sitepoint®

JQUERY
NOVICE TO NINJA

BY EARLE CASTLEDINE
& CRAIG SHARKIE

RICH, FAST, VERSATILE — JAVASCRIPT THE WAY IT SHOULD BE!

Thank You For Downloading This Book

Thank you for downloading the sample chapters of *jQuery: Novice to Ninja*, by Earle Castledine and Craig Sharkie, published by SitePoint.

If you're ready to fast-track your jQuery skills, this book is the perfect solution. All jQuery basics are covered, so you'll quickly learn how to unleash the power of this popular JavaScript framework.

After you've mastered the basics, you'll move progressively through to more advanced tips, tricks, and techniques that will wow even the most seasoned web designer or developer.

This sample includes:

- a summary of contents
- information about the author, editors, and SitePoint
- the Table of Contents
- the Preface
- Chapters 1 ("Falling in Love with jQuery"), 2 ("Selecting, Decorating, and Enhancing"), and 7 ("Forms, Controls, and Dialogs") from the book
- the book's Index

We can't wait to share all the valuable knowledge contained in the book, so enjoy these free chapters, and when you're ready to become a true jQuery Ninja, grab yourself a copy of the whole book.[1]

For more information, visit http://www.sitepoint.com/launch/25534b.

[1] https://sitepoint.com/bookstore/go/170/25534b

What's In This Excerpt?

Preface

Chapter 1: Falling in Love with jQuery

A brief overview of the advantages of using jQuery, and how to get it ready for use on your site

Chapter 2: Selecting, Decorating, and Enhancing

An introduction to jQuery's DOM selection and CSS capabilities

Chapter 7: Forms, Controls, and Dialogs

Learn how to manipulate and validate HTML forms with jQuery, as well as integrate more advanced interface controls and dialogs

Index

What's In the Rest of the Book?

Chapter 3: Animating, Scrolling, and Resizing

Learn the secrets of getting the most out of jQuery's advanced animation API

Chapter 4: Images, Slideshows, and Cross-Faders

All about building image galleries, lightboxes, and slideshows

Chapter 5: Menus, Tabs, Tooltips, and Panels

Make your web site into a desktop-like with UI widgets like dropdown menus, tabbed interfaces, and tooltips

Chapter 6: Construction, Ajax, and Interactivity

Harness the power of Ajax with ease thanks to jQuery

Chapter 8: Lists, Trees, and Tables

Enhance your site's lists and tables with some jQuery goodness

Chapter 9: Plugins, Themes, and Advanced Topics

Move on from the basics to some of jQuery's more advanced secrets, and learn to package your code into a plugin

JQUERY: NOVICE TO NINJA

BY EARLE CASTLEDINE
& CRAIG SHARKIE

jQuery: Novice to Ninja

by Earle Castledine and Craig Sharkie

Program Director: Andrew Tetlaw
Technical Editor: Louis Simoneau
Chief Technical Officer: Kevin Yank
Printing History:
First Edition: February 2010

Indexer: Fred Brown
Editor: Kelly Steele
Cover Design: Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: business@sitepoint.com

ISBN 978-0-9805768-5-6
Printed and bound in the United States of America

About Earle Castledine

Sporting a Masters in Information Technology and a lifetime of experience on the Web of Hard Knocks, Earle Castledine (aka Mr Speaker) holds an interest in everything computery. Raised in the wild by various 8-bit home computers, he settled in the Internet during the mid-nineties and has been living and working there ever since.

A Senior Systems Analyst and JavaScript flâneur, he is equally happy in the muddy pits of .NET code, the dense foliage of mobile apps and games, and the fluffy clouds of client-side interaction development.

As co-creator of the client-side opus TurnTubelist,[1] as well as countless web-based experiments, Earle recognizes the Internet not as a lubricant for social change but as a vehicle for unleashing frivolous ECMAScript gadgets and interesting time-wasting technologies.

About Craig Sharkie

A degree in Fine Art is a strange entrance to a career with a passion for programming, but that's where Craig started. A right-brain approach to code and problem solving has seen him plying his craft for many of the big names of the Web—AOL, Microsoft, Yahoo!, Ziff-Davis, and now Atlassian.

That passion, and a fondness for serial commas and the like, have led him on a path from journalism, through development, on to conferences, and now into print. Taking up JavaScript in 1995, he was an evangelist for the "good parts" before Crockford coined the term, and now has brought that keenness to jQuery.

About the Technical Editor

Louis Simoneau joined SitePoint in 2009, after traveling from his native Montréal to Calgary, Taipei, and finally Melbourne. He now gets to spend his days learning about cool web technologies, an activity that had previously been relegated to nights and weekends. He enjoys hip-hop, spicy food, and all things geeky.

About the Chief Technical Officer

As Chief Technical Officer for SitePoint, Kevin Yank keeps abreast of all that is new and exciting in web technology. Best known for his book, *Build Your Own Database Driven Web Site Using PHP & MySQL*, he also co-authored *Simply JavaScript* with Cameron Adams and

[1] http://www.turntubelist.com/

Everything You Know About CSS Is Wrong! with Rachel Andrew. In addition, Kevin hosts the *SitePoint Podcast* and co-writes the *SitePoint Tech Times*, a free email newsletter that goes out to over 240,000 subscribers worldwide.

Kevin lives in Melbourne, Australia and enjoys speaking at conferences, as well as visiting friends and family in Canada. He's also passionate about performing improvised comedy theater with Impro Melbourne (http://www.impromelbourne.com.au/) and flying light aircraft. Kevin's personal blog is *Yes, I'm Canadian* (http://yesimcanadian.com/).

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for Web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums.

Table of Contents

Preface

No matter what kind of ninja you are—a cooking ninja, a corporate lawyer ninja, or an actual *ninja* ninja—virtuosity lies in first mastering the basic tools of the trade. Once conquered, it's then up to the full-fledged ninja to apply that knowledge in creative and inventive ways.

In recent times, jQuery has proven itself to be a simple but powerful tool for taming and transforming web pages, bending even the most stubborn and aging browsers to our will. jQuery is a library with two principal purposes: manipulating elements on a web page, and helping out with Ajax requests. Sure, there are quite a few commands available to do this—but they're all consistent and easy to learn. Once you've chained together your first few actions, you'll be addicted to the jQuery building blocks, and your friends and family will wish you'd never discovered it!

On top of the core jQuery library is jQuery UI: a set of fine-looking controls and widgets (such as accordions, tabs, and dialogs), combined with a collection of full-featured behaviors for implementing controls of your own. jQuery UI lets you quickly throw together awesome interfaces with little effort, and serves as a great example of what you can achieve with a little jQuery know-how.

At its core, jQuery is a tool to help us improve the usability of our sites and create a better user experience. **Usability** refers to the study of the principles behind an object's *perceived efficiency or elegance*. Far from being merely flashy, trendy design, jQuery lets us speedily and enjoyably sculpt our pages in ways both subtle and extreme: from finessing a simple sliding panel to implementing a brand-new user interaction you invented in your sleep.

Becoming a ninja isn't about learning an API inside out and back to front—that's just called having a good memory. The real skill and value comes when you can apply your knowledge to making something exceptional: something that builds on the combined insights of the past to be even slightly better than anything anyone has done before. This is certainly not easy—but thanks to jQuery, it's fun just trying.

Who Should Read This Book

If you're a front-end web designer looking to add a dash of cool interactivity to your sites, and you've heard all the buzz about jQuery and want to find out what the fuss is about, this book will put you on the right track. If you've dabbled with JavaScript, but been frustrated by the complexity of many seemingly simple tasks, we'll show you how jQuery can help you. Even if you're familiar with the basics of jQuery, but you want to take your skills to the next level, you'll find a wealth of good coding advice and in-depth knowledge.

You should already have intermediate to advanced HTML and CSS skills, as jQuery uses CSS-style selectors to zero in on page elements. Some rudimentary programming knowledge will be helpful to have, as jQuery—despite its clever abstractions—is still based on JavaScript. That said, we've tried to explain any JavaScript concepts as we use them, so with a little willingness to learn you'll do fine.

What's in This Book

By the end of this book, you'll be able to take your static HTML and CSS web pages and bring them to life with a bit of jQuery magic. You'll learn how to select elements on the page, move them around, remove them entirely, add new ones with Ajax, animate them … in short, you'll be able to bend HTML and CSS to your will! We also cover the powerful functionality of the jQuery UI library.

This book comprises the following nine chapters. Read them in order from beginning to end to gain a complete understanding of the subject, or skip around if you only need a refresher on a particular topic.

