taught you what a module is and some of the patterns available to split your code, written using jQuery (but not limited to this case only), in modules. Among the advantages of this approach is that you have the possibility of creating "private" variables and functions and avoiding polluting the global scope.

A tool we presented is RequireJS, a JavaScript file and module loader for different environments. This library frees you from the burden of manually sorting modules, libraries, and frameworks based on their dependencies. In its section, we covered how to develop modules to take advantage of RequireJS and showed how you can adapt existing JavaScript code to work with it. Specifically, we showed you how to use an existing jQuery plugin with RequireJS without changing its source through the use of a simple configuration file.

Another tool we described is Bower. It's a package manager for the web that empowers you to manage your JavaScript, CSS, and other types of dependencies including jQuery. You saw how you can search a package your application may need through the use of the CLI. Then you also learned how to install, update, and delete a package using Bower.

In the last part of this chapter you learned about Backbone.js, one of the MV* frameworks available in the JavaScript world. Backbone.js enables you to create single-page applications (SPAs), a type of application widely employed today that helps

developers to reduce the backend of complex applications to a minimum. In fact, most of the business logic is written in JavaScript and resides on the client side. The section dedicated to Backbone.js showed you what a possible next step is now that you know all about jQuery. In addition, this framework integrates well with jQuery to enable you to develop amazing applications.

We also discussed the architecture and main concepts of Backbone.js: models, routers, views, templates, and collections. Then we put it all together to develop a basic application to keep your daily activities organized, called Todos manager.

15.7 *The end*

Oh my! How much time has passed since the start of this book! It has been an incredible experience for us to offer you the best resource possible, and we really hope to have achieved our goal. We're sure that it has been an incredible journey for you as well, and that you've had some moments of discouragement in the attempt to remember the huge amount of information we've provided. If you can't recall every function we explained or the list of arguments accepted by a function, don't worry: there's nothing wrong with that. Experience, practice, and a tab of your browser constantly pointing to the jQuery documentation can solve this issue.

jQuery is a constantly evolving project, and it's subject to numerous updates, additions, deprecation, and even feature deletions as you've discovered by reading the changes brought by jQuery 3. Sometimes it's hard to catch up with all the news, the updates introduced by the team in every release of the library and the documentation, and the bugs found.

In this book we tried hard to offer you the most up-to-date information pertaining to the functions and the properties offered by jQuery, the best practices adopted by the web community, and also some advanced programming techniques.

We hope you've enjoyed this book and that you won't stop learning. We also 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
- What's an IIFE?
- Determining (and controlling) what `this` means
- What's a closure?

One of the great benefits that jQuery brings to your web applications is the ability to implement a great deal of scripting-enabled behavior without having to write a whole lot of script yourself. jQuery handles the nuts-and-bolts details so that you can concentrate on the job of making your applications do what they need to do!

For the first few chapters in this book, you needed only rudimentary JavaScript skills to code and understand the examples provided. In the later chapters, such as those on event handling, animations, and Ajax, you had to understand a handful of fundamental JavaScript concepts to make effective use of the jQuery library. You may have found that a lot of things that you perhaps took for granted in JavaScript (or took on blind faith) started to make more sense.

We're not going to go into an exhaustive study of all JavaScript concepts here—that's not the purpose of this book. The purpose of this appendix is to give you the fundamental JavaScript concepts you need to make the most effective use of jQuery.

The most important of these concepts is that functions are first-class objects in JavaScript, which is a result of the way JavaScript defines and deals with functions. 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 you understand what a JavaScript object is all about. Let's dive right in.

1   *JavaScript Object fundamentals*

The majority of object-oriented (OO) languages define a fundamental `Object` data type of some kind from which all other objects are derived. 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 most other OO languages.

Once created, a JavaScript `Object` 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.

1.1   *How objects come to be*

A new object in JavaScript can be created in several ways. The first method we want to introduce is via the `new` operator paired with the `Object` constructor. Creating an object this way is as easy as this:

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

It could be even easier (as you'll see shortly), but this will do for now.

This new object contains nothing: no information, no complex semantics, nothing. It doesn't get interesting until you start adding things to it—things known as *properties*.

1.2   *Properties of objects*

JavaScript objects can contain data and possess methods (well, sort of) that you can create dynamically as needed. Take a look at the following code fragment:

```
var ride = new Object();
ride.make = 'Yamaha';
ride.model = 'XT660R';
ride.year = 2014;
ride.purchased = new Date(2015, 4, 10);
```

Here you create a new `Object` instance and assign it to a variable named `ride`. You then populate this variable with a number of *properties* of different data types: two strings, a number, and an instance of an instance of `Date`.

> **NOTE**   In `Date` months start with 0. January corresponds to 0, February to 1, March to 2, and so on.

You don't need to declare these properties prior to assigning them; they come into being merely by the act of your assigning a value to them. That's mighty powerful and it gives you a great deal of flexibility. But flexibility always comes at a price!

For example, let's say that in subsequent code on your scripted HTML page you want to change the value of the purchase date:

```
ride.purchased = new Date(2015, 7, 21);
```

No problem … unless you make an inadvertent typo such as

```
ride.purcahsed = new Date(2015, 7, 21);
```

There's no compiler to warn you that you've made a mistake; a new property named `purcahsed` is cheerfully created on your behalf, leaving you to wonder later why the new date didn't *take* when you reference the correctly spelled property.

With great power comes great responsibility (where have you heard that before?), so type carefully!

From this example, you've learned that an instance of the JavaScript `Object`, which we'll refer to as an *object* from here forward, is a collection of *properties*. Each of these properties consists of a *name* and a *value*. The name of a property is a string, and the value can be any JavaScript data type: `Number`, `String`, `Boolean`, `Object`, and so on. This means the primary purpose of an `Object` instance is to serve as a container for a named collection of other types.

An object property can be another `Object` instance, which in turn has its own set of properties, which can in turn be objects with their own properties, and so on, to any depth that makes sense for the data that you're trying to model.

Let's say that you add a new property to your `ride` instance that identifies 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, you write the following:

```
var ownerName = ride.owner.name;
```

There are no limits to the nesting levels you can employ (except the limits of good sense). When finished—up to this point—the object hierarchy is as shown in figure 1.

Note how each value in the figure is a distinct instance of a JavaScript type.

> **NOTE** There's no need for all the intermediary variables (such as `owner`) that we created for illustrative purposes in these code fragments. In a short while, you'll see more efficient and compact ways to declare objects and their properties.

Figure 1 **The object hierarchy shows that `Objects` are containers for named references to other `Objects` or JavaScript built-in types.**

Up to this point, you've referenced properties of an object by using the dot (period character) operator. Now ponder this: what will happen if you have a property named `color.scheme` (note the period in the middle of the name)? In this case, 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 part of a name? And most importantly, what if you don't even know what the property name is, but you 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 you must use the more general *square brackets operator* for accessing properties

```
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']
```

So is this reference:

```
var p = 'make';
ride[p];
```

Using the square brackets operator is the only way to reference properties whose names don't form valid JavaScript identifiers, such as this,

```
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 objects and their properties.

1.3   *Object literals*

In the previous section, you created an object that modeled some of the properties of a motorcycle, assigning it to a variable named `ride`. To do so, you used two `new` operations, an intermediary variable named `owner`, and a bunch of assignment statements. This is tedious and error-prone. In addition, it's difficult to visually grasp the structure of the object during a quick inspection of the code.

Luckily, you can use a notation that's more compact and easier to visually scan. Consider the following statement:

```
var ride = {
  make: 'Yamaha',
  model: 'XT660R',
  year: 2014,
  purchased: new Date(2015, 7, 21),
  owner: {
    name: 'Spike Spiegel',
    occupation: 'bounty hunter'
  }
};
```

Using an *object literal*, this fragment creates the same `ride` object that you built up with assignment statements in the previous section but in a unique, compact statement. This notation is preferred by most page authors.

The structure is pretty 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. As you can see by the declaration of the `owner` property, object declarations can be nested.

You can also express arrays using the *array literal* notation, which consists of a comma-delimited list of elements within square brackets, as in the following:

```
var someValues = [2, 3, 5, 7, 11, 13, 17];
```

In the examples presented in this section, object references are frequently stored in variables or in properties of other objects. Let's take a look at a special case of the latter scenario.

1.4   *Objects as window properties*

Up to this point, you'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 = 'This is a text.';
someObject.aProperty = 'This is another text.';
```