Chapter 1: ***Falling in Love with jQuery***

Before we dive into learning all the ins and outs of jQuery, we'll have a quick look at why you'd want to use it in the first place: why it's better than writing your own JavaScript, and why it's better than the other JavaScript libraries out there. We'll brush up on some CSS concepts that are key to understanding jQuery, and briefly touch on the basic syntax required to call jQuery into action.

Chapter 2: ***Selecting, Decorating, and Enhancing***

Ostensibly, jQuery's most significant advantage over plain JavaScript is the ease with which it lets you select elements on the page to play with. We'll start off

this chapter by teaching you how to use jQuery's selectors to zero in on your target elements, and then we'll look at how you can use jQuery to alter those elements' CSS properties.

Chapter 3: ***Animating, Scrolling, and Resizing***

jQuery excels at animation: whether you'd like to gently slide open a menu, or send a dialog whizzing across the screen, jQuery can help you out. In this chapter, we'll explore jQuery's wide range of animation helpers, and put them into practice by enhancing a few simple user interface components. We'll also have a quick look at some animation-like helpers for scrolling the page and making elements resizable.

Chapter 4: ***Images, Slideshows, and Cross-fading***

With the basics well and truly under our belts, we'll turn to building some of the most common jQuery widgets out there: image galleries and slideshows. We'll learn how to build lightbox displays, scrolling thumbnail galleries, cross-fading galleries, and even take a stab at an iPhoto-style flip-book.

Chapter 5: ***Menus, Tabs, Tooltips, and Panels***

Now that we're comfortable with building cool UI widgets with jQuery, we'll dive into some slightly more sophisticated controls: drop-down and accordion-style menus, tabbed interfaces, tooltips, and various types of content panels. We're really on a roll now: our sites are looking less and less like the brochure-style pages of the nineties, and more and more like the Rich Internet Applications of the twenty-first century!

Chapter 6: ***Construction, Ajax, and Interactivity***

This is the one you've all been waiting for: Ajax! In order to make truly desktop-style applications on the Web, you need to be able to pass data back and forth to and from the server, without any of those pesky refreshes clearing your interface from the screen—and that's what Ajax is all about. jQuery includes a raft of convenient methods for handling Ajax requests in a simple, cross-browser manner, letting you leave work with a smile on your face. But before we get too carried away—our code is growing more complex, so we'd better take a look at some best practices for organizing it. All this and more, in Chapter 6.

Chapter 7: ***Forms, Controls, and Dialogs***

The bane of every designer, forms are nonetheless a pivotal cornerstone of any web application. In this chapter, we'll learn what jQuery has to offer us in terms of simplifying our form-related scripting. We'll learn how to validate forms on the fly, offer assistance to our users, and manipulate checkboxes, radio buttons, and select lists with ease. Then we'll have a look at some less conventional ways of allowing a site's users to interact with it: a variety of advanced controls like date pickers, sliders, and drag and drop. We'll round it off with a look at modal dialogs in the post-popup world, as well as a few original nonmodal notification styles. What a chapter!

Chapter 8: ***Lists, Trees, and Tables***

No matter how "Web 2.0" your application may be, chances are you'll still need to fall back on the everyday list, the humdrum tree, or even the oft-derided table to present information to your users. This chapter shows how jQuery can make even the boring stuff fun, as we'll learn how to turn lists into dynamic, sortable data, and transform tables into data grids with sophisticated functionality.

Chapter 9: ***Plugins, Themes, and Advanced Topics***

jQuery is more than just cool DOM manipulation, easy Ajax requests, and funky UI components. It has a wealth of functionality aimed at the more *ninja-level* developer: a fantastic plugin architecture, a highly extensible and flexible core, customizable events, and a whole lot more. In this chapter, we'll also cover the jQuery UI theme system, which lets you easily tailor the appearance of jQuery UI widgets to suit your site, and even make your own plugins skinnable with themes.

Where to Find Help

jQuery is under active development, so chances are good that, by the time you read this, some minor detail or other of these technologies will have changed from what's described in this book. Thankfully, SitePoint has a thriving community of JavaScript and jQuery developers ready and waiting to help you out if you run into trouble. We also maintain a list of known errata for this book, which you can consult for the latest updates; the details are below.

The SitePoint Forums

The SitePoint Forums[1] are discussion forums where you can ask questions about anything related to web development. You may, of course, answer questions too. That's how a discussion forum site works—some people ask, some people answer, and most people do a bit of both. Sharing your knowledge benefits others and strengthens the community. A lot of interesting and experienced web designers and developers hang out there. It's a good way to learn new stuff, have questions answered in a hurry, and have a blast.

The JavaScript Forum[2] is where you'll want to head to ask any questions about jQuery.

The Book's Web Site

Located at http://www.sitepoint.com/books/jquery1/, the web site that supports this book will give you access to the following facilities:

The Code Archive

As you progress through this book, you'll note a number of references to the code archive. This is a downloadable ZIP archive that contains each and every line of example source code that's printed in this book. If you want to cheat (or save yourself from carpal tunnel syndrome), go ahead and download the archive.[3]

Updates and Errata

No book is perfect, and we expect that watchful readers will be able to spot at least one or two mistakes before the end of this one. The Errata page[4] on the book's web site will always have the latest information about known typographical and code errors.

The SitePoint Newsletters

In addition to books like this one, SitePoint publishes free email newsletters, such as the *SitePoint Tech Times*, *SitePoint Tribune*, and *SitePoint Design View*, to name

[1] http://www.sitepoint.com/forums/
[2] http://www.sitepoint.com/forums/forumdisplay.php?f=15
[3] http://www.sitepoint.com/books/jquery1/code.php
[4] http://www.sitepoint.com/books/jquery1/errata.php

a few. In them, you'll read about the latest news, product releases, trends, tips, and techniques for all aspects of web development. Sign up to one or more SitePoint newsletters at http://www.sitepoint.com/newsletter/.

The SitePoint Podcast

Join the SitePoint Podcast team for news, interviews, opinion, and fresh thinking for web developers and designers. We discuss the latest web industry topics, present guest speakers, and interview some of the best minds in the industry. You can catch up on the latest and previous podcasts at http://www.sitepoint.com/podcast/, or subscribe via iTunes.

Your Feedback

If you're unable to find an answer through the forums, or if you wish to contact us for any other reason, the best place to write is `books@sitepoint.com`. We have a well-staffed email support system set up to track your inquiries, and if our support team members are unable to answer your question, they'll send it straight to us. Suggestions for improvements, as well as notices of any mistakes you may find, are especially welcome.

Acknowledgments

Earle Castledine

I'd like to thank the good folks at Agency Rainford for running Jelly (and getting me out of the house), Stuart Horton-Stephens for teaching me how to do Bézier Curves (and puppet shows), Andrew Tetlaw, Louis Simoneau, and Kelly Steele from SitePoint for turning pages of rambling nonsense into English, the Sydney web community (who do truly rock), the jQuery team (and related fellows) for being a JavaScript-fueled inspiration to us all, and finally, my awesome Mum and Dad for getting me a Spectravideo 318 instead of a Commodore 64—thus forcing me to read the manuals instead of playing games, all those years ago.

Craig Sharkie

Firstly, I'd like to thank Earle for bringing me onto the project and introducing me to the real SitePoint. I'd met some great SitePointers at Web Directions, but dealing

with them professionally has been a real eye-opener. I'd also like to thank my wonderful wife Jennifer for understanding when I typed into the wee small hours, and my parents for letting me read into the wee small hours when I was only wee small. Lastly, I'd like to thank the web community that have inspired me—some have inspired me to reach their standard, some have inspired me to help them reach a higher standard.

Conventions Used in This Book

You'll notice that we've used certain typographic and layout styles throughout the book to signify different types of information. Look out for the following items.

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code is to be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css *(excerpt)*

```
  border-top: 1px solid #333;
```

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

```
function animate() {
  new_variable = "Hello";
}
```

Also, where existing code is required for context, rather than repeat all the code, a vertical ellipsis will be displayed:

```
function animate() {
  ⋮
  return new_variable;
}
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➥ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/blogs/2007/05/28/user-style-she
➥ets-come-of-age/");
```

Tips, Notes, and Warnings

Hey, You!

Tips will give you helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

Chapter 1

Falling in Love with jQuery

So you have the coding chops to write lean, semantic HTML—and you can back it up with masterful CSS to transform your design ideas into gorgeous web sites that enthrall your visitors. But these days, you realize, inspiring designs and impeccable HTML alone fall short when you're trying to create the next Facebook or Twitter. So, what's the missing piece of the front-end puzzle?

It's JavaScript. That rascally scripting language, cast as the black sheep of the web development family for so many years. JavaScript is how you add complex behaviors, sophisticated interactions, and extra pizazz to your site. To conquer the sleeping giant that is JavaScript, you just need to buckle down and spend the next few years learning about programming languages: functions, classes, design patterns, prototypes, closures ...

Or there's a secret that some of the biggest names on the Web—like Google, Digg, WordPress, and Amazon—will probably be okay about us sharing with you: "Just use jQuery!" Designers and developers the world over are using the jQuery library to elegantly and rapidly implement their interaction ideas, completing the web development puzzle.

In the following chapter we'll have a look at what makes jQuery so good, and how it complements HTML and CSS in a more natural way than our old friend and bitter enemy: plain old JavaScript. We'll also look at what's required to get jQuery up and running, and working with our current sites.

What's so good about jQuery?

You've read that jQuery makes it easy to play with the DOM, add effects, and execute Ajax requests, but what makes it better than, say, writing your own library, or using one of the other (also excellent) JavaScript libraries out there?

First off, did we mention that jQuery makes it easy to play with the DOM, add effects, and execute Ajax requests? In fact, it makes it so easy that it's downright good, nerdy fun—and you'll often need to pull back from some craziness you just invented, put on your web designer hat, and exercise a little bit of restraint (ah, the cool things we could create if good taste were not a barrier!). But there are a multitude of notable factors you should consider if you're going to invest your valuable time in learning a JavaScript library.

Cross-browser Compatibility

Aside from being a joy to use, one of the biggest benefits of jQuery is that it handles a lot of infuriating cross-browser issues for you. Anyone who has written serious JavaScript in the past can attest that cross-browser inconsistencies will drive you mad. For example, a design that renders perfectly in Mozilla Firefox and Internet Explorer 8 just falls apart in Internet Explorer 7, or an interface component you've spent days handcrafting works beautifully in all major browsers except Opera on Linux. And the client just happens to use Opera on Linux. These types of issues are never easy to track down, and even harder to completely eradicate.

Even when cross-browser problems are relatively simple to handle, you always need to maintain a mental knowledge bank of them. When it's 11:00 p.m. the night before a major project launch, you can only hope you recall why there's a weird padding bug on a browser you forgot to test!

The jQuery team is keenly aware of cross-browser issues, and more importantly they understand *why* these issues occur. They have written this knowledge into the library—so jQuery works around the caveats for you. Most of the code you write

will run exactly the same on all the major browsers, including everybody's favorite little troublemaker: Internet Explorer 6.

This feature alone will save the average developer a lifetime of headaches. Of course, you should always aim to keep up to date with the latest developments and best practices in our industry—but leaving the task of hunting down obscure browser bugs to the jQuery Team (and they fix more and more with each new version) allows you more time to implement your ideas.

CSS3 Selectors

Making today's technologies cross-browser compliant is all well and good, but jQuery also fully supports the upcoming CSS3 selector specification. Yes, even in Internet Explorer 6.0! You can gain a head start on the future by learning and using CSS3 selectors right now in your production code. Selecting elements you want to change lies at the heart of jQuery's power, and CSS3 selectors give you even more tools to work with.

Helpful Utilities

Also included is an assortment of utility functions that implement common functions useful for writing jQuery (or are missing from JavaScript!): string trimming, the ability to easily extend objects, and more. These functions by themselves are particularly handy, but they help promote a seamless integration between jQuery and JavaScript which results in code that's easier to write and maintain.

One noteworthy utility is the `supports` function, which tests to find certain features are available on the current user's browser. Traditionally, developers have resorted to browser sniffing—determining which web browser the end user is using, based on information provided by the browser itself—to work around known issues. This has always been an unsatisfying and error-prone practice. Using the jQuery `supports` utility function, you can test to see if a certain feature is available to the user, and easily build applications that degrade gracefully on older browsers, or those not standards-compliant.

jQuery UI

jQuery has already been used to make some impressive widgets and effects, some of which were useful enough to justify inclusion in the core jQuery library itself.

However, the jQuery team wisely decided that in order to keep the core library focused, they'd separate out higher-level constructs and package them into a neat library that sits on top of jQuery.

That library is called **jQuery User Interface** (generally abbreviated to just **jQuery UI**), and it comprises a menagerie of useful effects and advanced widgets that are accessible and highly customizable through the use of themes. Some of these features are illustrated in Figure 1.1.

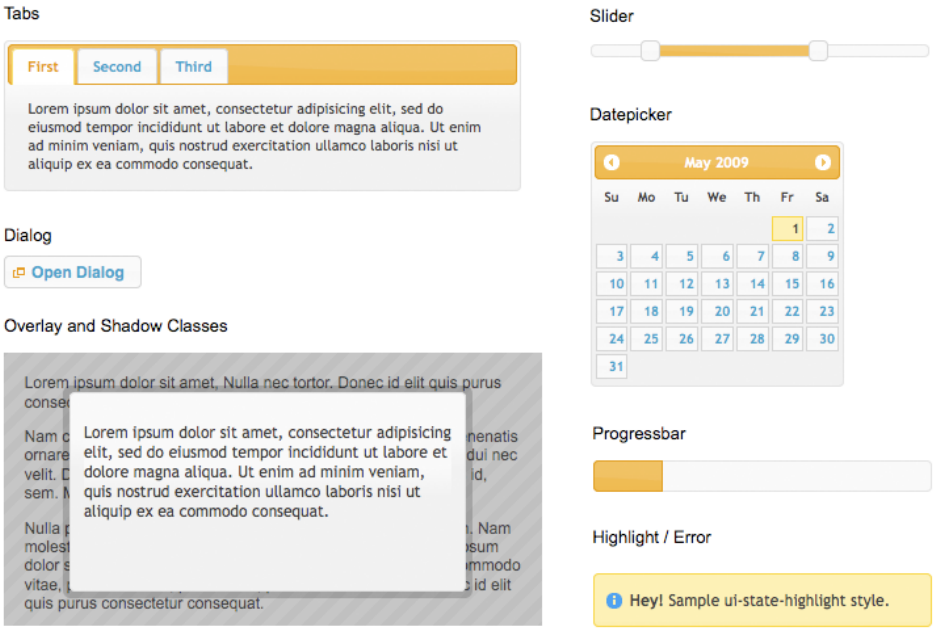

Figure 1.1. A few jQuery UI widgets

Accordions, sliders, dialog boxes, date pickers, and more—all ready to be used right now! You could spend a bunch of time creating them yourself in jQuery (as these have been) but the jQuery UI controls are configurable and sophisticated enough that your time would be better spent elsewhere—namely implementing your unique project requirements rather than ensuring your custom date picker appears correctly across different browsers!

We'll certainly be using a bunch of jQuery UI functionality as we progress through the book. We'll even integrate some of the funky themes available, and learn how to create our own themes using the jQuery UI ThemeRoller tool.

Plugins

The jQuery team has taken great care in making the jQuery library extensible. By including only a core set of features while providing a framework for extending the library, they've made it easy to create plugins that you can reuse in all your jQuery projects, as well as share with other developers. A lot of fairly common functionality has been omitted from the jQuery core library, and relegated to the realm of the plugin. Don't worry, this is a feature, not a flaw. Any additional required functionality can be included easily on a page-by-page basis to keep bandwidth and code bloat to a minimum.

Thankfully, a lot of people have taken advantage of jQuery's extensibility, so there are already hundreds of excellent, downloadable plugins available from the jQuery plugin repository, with new ones added all the time. A portion of this can be seen in Figure 1.2.

Figure 1.2. The jQuery plugin repository

Whenever you're presented with a task or problem, it's worth checking first to see if there's a plugin that might suit your needs. That's because almost any functionality you might require has likely already been turned into a plugin, and is available and ready for you to start using. Even if it turns out that you need to do some work yourself, the plugin repository is often the best place to steer you in the right direction.

Keeping Markup Clean

Separating script behavior from page presentation is best practice in the web development game—though it does present its share of challenges. jQuery makes it a

cinch to completely rid your markup of inline scripting, thanks to its ability to easily hook elements on the page and attach code to them in a natural, CSS-like manner. jQuery lacks a mechanism for adding inline code, so this separation of concerns leads to leaner, cleaner, and more maintainable code. Hence, it's easy to do things the right way, and almost impossible to do them the wrong way!

And jQuery isn't limited to meddling with a page's existing HTML—it can also add new page elements and document fragments via a collection of handy functions. There are functions to insert, append, and prepend new chunks of HTML anywhere on the page. You can even replace, remove, or clone existing elements—all functions that help you to progressively enhance your sites, thus providing a fully featured experience to users whose browsers allow it, and an acceptable experience to everyone else.