These two statements each assign a string to something: a variable in the first statement and an object property in the second. 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 predefined JavaScript `window` object. Any reference created in top-level scope is implicitly made on the `window` instance. This means that all of the following statements, if made at the top level (that is, outside the scope of a function), are equivalent:

```
var foo = 'bar';
window.foo = 'bar';
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`. This concept might not seem hard to understand, but the scoping rules get more complex when you delve deeper into the bodies of functions.

That pretty much covers things for our overview of the JavaScript `Object`. These are the important concepts to take away from this discussion:

- A JavaScript object is an unordered collection of properties.
- Properties consist of a name and a value.
- Objects can be declared using object literals.
- Arrays can be declared using array 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*.

2   *Functions as first-class citizens*

In many traditional OO languages, objects can contain data and possess methods. In these languages, the data and methods are usually distinct concepts; JavaScript walks a different path.

Functions in JavaScript are considered objects like any other object type that's defined in JavaScript such as `Strings`, `Numbers`, or `Dates`. Like other objects, functions are defined by a JavaScript constructor—in this case `Function`—and they 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*.

In JavaScript, functions can serve different purposes and can be defined in different ways. Let's discover more.

2.1 *Function expressions and function declarations*

Although it might seems odd, functions are nothing but values that can be called, and we'll prove that this is true shortly. One way of defining a function is called a *function declaration*. Consider the following code:

```
function doSomethingWonderful() {
  alert('Does something wonderful');
}
```

A function declaration is composed of the keyword `function`, followed by the name of the function, followed by its parameters list enclosed in parentheses, followed by the function body. In the previous snippet, you define a function whose name is `doSomethingWonderful` that has no parameters. When invoked, it executes its body, which in this case is made of a single call to `alert()`. It may seem that the function doesn't return a value. But in JavaScript if an explicit value isn't returned, by default a function returns `undefined`.

Browsers and function names

Browsers that have implemented partly or in full the specifications of ECMAScript 6 expose a property of functions called `name` that stores the name of the function. You can find out more about this property at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/name.

A few moments ago we stated that variables defined at the top level create properties of the `window` object. `Function` objects are no different. If the previous function declaration is declared at the top level, you'll obtain the creation of a `window` property with the same name as the function name. Therefore, the following statements are all equivalent:

```
function hello() { alert('Hi there!'); }
hello = function hello() { alert('Hi there!'); }
window.hello = function hello() { alert('Hi there!'); }
```

In browsers that have implemented partly or in full the specifications of ECMAScript 6 you can access the name of the function through a property of the function itself called `name`.

In JavaScript, functions can be defined as a part of a statement and are therefore named *function expressions*. A function expression produces as its value a function object. Consider the following code:

```
var myFunc = function() {
  alert('this is a function');
};
```

As you can see, you define a variable called `myFunc` to which you assign a function. Because this is a statement, note the semicolon at the end of the statement, after the closing curly bracket. The function doesn't have a name (thus its `name` property will be an empty string), so you can't invoke it using its name. But because you've assigned it to a variable, you can execute the function as follows:

```
myFunc();
```

This isn't the only difference between a function declaration and a function expression. Another important difference is that function declarations are *hoisted* but function expressions are not. To understand what this means in practice, consider the following example:

```
funcDecl();                    <——— The message is alerted correctly.
funcExpr();

function funcDecl() {                    ⟵
    alert('function declaration');            ❶ Creates a function via
}                                               a function declaration

var funcExpr = function() {              ⟵
    alert('function expression');             ❷ Creates a function via
};                                              a function expression
```

Error! ⟶ appears to the left of `funcExpr();`

In this example you define two functions: `funcDecl()` ❶ and `funcExpr()` ❷. But before they're actually defined, you try to execute them. The first call (`funcDecl();`) succeeds, but the second (`funcExpr();`) raises an error. The different behavior is caused by the fact that `funcDecl()` is hoisted but `funcExpr()` isn't.

In the same way you can assign a function expression to a variable, you can assign it to a property of an object:

```
var myObj = {
    bar: function() {}
};
```

You'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 you do that.

2.2   Functions as callbacks

When dealing with events or timers, or when you're performing Ajax requests, the nature of the code in a web page is asynchronous. One of the most prevalent concepts in asynchronous programming is the notion of a *callback* function.

Let's take the example of a timer. You 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 you know when the timer has expired so that you can do whatever it is that you're waiting around for? It does so by invoking a function that you supply.

Consider the following code:

```
function hello() { alert('Hi there!'); }
setTimeout(hello, 5000);
```

You 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, you pass a function reference. Passing a function as a parameter is no different than passing any other value, just as you passed a number.

When the timer expires, the `hello` function is called. Because the `setTimeout()` method makes a call *back* to a function in your own code, that function is termed a *callback* function.

This code example would be considered naive by most advanced JavaScript coders because the creation of the `hello` name is unnecessary as you're using it only once. 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 you express the function directly in the parameter list (as an *inline anonymous function*), and no needless name is generated. You'll often see this idiom used in jQuery code when there's no need for a function instance to be assigned to a top-level property.

The functions you've created in the examples so far are either top-level functions (which you know are top-level `window` properties) or assigned to parameters in a function call. You can also assign `Function` instances to properties of objects. Let's see how.

2.3   *What this is 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 the JavaScript implementation of `this` differs from its OO counterparts.

In class-based OO languages, `this` generally references the instance of the class for 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 the motorcycle example for a demonstration, amending the object creation as follows (additions are highlighted in bold):

```
var ride = {
  make: 'Yamaha',
  model: 'XT660R',
  year: 2014,
  purchased: new Date(2015, 7, 21),
  owner: {
    name: 'Spike Spiegel',
    occupation: 'bounty hunter'
  },
  whatAmI: function() {
    return this.year + ' ' + this.make + ' ' + this.model;
  }
};
```

To your original example code, add a property named `whatAmI` that references a `Function` instance. The new object hierarchy, with the `Function` instance assigned to the property named `whatAmI`, is shown in figure 2.

When the function is invoked through the property reference, like this,

```
var bike = ride.whatAmI();
```

the function context (`this`) is set to the object instance pointed to by `ride`. As a result, the variable `bike` gets set to the string `'2014 Yamaha XT660R'` because the function picks up the properties of the object through which it was invoked via `this`.

Figure 2   This model clearly shows that the function isn't part of the `Object` but is only referenced from the `Object` property named `whatAmI`.

The same is true of top-level functions. Remember that top-level functions are properties of `window`, so their function context, when called as top-level functions, is the `window` object.

Although that may be the usual and implicit behavior, JavaScript gives you the means to explicitly control what's used as the function context. You can set the function context to whatever you want by invoking a function via the `Function` methods `call()` or `apply()`. Although it seems crazy, 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, whereas 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.

To reinforce the concept, let's see an example. Consider the code of the following listing (available in the downloadable code as appendix-a/function.context.html and as a JS Bin at http://jsbin.com/dumac/edit?html,js,output).

Listing 1 Function context value depends on how the function is invoked

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Function Context Example</title>
   </head>
   <body>
      <script>
         var obj1 = { handle: 'obj1' };
         var obj2 = { handle: 'obj2' };
         var obj3 = { handle: 'obj3' };
         var value = 'test';
         window.handle = 'window';

         function whoAmI(param) {
            return this.handle + ' ' + param;
         }

         obj1.identifyMe = whoAmI;

         alert(whoAmI(value));
         alert(obj1.identifyMe(value));
         alert(whoAmI.call(obj2, value));
         alert(whoAmI.apply(obj3, [value]));
      </script>
   </body>
</html>
```

① Defines three object literals with the same property name but different values

② Defines a function with one parameter

③ Assigns the function to a property of the obl object

④ Invokes the whoAmI() function

⑤ Invokes the whoAmI() function using the reference stored in objl.indentifyMe

⑥ Invokes the whoAmI() function using call()

⑦ Invokes the whoAmI() function using apply()

In the code, you define three simple objects, each with a `handle` property that makes it easy to identify the object given a reference ❶. You also add a `handle` property to the `window` instance so that it's also easily identifiable.

You then define a top-level function that returns the value of the `handle` property for whatever object serves as its function context ❷ and assign the *same* function instance to a property of object `obj1` named `identifyMe` ❸. You can say that this creates a method on `obj1` named `identifyMe`, although it's important to note that the function is declared independently of the object.

Finally, you 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 3.

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 ❹.
- When called as a property of an object (`obj1` in this case), the object becomes the function context of the function invocation ❺. You 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.

Figure 3   The object serving as the function context changes with the manner in which the function is called.

- 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, `obj2` ❻. In this example, the function acts like a method to `obj2`, even though it has no association whatsoever—even as a property—with `obj2`. It also shows how parameters can be passed when using `call()`.
- As with `call()`, using the `apply()` method of `Function` sets the function context to whatever object is passed as the first parameter ❼. The difference between these two methods becomes significant only when parameters are passed to the function. In fact, when using `apply()` all the parameters must be provided as elements of a single array passed as the second argument.

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. As a result, it's 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 `func` acts as a method of object `obj` when `obj` serves as the function context of the invocation of `func`.