Widespread Adoption

If you care to put every JavaScript library you can think of into Google Trends,[1] you'll witness jQuery's exponential rise to superstardom. It's good to be in the in crowd when it comes to libraries, as popularity equates to more active code development and plenty of interesting third-party goodies.

Countless big players on the Web are jumping on the jQuery bandwagon: IBM, Netflix, Google (which both uses and hosts the jQuery library), and even Microsoft, which now includes jQuery with its MVC framework. With such a vast range of large companies on side, it's a safe bet that jQuery will be around for some time to come—so the time and effort you invest in learning it will be well worth your while!

jQuery's popularity has also spawned a large and generous community that's surprisingly helpful. No matter what your level of skill, you'll find other developers patient enough to help you out and work through any issues you have. This caring and sharing spirit has also spread out to the wider Internet, blossoming into an encyclopedia of high quality tutorials, blog posts, and documentation.

[1] http://www.google.com/trends/

What's the downside?

There barely is a downside! The main arguments against using any JavaScript library have always been speed and size: some say that using a library adds too much download bloat to pages, while others claim that libraries perform poorly compared with leaner custom code. Though these arguments are worth considering, their relevance is quickly fading.

First, as far as size is concerned, jQuery is lightweight. The core jQuery library has always had a fairly small footprint—about 19KB for the basics, less than your average JPG image. Any extras your project needs (such as plugins or components from the jQuery UI library) can be added in a modular fashion—so you can easily count your bandwidth calories.

Speed (like size) is becoming a decreasing concern as computer hardware specifications rise and browsers' JavaScript engines grow faster and faster. Of course, this is far from implying that jQuery is slow—the jQuery team seem to be obsessed with speed! Every new release is faster than the last, so any benefit you might derive from rolling your own JavaScript is shrinking every day.

When it comes to competing JavaScript libraries (and there are more than a handful out there), jQuery is the best at doing what jQuery does: manipulating the DOM, adding effects, and making Ajax requests. Still, many of the libraries out there are of excellent quality and excel in other areas, such as complex class-based programming. It's always worth looking at the alternatives, but if the reasons we've outlined appeal to you, jQuery is probably the way to go.

But enough talk: time for jQuery to put its money where its mouth is!

Downloading and Including jQuery

Before you can fall in love with jQuery (or at least, judge it for yourself) you need to obtain the latest version of the code and add it to your web pages. There are a few ways to do this, each with a couple of options available. Whatever you choose, you'll need to include jQuery in your HTML page, just as you would any other JavaScript source file.

It's Just JavaScript!

Never forget that jQuery is just JavaScript! It may look and act superficially different—but underneath it's written in JavaScript, and consequently it's unable to do anything that plain old JavaScript can't. This means we'll include it in our pages the same way we would any other JavaScript file.

Downloading jQuery

This is the most common method of acquiring the jQuery library—just download it! The latest version is always available from the jQuery web site.[2] The big shiny download button will lead us to the Google code repository, where we can grab the latest "production compression level" version.

Click the download link and save the JavaScript file to a new working folder, ready for playing with. You'll need to put it where our HTML files can see it: commonly in a **scripts** or **javascript** directory beneath your site's document root. For the following example, we'll keep it very simple and put the library in the same directory as the HTML file.

To make it all work, we need to tell our HTML file to include the jQuery library. This is done by using a `script` tag inside the `head` section of the HTML document. The `head` element of a very basic HTML file including jQuery would look a little like this:

```
<head>
  <title>Hello jQuery world!</title>
  <script type='text/javascript' src='jquery-1.4-min.js'></script>
  <script type='text/javascript' src='script.js'></script>
</head>
```

The first script tag on the page loads the jQuery library, and the second script tag points to a **script.js** file, which is where we'll run our own jQuery code. And that's it: you're ready to start using jQuery.

We said earlier that downloading the jQuery file is the most common approach—but there are a few other options available to you, so let's have a quick look at them

[2] http://jquery.com/

before we move on. If you just want to start playing with jQuery, you can safely skip the rest of this section.

The Google CDN

An alternative method for including the jQuery library that's worth considering is via the Google **Content Delivery Network (CDN)**. A CDN is a network of computers that are specifically designed to serve content to users in a fast and scalable manner. These servers are often distributed geographically, with each request being served by the nearest server in the network.

Google hosts several popular, open-source libraries on their CDN, including jQuery (and jQuery UI—which we'll visit shortly). So, instead of hosting the jQuery files on your own web server as we did above, you have the option of letting Google pick up part of your bandwidth bill. You benefit from the speed and reliability of Google's vast infrastructure, with the added bonus of the option to always use the latest version of jQuery.

Another benefit of using the Google CDN is that many users will already have downloaded jQuery from Google when visiting another site. As a result, it will be loaded from cache when they visit your site (since the URL to the JavaScript file will be the same), leading to significantly faster load times. You can also include the more hefty jQuery UI library via the same method, which makes the Google CDN well worth thinking about for your projects: it's going to save you money and increase performance when your latest work goes viral!

There are a few different ways of including jQuery from the Google CDN. We're going to use the simpler (though slightly less flexible) path-based method:

```
<head>
  <title>Hello jQuery world!</title>
  <script type="text/javascript" src="http://ajax.googleapis.com/
➥ajax/libs/jquery/1.4.0/jquery.min.js"></script>
  <script type='text/javascript' src='script.js'></script>
</head>
```

It looks suspiciously like our original example—but instead of pointing the `script` tag to a local copy of jQuery, it points to one of Google's servers.

Obtaining the Latest Version with Google CDN

If you look closely at the URL pointing to Google's servers, you'll see that the version of jQuery is specified by one of the path elements (the `1.4.0` in our example). If you like using the latest and greatest, however, you can remove a number from the end of the version string (for example, 1.4) and it will return the latest release available in the 1.4 series (1.4.1, 1.4.2, and so on). You can even take it up to the whole number (1), in which case Google will give you the latest version even when jQuery 1.5 and beyond are released!

Be careful though: there'll be no need to update your HTML files when a new version of jQuery is released, but it will be necessary to look out for any library changes that might affect your existing functionality.

If you'd like to examine the slightly more complex "Google loader" method of including libraries, there's plenty to read about the Google CDN on its web site.[3]

Nightlies and Subversion

Still more advanced options for obtaining jQuery are listed on the official Downloading jQuery documentation page.[4] The first of these options is the nightly builds. **Nightlies** are automated builds of the jQuery library that include all new code added or modified during the course of the day. Every night the very latest development versions are made available for download, and can be included in the same manner as the regular, stable library.

And if every single night is *still* too infrequent for you, you can use the Subversion repository to retrieve the latest up-to-the-minute source code. **Subversion** is an open-source version control system that the jQuery team uses. Every time a developer submits a change to jQuery, you can download it instantly.

Beware, however: both the nightly and Subversion jQuery libraries are often untested. They can (and will) contain bugs, and are subject to frequent changes. Unless you're looking to work on the jQuery library itself, it's probably best to skip these options.

[3] http://code.google.com/apis/ajaxlibs/documentation/
[4] http://docs.jquery.com/Downloading_jQuery

Uncompressed or compressed?

If you had a poke around on the jQuery download page, you might have also spied a couple of different download format options: compressed (also called **minified**), and uncompressed (also called "development").

Typically, you'll want to use the minified version for your production code, where the jQuery source code is compressed: spaces and line breaks have been removed and variable names are shortened. The result is exactly the same jQuery library, but contained in a JavaScript file that's much smaller than the original. This is great for reducing bandwidth costs for you, and speeding up page requests for the end user.

The downside of the compressed file is readability. If you examine the minified jQuery file in your text editor (go on!), you'll see that it's practically illegible: a single line of garbled-looking JavaScript. The readability of the library is inconsequential most of the time, but if you're interested in how jQuery is actually working, the uncompressed development version is a commented, readable, and quite beautiful example of JavaScript.

Anatomy of a jQuery Script

Now that we've included jQuery in our web page, let's have a look at what this baby can do. The jQuery syntax may look a little bit odd the first time you see it, but it's really quite straightforward, and best of all, it's highly consistent. After writing your first few commands, the style and syntax will be stuck in your head and will leave you wanting to write more.

The jQuery Alias

Including jQuery in your page gives you access to a single magical function called (strangely enough) `jQuery`. Just one function? It's through this one function that jQuery exposes hundreds of powerful tools to help add another dimension to your web pages.

Because a single function acts as a gateway to the entire jQuery library, there's little chance of the library function names conflicting with other libraries, or with your own JavaScript code. Otherwise, a situation like this could occur: let's say jQuery defined a function called `hide` (which it has) and you also had a function called

`hide` in your own code, one of the functions would be overwritten, leading to unanticipated events and errors.

We say that the jQuery library is contained in the jQuery *namespace*. Namespacing is an excellent approach for playing nicely with other code on a page, but if we're going to use a lot of jQuery (and we are), it will quickly become annoying to have to type the full jQuery function name for every command we use. To combat this issue, jQuery provides a shorter alias for accessing the library. Simply, it's `$`.

The dollar sign is a short, valid, and cool-looking JavaScript variable name. It might seem a bit lazy (after all, you're only saving five keystrokes by using the alias), but a full page of jQuery will contain scores of library calls, and using the alias will make the code much more readable and maintainable.

Using Multiple Libraries

The main reason you might want to use the full `jQuery` call rather than the alias is when you have multiple JavaScript libraries on the same page, all fighting for control of the dollar sign function name. The dollar sign is a common function name in several libraries, often used for selecting elements. If you're having issues with multiple libraries, check out Appendix A: Dealing with Conflicts.

Dissecting a jQuery Statement

We know that jQuery commands begin with a call to the `jQuery` function, or its alias. Let's now take out our scalpels and examine the remaining component parts of a jQuery statement. Figure 1.3 shows both variants of the same jQuery statement (using the full function name or the `$` alias).

selector	action	parameters
`jQuery('p')`	`.css`	`('color', 'blue');`
`$('p')`	`.css`	`('color', 'blue');`

Figure 1.3. A typical jQuery statement

Each command is made up of four parts: the jQuery function (or its alias), selectors, actions, and parameters. We already know about the jQuery function, so let's look at each of the other elements in turn. First, we use a selector to select one or more

elements on the web page. Next, we choose an action to be applied to each element we've selected. We'll see more and more actions as we implement effects throughout the book. And finally, we specify some parameters to tell jQuery how exactly we want to apply the chosen action. Whenever you see jQuery code, try to break it up into these component parts. It will make it a lot easier to comprehend when you're just starting out.

In our example above, we've asked the selector to select all the paragraph tags (the HTML `<p>` tags) on the page. Next, we've chosen jQuery's `css` action, which is used to modify a CSS property of the paragraph elements that were initially selected. Finally, we've passed in some parameters to set the CSS `color` property to the value `blue`. The end result? All our paragraphs are now blue! We'll delve deeper into selectors and the `css` action in Chapter 2.

Our example passed two parameters (`color` and `blue`) to the `css` action, but the number of parameters passed to an action can vary. Some require zero parameters, some accept multiple sets of parameters (for changing a whole bunch of properties at once), and some require that we specify another JavaScript function for running code when an event (like an element being clicked) happens. But all commands follow this basic anatomy.

Bits of HTML—aka "The DOM"

jQuery has been designed to integrate seamlessly with HTML and CSS. If you're well-versed in CSS selectors and know, for example, that `div#heading` would refer to a `div` element with an `id` of `heading`, you might want to skip this section. Otherwise, a short crash course in CSS selectors and the **Document Object Model (DOM)** is in order.

The DOM doesn't pertain specifically to jQuery; it's a standard way of representing objects in HTML that all browser makers agreed to follow. A good working knowledge of the DOM will ensure a smooth transition to jQuery ninja-hood.

The DOM is what you call bits of rendered HTML when you're talking to the cool kids around the water cooler. It's a hierarchal representation of your HTML markup—where each element (such as a `div` or a `p`) has a **parent** (its "container"), and can also have one or more nested **child** elements. Each element can have an `id` and/or it can have one or more `class` attributes—which generally you assign in

your HTML source file. When the browser reads an HTML page and constructs the DOM, it displays it as a web page comprising objects that can either sit there looking pretty (as a static page) or, more interestingly, be manipulated by our code.

A sample DOM fragment is illustrated in Figure 1.4. As you can see, `body` has two child elements: an `h1` and a `p`. These two elements, by virtue of being contained in the same parent element, are referred to as **siblings**.

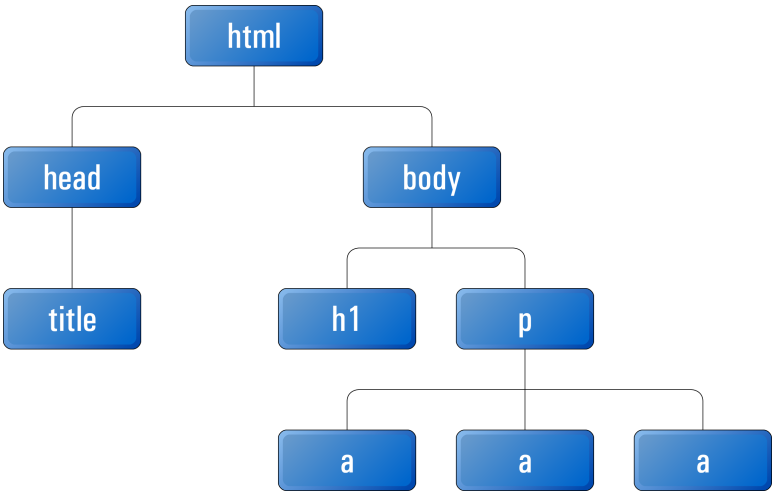

Figure 1.4. An example of a DOM fragment

An element's `id` uniquely identifies the element on the page:

```
<div id="footer">Come back and visit us soon!</div>
```

The `div` has been assigned an `id` of `footer`. It uses an `id` because it's unique: there should be one, and only one, on the page. The DOM also lets us assign the same name to multiple page elements via the `class` attribute. This is usually done on elements that share a characteristic:

```
<p class="warning">Sorry, this field must be filled in!</p>
<span class="warning">Please try again</span>
```

In this example, multiple elements on the same page are classified as a "warning." Any CSS applied to the `warning` `class` will apply to both elements. Multiple `class` attributes on the same element (when they're required) are separated by spaces.

When you write your CSS, you can hook elements by `id` with a hash symbol, or by `class` with a period:

```
#footer { border: 2px solid black }
.warning { color: red }
```

These CSS rules will give a black border to the element with an `id` of `footer`, and ensure that all elements with a `class` of `warning` will be displayed in red text.

When it comes time to write some jQuery, you will find that knowing about CSS selectors and the DOM is important: jQuery uses the same syntax as CSS for selecting elements on the page to manipulate. And once you've mastered selecting, the rest is easy—thanks to jQuery!

If You Choose to Accept It ...

jQuery is a stable and mature product that's ready for use on web sites of any size, demonstrated by its adoption by some of the veritable giants of the Internet. Despite this, it's still a dynamic project under constant development and improvement, with each new version offering up performance boosts and clever additional functionality. There's no better time than now to start learning and using jQuery!

As we work through the book you'll see that there's a lot of truth in the jQuery motto, "write less, do more." It's an easy and fun library with a gentle learning curve that lets you do a lot of cool stuff with very little code. And as you progress down the path to jQuery ninja-hood, we hope you'll also acquire a bit of respect for and understanding of JavaScript itself.

In the Chapter 2, we'll dive into jQuery and start using it to add some shine to our client's web site. Speaking of our client, it's time we met him …

Chapter 2

Selecting, Decorating, and Enhancing

"In phase two, we are going to want to harness the social and enable Web 2.0 community-based, crowd-sourced, Ajax, um, interactions," says our new client. "But for now we just need some basic stuff changed on our site."

Our client is launching a startup called StarTrackr! It uses GPS and RFID technology to track popular celebrities' exact physical location—then sells that information to fans. It's been going great guns operating out of a friend's local store, but now they're taking the venture online.

"Can you do it? Here's a list that needs to be live by Friday, close of business."

You survey the list. By amazing coincidence you notice that all of the requests can be implemented using jQuery. You look at your calendar. It's Friday morning. Let's get started!

The first task on the list is to add a simple JavaScript alert when the existing site loads. This is to let visitors know that StarTrackr! is not currently being sued for invasion of privacy (which was recently implied in a local newspaper).

Sure, we could use plain old JavaScript to do it, but we know that using jQuery will make our lives a lot easier—plus we can learn a new technology as we go along! We already saw the anatomy of a jQuery statement in Chapter 1; now let's look at the steps required to put jQuery into action: we wait until the page is ready, select our target, and then change it.

You may have probably guessed that jQuery can be more complicated than this—but only a little! Even advanced effects will rely on this basic formula, with multiple iterations of the last two steps, and perhaps a bit of JavaScript know-how. For now, let's start nice and easy.

Making Sure the Page Is Ready

Before we can interact with HTML elements on a page, those elements need to have been loaded: we can only change them once they're already there. In the old days of JavaScript, the only reliable way to do this was to wait for the entire page (including images) to finish loading before we ran any scripts.