As a further illustration of this concept, consider the effect of adding the following statement to your example:

```
alert(obj1.identifyMe.call(obj3));
```

Even though you reference the function as a property of `obj1`, the function context for this invocation is `obj3`, further emphasizing that it's not how a function is declared but how it's invoked that determines its function context.

Now that you understand how functions can act as methods of objects, turn your attention to another advanced function topic that will play an important role in effective usage of jQuery: closures.

2.4   *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. This is expected and should be no surprise to any developer from any background. But with closures, 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, let's look at the illustrative example in the next listing. The code is available in the file appendix-a/closure.html and also as a JS Bin at http://jsbin.com/jurub/1/edit?html ,js,output.

Listing 2   Closures allow access to the scope of a function's declaration

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Closure Example</title>
   </head>                                            ❶ Defines the element where
   <body>                                                the current time is written
      <div id="display"></div>           ◁

      <script src="../js/jquery-1.11.1.min.js"></script>
      <script>
         function timer() {
            var local = 1;                    ◁
            window.setInterval(                         ❷ Initializes the local variable to 1
               function() {
                  $('#display').append(
                     '<div>At ' + new Date() + ' local=' + local + '</div>'
```

Defines a function that is executed every two seconds ❸

```
            );
            local++;                          Increments the local
          },                           4      variable by 1
          2000
        );
    }

      timer();                                Executes the timer()
    </script>                          5      function
  </body>
</html>
```

In this example you create a function called `timer()` that's executed after it has been defined **5**. In the `timer()` function you declare a local variable named `local` **2** and assign it a numeric value of 1. You then use the `window.setInterval()` method to establish a timer that will fire every two seconds **3**. As the callback for the timer, you specify an inline function that references the `local` variable and shows the current time and the value of `local` by appending a `div` element into an element named `display` that's defined in the page body **1**. As part of the callback, the `local` variable's value is also incremented by 1 **4**.

If you haven't had any experience with closures, you might think that because the callback will fire off two seconds after the `timer()` function has been invoked, the value of `local` is undefined during the execution of the callback. But upon loading the page and letting it run for a short time, you'll see the display shown in figure 4.

```
At Sun Mar 15 2015 01:12:21 GMT+0000 (GMT Standard Time) local=1
At Sun Mar 15 2015 01:12:23 GMT+0000 (GMT Standard Time) local=2
At Sun Mar 15 2015 01:12:25 GMT+0000 (GMT Standard Time) local=3
At Sun Mar 15 2015 01:12:27 GMT+0000 (GMT Standard Time) local=4
At Sun Mar 15 2015 01:12:29 GMT+0000 (GMT Standard Time) local=5
```

Figure 4   Closures allow callbacks to access their environment even if that environment has gone out of scope.

This example works because 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 it 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.

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 you might expect:

```
...
this.id = 'someID';
$('*').each(function() {
   alert(this.id);
});
```

Remember that each function invocation has its own function context so that, in the preceding code, the function context within the callback function passed to `each()` is an element from the jQuery collection (that is an element of the DOM), 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 jQuery collection in turn.

When access to the object serving as the function context in the outer function is needed, you 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 the example:

```
this.id = 'someID';
var outer = this;
$('*').each(function() {
   alert(outer.id);
});
```

The variable `outer` (most times named `that`) becomes part of the closure because it has been referenced inside the callback function and, as such, can be accessed in it. The variable `outer` is assigned a reference to whatever is the context outside of the callback function defined. For example, if the previous code is wrapped inside a function named `foo`, the variable `outer` will be a reference to the `foo`'s function context. If the previous code was defined in an HTML page without being wrapped by a function, the variable `outer` will be a reference to the `window` object.

The changed code now displays an alert showing the string `'someID'` as many times as there are elements in the jQuery collection.

You'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.

Before concluding this appendix, we want to discuss with you one last concept that we've used extensively throughout this book: Immediately-Invoked Function Expression (IIFE).

2.5 *Immediately-Invoked Function Expression*

An Immediately-Invoked Function Expression is a JavaScript design pattern that creates a function expression and then immediately executes the function. This term was coined by Ben Alman in his article "Immediately-Invoked Function Expression (IIFE)" (http://benalman.com/news/2010/11/immediately-invoked-function-expression/).

Before discussing what the benefits of using an IIFE are, let's see how you can implement it:

```
(function() {
   // The code of the function goes here...
})();
```

In this code, you create an anonymous function that's immediately executed thanks to the `()` at the end. The function is wrapped inside two parentheses because you need to tell the parser to expect a function expression and not a function declaration.

This pattern is useful in several situations. You can use it to create "private" variables and functions so they won't be accessible outside the scope of the function. An additional benefit is that because you can create "private" variables and functions, you avoid polluting the global namespace.

When employing an IIFE you can also pass arguments to it in the same way you can with other functions. For example, you could do something like this:

```
var i = 10;
(function(index) {
   // The code of the function goes here...
})(i);
```

In this example you declare a variable called `i` and then pass it to the IIFE. Inside the function you can perform any action needed that uses the value of `i` through the `index` parameter.

Immediately-Invoked Function Expressions are often used when you need to use variables of the outer scope inside an event handler. Consider the case where a page has the following three buttons:

```
<button id="button-1">Button 1</button>
<button id="button-2">Button 2</button>
<button id="button-3">Button 3</button>
```

What you want to do is to attach a handler to each of them, prompting their index (1 for `button-1`, 2 for `button-2`, and 3 for `button-3`). A possible implementation could be as follows:

```
for (var i = 1; i <= 3; i++) {
   document.getElementById('button-' + i).addEventListener(
      'click',
      function() { alert(i); }
   );
}
```

Unfortunately, this code doesn't work as expected. Regardless of the button clicked, the page will always prompt an alert showing the number 3. The reason is that at the time the callback is executed, the `for` loop is concluded and the value of the variable `i`, accessible via closure, is 3.

To solve this issue, you can employ an IIFE:

```
for (var i = 1; i <= 3; i++) {
    (function(index) {
        document.getElementById('button-' + index).addEventListener(
            'click',
            function() { alert(index); });
    })(i);
}
```

This code creates a new closure for each of the three iterations of the `for` loop. Therefore, each function retains its own value of the `index` parameter, which in turn is set passing the `i` variable.

As you can see, this pattern is extremely useful and we're sure you'll adopt it extensively in your code.

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 appendix, we introduced some aspects of the language that you must understand to use jQuery effectively.

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. You can declare them using a function declaration or a function expression, 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 `this` 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.

You also learned 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.

Finally, we discussed a JavaScript pattern called IIFE. It enables you to create "private" variables and functions and avoid polluting the global namespace. It's useful when dealing with an event's callback, where you may need to create a closure to ensure using the right value of a variable.

With these concepts firmly under your belt, you're ready to face the challenge of writing effective JavaScript using jQuery on your pages.

index

WEB DEVELOPMENT

jQuery IN ACTION Third Edition

Bear Bibeault • Yehuda Katz • Aurelio De Rosa

Thanks to jQuery, no one remembers the bad old days when programmers manually managed browser inconsistencies, CSS selectors support, and DOM navigation, and when every animation was a frustrating exercise in raw JavaScript. The elegant, intuitive jQuery library beautifully manages these concerns, and jQuery 3 adds even more features to make your life as a web developer smooth and productive.

jQuery in Action, Third Edition, is a fast-paced guide to jQuery, focused on the tasks you'll face in nearly any web dev project. In it, you'll learn how to traverse the DOM, handle events, perform animations, write jQuery plugins, perform Ajax requests, and even unit test your code. Its unique Lab Pages anchor each concept in real-world code. This expanded Third Edition adds new chapters that teach you how to interact with other tools and frameworks and build modern single-page web applications.

What's Inside

- Updated for jQuery 3
- DOM manipulation and event handling
- Animations and effects
- Advanced topics including Unit Testing and Promises
- Practical examples and labs

Readers are assumed to have only beginning-level JavaScript knowledge.

Bear Bibeault is coauthor of *Secrets of the JavaScript Ninja*, *Ajax in Practice*, and *Prototype and Scriptaculous in Action*. **Yehuda Katz** is an early contributor to jQuery and cocreator of Ember.js. **Aurelio De Rosa** is a full-stack web developer and a member of the jQuery content team.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/jquery-in-action-third-edition

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]

Free eBook
SEE INSERT

"Does a great job of showing how all the parts of jQuery fit together and demonstrates important concepts."
—From the Foreword by Dave Methvin, President jQuery Foundation

"The best-thought-out and researched piece of literature on the jQuery library."
—From the Foreword by John Resig, Creator of jQuery

"For three editions now, this is the only jQuery book I recommend to my clients, period."
—Christopher Haupt Mobirobo Inc.