Fortunately for us, jQuery has a very cool built-in event that executes our magic as soon as possible. Because of this, our pages and applications appear to load much faster to the end user:

chapter_02/01_document_ready/script.js

```
$(document).ready(function() {
  alert('Welcome to StarTrackr! Now no longer under police …');
});
```

The important bits here (highlighted in bold) say, "When our document is ready, run our function." This is one of the most common snippets of jQuery you're likely to see. It's usually a good idea to do a simple alert test like this to ensure you've properly included the jQuery library—and that nothing funny is going on.

You'll Be Seeing `$(document).ready()` a Lot!

Almost everything you do in jQuery will need to be done *after* the document is ready—so we'll be using this action a lot. It will be referred to as the document-ready event from now on. Every example that follows in this book, unless otherwise stated, needs to be run from inside the document-ready event. You should only need to declare it once per page though.

The document-ready idiom is so common, in fact, that there's a shortcut version of it:

```
$(function() { alert('Ready to do your bidding!'); });
```

If you'd like to use the shortcut method, go ahead! The expanded version is arguably a better example of self-documenting code; it's much easier to see at a glance exactly what's going on—especially if it's buried in a page of another developer's JavaScript!

At a cursory glance, the document-ready event looks much removed from the structure we encountered back in our jQuery anatomy class, but on closer inspection we can see that the requisite parts are all accounted for: the selector is `document`; the action is `ready`; and the parameter is a function that runs some code (our `alert`).

Selecting: The Core of jQuery

Time is ticking, and deadlines wait for no one. The client has noted that people have been having quoting incorrect celebrity IDs from the web site. This is because the celebrities' names are all laid out in one big table and it's difficult for users to line up a celebrity with the correct reference ID. Our client tells us that he wants every other row to be a light gray color so the users can easily find their favorite celebrity.

We have jQuery ready to do our bidding—it just needs us to choose a target for it. Selecting the elements you want to modify on the page is really the art of jQuery. One of the biggest differences between being a novice and ninja is the amount of time it takes you to grab the elements you want to play with!

You might remember from our jQuery anatomy class that all of our selectors are wrapped in the `jQuery` function:

```
jQuery(<selectors go here>)
```

Or the alias:

```
$(<selectors go here>)
```

We'll be using the shortcut alias for the remainder of the book—it's much more convenient. As we mentioned earlier, there's no real reason to use the full jQuery name unless you're having conflict issues with other libraries (see the section called "Avoiding Conflicts" in Chapter 9).

Simple Selecting

Our task is to select alternate table rows on the celebrity table. How do we do this? When selecting with jQuery, your goal should be to be only as specific as required: you want to find out the most concise selector that returns exactly what you want to change. Let's start by taking a look at the markup of the Celebrities table, shown in Figure 2.1.

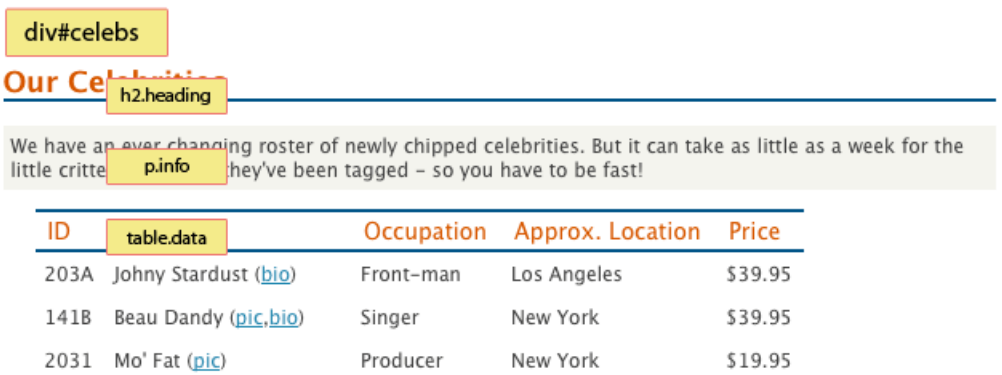

Figure 2.1. `class` and `id` attributes in the HTML page

We could start by selecting every table row element on the entire page. To select by element type, you simply pass the element's HTML name as a string parameter to the `$` function. To select all table row elements (which are marked up with the `<tr>` tag), you would simply write:

```
$('tr')
```

Nothing Happens!

If you run this command, nothing will happen on the page. This is expected—after all, we're just selecting elements. But there's no need to worry; soon enough we'll be modifying our selections in all sorts of weird and wonderful ways.

Similarly, if we wanted to select every paragraph, `div` element, `h1` heading, or `input` box on the page, we would use these selectors accordingly:

```
$('p')
$('div')
$('h1')
$('input')
```

But we don't want to change *every* table row on the celebrity page: just the rows in the table that have the celebrity data. We need to be a bit more specific, and select first the containing element that holds the list of celebrities. If you have a look at the HTML and at Figure 2.1, you can see that the `div` that contains our celebrity `table` has an `id` of `celebs`, while the `table` itself has a `class` of `data`. We could use either of these to select the `table`.

jQuery borrows the conventions from CSS for referring to `id` and `class` names. To select by `id`, use the hash symbol (#) followed by the element's `id`, and pass this as a string to the jQuery function:

```
$('#celebs')
```

You should note that the string we pass to the jQuery function is exactly the same format as a CSS `id` selector. Because `id`s should be unique, we expect this to only return one element. jQuery now holds a reference to this element.

Similarly, we can use a CSS `class` selector to select by `class`. We pass a string consisting of a period (`.`) followed by the element's `class` name to the jQuery function:

```
$('.data')
```

Both of these statements will select the table but, as mentioned earlier when we talked about the DOM, a `class` can be shared by multiple elements—and jQuery

will happily select as many elements as we point it to. If there were multiple tables (or any other elements for that matter) that also had the `class` `data`, they'd all be selected. For that reason, we'll stick to using the `id` for this one!

Can You Be More Specific?

Just like with CSS, we can select either `$('.data')` or the more specific `$('table.data')`. By specifying an element type in addition to the `class`, the selector will only return `table` elements with the `class` `data`—rather than *all* elements with the class `data`. Also, like CSS, you can add parent container selectors to narrow your selection even further.

Narrowing Down Our Selection

We've selected the table successfully, though the table itself is of no interest to us—we want *every other row* inside it. We've selected the containing element, and from that containing element we want to pick out all the descendants that are table rows: that is, we want to specify all table rows *inside* the containing `table`. To do this, we put a space between the ancestor and the descendant:

```
$('#celebs tr')
```

You can use this construct to drill down to the elements that you're looking for, but for clarity's sake try to keep your selectors as succinct as possible.

Let's take this idea a step further. Say we wanted to select all `span` elements inside of `p` elements, which are themselves inside `div` elements—but only if those `divs` happen to have a class of `fancy`. We would use the selector:

```
$('div.fancy p span')
```

If you can follow this, you're ready to select just about anything!

Testing Our Selection

Right, back to our task at hand. It feels like we're getting closer, but so far we've just been selecting blindly with no way of knowing if we're on the right path. We need a way of confirming that we're selecting the correct elements. A simple way to achieve this is to take advantage of the `length` property. `length` returns the number

of elements currently matched by the selector. We can combine this with the good ol' trusty `alert` statement to ensure that our elements have been selected:

chapter_02/02_selecting/script.js

```
$(document).ready(function() {
  alert($('#celebs tr').length + ' elements!');
});
```

This will alert the length of the selection—7 elements—for the celebrity `table`. This result might be different from what you'd expect, as there are only six celebrities in the table! If you have a look at the HTML, you'll see where our problem lies: the `table` header is also a `tr`, so there are seven rows in total. A quick fix involves narrowing down our selector to find only table rows that lie inside the `tbody` element:

chapter_02/03_narrowing_selection/script.js

```
$(document).ready(function() {
  alert($('#celebs tbody tr').length + ' elements!');
});
```

This will alert the correct length of 6 elements—the jQuery object is now holding our six celebrity table row elements.

If the alert shows 0, you'll know there's a mistake in your selector. A good way to troubleshoot this sort of issue is to reduce your selector to the smallest, simplest one possible.

In our example, we could simply write `$('#celebs')`, which would select just the table element and alert a length of 1. From here you can make your selectors more specific, and check that you're selecting the correct number of elements as you go.

Filters

With the knowledge that we've successfully selected all of the table rows, narrowing our selection down to every other row is simple—because jQuery has a **filter** to do it. A filter removes certain items, and keeps only the ones we want. You'll acquire a feel for what can be filtered as we work through some more examples, but for now we'll just jump straight to the filter we need for our zebra stripes:

chapter_02/04_filters/script.js

```
$(document).ready(function() {
  alert($('#celebs tbody tr:even').length + ' elements!');
});
```

Filters are attached to the item you want to filter (in this case, the table rows) and are defined by a colon, followed by the filter name. The `:even` filter used here keeps every even-indexed element in the selection and removes the rest, which is what we want. When we alert the selection length now, we see 3, as expected. All of our odd-numbered rows have been filtered out of the selection. There is a wide array of jQuery selector filters available to us: `:odd` (as you might expect), `:first`, `:last`, `:eq()` (for selecting, for example, the third element), and more. We'll look at each of these in more detail as we need them throughout the book.

Selecting Multiple Elements

One last trick for basic selecting is the ability to select multiple elements in a single statement. This is very useful, as we'll often want to apply the same action to several elements in unrelated parts of the page. Separating the selector strings with commas allows you to do this. For example, if we wanted to select every paragraph, `div` element, `h1` heading, and `input` box on the page, we'd use this selector:

```
$('p,div,h1,input')
```

Learning how to use all these different selectors together to access exactly the page elements you want is a big part of mastering jQuery. It's also one of the most satisfying parts of using jQuery, since you can pack some fairly complex selection logic into a single short line of code!

Becoming a Good Selector

Selecting may seem quite easy and, up to a point, it is. But what we've covered so far has only just scratched the surface of selecting. In most cases the basics are all you'll need: if you're simply trying to target an element or a bunch of related elements, the element name, `id`, and `class` are the most efficient and easiest ways to achieve this.

When moving around the DOM from a given element, the situation becomes a little trickier. jQuery provides a myriad of selectors and actions for traversing the DOM. **Traversing** means traveling up and down the page hierarchy, through parent and child elements. You can add and remove elements as you go, applying different actions at each step—which lets you perform some mind-bogglingly complex actions in a single jQuery statement!

If you're a wiz at CSS, you'll already be familiar with a lot of the statements; they're mostly borrowed directly from the CSS specification. But there are probably a few that you're unfamiliar with, especially if you've yet to spend much time learning CSS3 selectors. Of course, we'll be covering and learning advanced selection techniques as we implement them in our examples and demos. For this reason, any time you want to find out more about all the jQuery selectors available, you can just head over to the online documentation[1] and browse away!

Decorating: CSS with jQuery

Selecting elements in jQuery is the hard part. Everything else is both easy and fun. After we have selected our targets, we are able to manipulate them to build effects or interfaces. In this section we will cover a series of jQuery actions relating to CSS: adding and removing styles, classes, and more. The actions we execute will be applied individually to every element we've selected, letting us bend the page to our will!

Reading CSS Properties

Before we try changing CSS properties, let's look first into how we can simply access them. jQuery lets us do this with the `css` function. Try this:

chapter_02/05_reading_css_properties/script.js

```
$(document).ready(function() {
  var fontSize = $('#celebs tbody tr:first').css('font-size');
  alert(fontSize);
});
```

[1] http://api.jquery.com/category/selectors/

This code will alert the font size of the first element matched by the selector (as you've likely guessed, the `:first` filter will return the first element among those matched by the selector).

CSS Properties of Multiple Elements

You *can* ask for a CSS property after selecting multiple elements, but this is almost always a bad idea: a function can only return a single result, so you'll still only obtain the property for the first matched element.

The nifty aspect about retrieving CSS properties with this method is that jQuery gives you the element's *calculated* style. This means that you'll receive the value that's been rendered in the user's browser, rather than the value entered in the CSS definition. So, if you gave a `div` a height of, say, 200 pixels in the CSS file, but the content inside it pushed the height over 200 pixels, jQuery would provide you with the actual height of the element, rather than the 200 pixels you'd specified.

We'll see why that's really important when we come to implement some funky tricks a bit later.

Setting CSS Properties

So far we've yet to see jQuery actually *do* anything, and it's high time to remedy that. We know the page is ready (since we popped up an alert), and we're fairly sure we've selected the elements we're interested in. Let's check that we really have:

chapter_02/06_zebra_striping/script.js

```
$(document).ready(function() {
  $('#celebs tbody tr:even').css('background-color','#dddddd');
});
```

You probably saw that coming! This is the same `css` function we used to read a CSS property, but now it's being passed an extra parameter: the value we wish to set for that property. We've used the action to set the `background-color` to the value `#dddddd` (a light gray). Open the file from the code archive in your browser and test that it's working correctly. You can see the result in Figure 2.2.

ID	Name	Occupation	Approx. Location	Price
203A	Johny Stardust (bio)	Front-man	Los Angeles	$39.95
141B	Beau Dandy (pic,bio)	Singer	New York	$39.95
2031	Mo' Fat (pic)	Producer	New York	$19.95
007F	Kellie Kelly (bio,press)	Singer	Omaha	$11.95
8A05	Darth Fader (pic)	DJ	London	$19.95
6636	Glendatronix (bio,press)	Keytarist	London	$39.95

Figure 2.2. Zebra striping implemented with jQuery

Were You Ready?

As mentioned previously, this command must be issued from within our document-ready function. If we run the command before the DOM is ready, the selector will go looking for the `#celebs` element, but will find nothing that matches. At this point it will give up; it won't even look for the `tr` elements, let alone change the background style.

This is true for all of the examples that follow, so remember to wrap your code in the document-ready function.

It's looking good! But perhaps we should add a little extra to it—after all, more is more! What about a shade lighter font color to really define our stripes? There are a few ways we could add a second CSS property. The simplest way is to repeat the entire jQuery statement with our new values:

chapter_02/07_multiple_properties_1/script.js *(excerpt)*

```
$('#celebs tbody tr:even').css('background-color','#dddddd');
$('#celebs tbody tr:even').css('color', '#666666');
```

These lines are executed one after the other. Though the end result is correct, it will become quite messy and inefficient if we have to change a whole slew of properties. Thankfully, jQuery provides us with a nice way to set multiple properties at the same time, using an **object literal**. Object literals are a JavaScript concept beyond the scope of this book, but for our purposes, all you need to know is that they provide an easy way of grouping together key/value pairs. For CSS, object literals allow us

to match up our CSS properties (the keys) with the matching CSS values (the values) in a neat package:

chapter_02/08_multiple_properties_2/script.js *(excerpt)*

```
$('#celebs tbody tr:even').css(
  {'background-color': '#dddddd', 'color': '#666666'}
);
```

The object literal is wrapped in curly braces, with each key separated from its corresponding value by a colon, and each key/value pair separated by a comma. It's passed as a single parameter to the `css` function. Using this method you can specify as many key/value pairs as you like—just separate them with commas. It's a good idea to lay out your key/value pairs in a readable manner so you can easily see what's going on when you come back to your code later. This is especially helpful if you need to set a larger number of properties. As an example:

chapter_02/09_multiple_properties_3/script.js *(excerpt)*

```
$('#celebs tbody tr:even').css({
  'background-color': '#dddddd',
  'color': '#666666',
  'font-size': '11pt',
  'line-height': '2.5em'
});
```

To Quote or Not to Quote

In general, when dealing with JavaScript objects, it's unnecessary for the keys to be in quotes. However, for jQuery to work properly, any key that contains a hyphen (as our `background-color` and `font-size` examples do) must be placed in quotes, or written in camel case (`like backgroundColor`).

Additionally, any key that's already a keyword in the JavaScript language (such as `float` and `class`) must also be written in quotes.

It can be confusing trying to remember which keys need to be quoted and which don't, so it's to be recommended that you just put all object keys in quotes each time.

Classes

Excellent! We've already struck two tasks off the client's list, and we have some funky jQuery happening. But if you stop and have a look at our last solution, you might notice something a little fishy. If you were to inspect the zebra-striped rows in a development tool such as Firebug, you'd notice that the CSS properties have been added to the paragraphs *inline*, as illustrated in Figure 2.3.

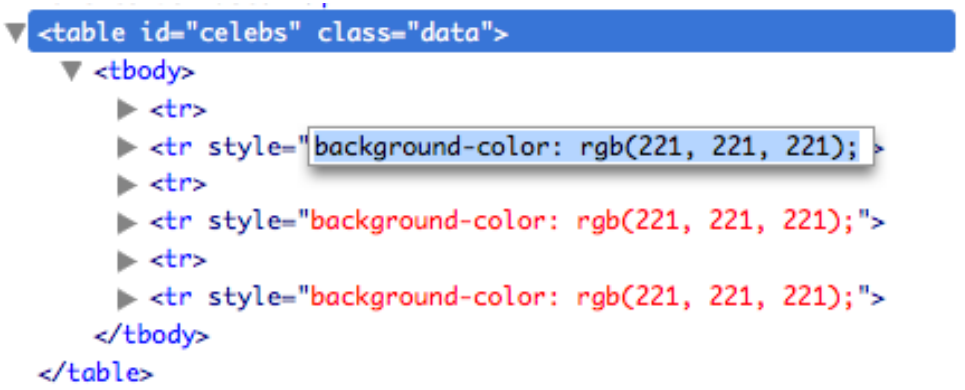

Figure 2.3. Inline styles viewed with Firebug

Firebug

Firebug is a particularly useful tool for examining the DOM in your browser, as well as monitoring and editing CSS, HTML, and JavaScript (including jQuery). A debugger's Swiss Army knife for the Web, it will save you hours by helping you see exactly what your browser thinks is going on. It's available as a Mozilla Firefox extension, or as a stand-alone JavaScript file that you can include in your projects if you develop using another browser.

Inline styles are a big no-no in HTML/CSS best practice, right? That's quite true, and this also applies in jQuery: to keep your code clear and maintainable, it makes more sense for all the styling information to be in the same place, in your CSS files. Then, as we'll soon see, you can simply toggle those styles by attaching or removing `class` attributes to your HTML tags.

There are times when it *is* a good idea to use the `css` jQuery method in the way we've just seen. The most common application is when quickly debugging code: if

you just want to outline an element in red to make sure you've selected it correctly, switching to your CSS file to add a new rule seems like a waste of time.

Adding and Removing Classes

If we need to remove the CSS from inline style rules, where should we put it? In a separate style sheet, of course! We can put the styles we want in a rule in our CSS that's targeted to a given `class`, and use jQuery to add or remove that class from targeted elements in the HTML. Perhaps unsurprisingly, jQuery provides some handy methods for manipulating the `class` attributes of DOM elements. We'll use the most common of these, `addClass`, to move our zebra stripe styles into the CSS file where they belong.

The `addClass` function accepts a string containing a `class` name as a parameter. You can also add multiple `class`es at the same time by separating the `class` names with a space, just as you do when writing HTML:

```
$('div').addClass('class_name');
$('div').addClass('class_name1 class_name2 class_name3');
```

We only want to add one `class` name, though, which we'll call `zebra`. First, we'll add the rule to a new CSS file (including it with a `link` tag in our HTML page):

chapter_02/10_adding_classes/zebra.css

```
.zebra {
  background-color: #dddddd;
  color: #666666;
}
```

Then, back in our JavaScript file, we'll modify the selector to use jQuery's `addClass` method rather than `css`:

chapter_02/10_adding_classes/script.js

```
$('#celebs tr:even').addClass('zebra');
```

The result is exactly the same, but now when we inspect the table in Firebug, we'll see that the inline styles are gone—replaced by our new class definition. This is shown in Figure 2.4.

Figure 2.4. Adding classes to table rows

That's much better. Now, if we want to change the appearance of the zebra stripes in the future, we can simply modify the CSS file; this will save us hunting through our jQuery code (potentially in multiple locations) to change the values.

There'll also be times when we want to remove class names from elements (we'll see an example of when this is necessary very soon). The action to remove a class is conveniently known as `removeClass`. This function is used in exactly the same way as `addClass`; we just pass the (un)desired class name as a parameter:

```
$('#celebs tr.zebra').removeClass('zebra');
```

It's also possible to manipulate the `id` attribute, or any other attribute for that matter, using jQuery's `attr` method. We'll cover this method in more detail later in the book.

Enhancing: Adding Effects with jQuery

Now you've reached an important milestone. You've learned the component parts of a jQuery statement: the selector, the action, and the parameters. And you've learned the steps to use the statement: make sure the document is ready, select elements, and change them.

In the following section, we'll apply these lessons to implement some cool and useful effects—and with any luck reinforce your understanding of the jQuery basics.

Hiding and Revealing Elements

The client dislikes the disclaimer on the site—he feels it reflects badly on the product—but his lawyer insists that it's necessary. So the client has requested that you add a button that will remove the text after the user has had a chance to read it:

chapter_02/11_hiding/index.html *(excerpt)*

```
<input type="button" id="hideButton" value="hide" />
```

We've added an HTML button on the page with an ID of `hideButton`. When a user clicks on this button we want the disclaimer element, which has an ID of `disclaimer`, to be hidden:

chapter_02/11_hiding/script.js *(excerpt)*

```
$('#hideButton').click(function() {
  $('#disclaimer').hide();
});
```

Run this code and make sure the disclaimer element disappears when you click the hide button.

The part in this example that makes the element actually disappear is the `hide` action. So, you might ask, what's all the other code that surrounds that line? It's what's called an event handler—an understanding of which is crucial to becoming a jQuery ninja. There are many event handlers we can use (we've used the `click` event handler here) and we'll be using a lot of them as we move on.

Event Handlers

Event handlers are named for their function of handling events. Events are actions and user interactions that occur on the web page. When an event happens, we say that it has *fired*. And when we write some code to handle the event, we say we *caught* the event.

There are thousands of events fired on a web page all the time: when a user moves the mouse, or clicks a button, or when a browser window is resized, or the scroll bar moved. We can catch, and act on, any of these events.

The first event that you were introduced to in this book was the document-ready event. Yes, that was an event handler: when the document said, "I'm ready" it fired an event, which our jQuery statement caught.

We used the `click` event handler to tell jQuery to hide the disclaimer when the button is clicked:

```
$('#hideButton').click(function() {
  $('#disclaimer').hide();
});
```

this

When an event fires, we will often want to refer to the element that fired it. For example, we might want to modify the button that the user has just clicked on in some way. Such a reference is available inside our event handler code via the JavaScript keyword `this`. To convert the JavaScript object to a jQuery object, we wrap it in the jQuery selector:

chapter_02/12_this/script.js *(excerpt)*

```
$('#hideButton').click(function() {
  $(this).hide(); // a curious disappearing button.
});
```

`$(this)` provides a nicer way to talk about the element that fired the event, rather than having to re-select it.

Where's the Action?

This might be a bit confusing when you're starting out, as the "action" component of a jQuery statement seems to have several purposes: we've seen it used to run animations, retrieve values and now, handle events! It's true—it gets around! Usually the action's name gives you a good clue to its purpose, but if you become lost, it's best to consult the index. After a while, you'll sort out the handlers from the animations from the utilities.

Revealing Hidden Elements

On with our task! The client has also specified that the user needs to be able to retrieve the disclaimer in case they close it by mistake. So let's add another button to the HTML, this time with an `id` of `showButton`:

chapter_02/13_revealing/index.html *(excerpt)*

```
<input type="button" id="showButton" value="show" />
```

We'll also add another jQuery statement to our script file, to handle showing the disclaimer when the **show** button is clicked:

chapter_02/13_revealing/script.js *(excerpt)*

```
$('#showButton').click(function() {
  $('#disclaimer').show();
});
```

Toggling Elements

Having separate buttons for hiding and showing the disclaimer seems like a waste of valuable screen real estate. It would be better to have one button that performed both tasks—hiding the disclaimer when it's visible, and showing it when it's hidden. One way we could do this is by checking if the element is visible or not, and then showing or hiding accordingly. We'll remove the old buttons and add this nice new one:

chapter_02/14_toggle_1/index.html *(excerpt)*

```
<input type="button" id="toggleButton" value="toggle" />
```

When it's clicked, we check to find out if we should show or hide the disclaimer:

chapter_02/14_toggle_1/script.js *(excerpt)*

```
$('#toggleButton').click(function() {
  if ($('#disclaimer').is(':visible')) {
    $('#disclaimer').hide();
  } else {
```

```
    $('#disclaimer').show();
  }
});
```

This introduces the `is` action. `is` takes any of the same selectors we normally pass to the jQuery function, and checks to see if they match the elements it was called on. In this case, we're checking to see if our selected `#disclaimer` is also selected by the pseudo-selector `:visible`. It can also be used to check for other attributes: if a selection is a `form` or `div`, or is enabled.

The `if` Statement

If you're entirely new to programming (that is, if you've only ever worked with HTML and CSS), that whole block of code is probably quite confusing! Don't worry, it's actually quite straightforward:

```
if (condition) {
  // this part happens if the condition is true
} else {
  // this part happens if the condition is false
}
```

The condition can be anything that JavaScript will evaluate to `true` or `false`. This sort of structure is extremely common in any type of programming, and we'll be using it a lot for the rest of the book. If you're uncomfortable with it, the best way to learn is to play around: try writing different `if` / `else` blocks using jQuery's `is` action like the one we wrote above. You'll get the hang of it in no time!

`is` will return `true` or `false` depending on whether the elements match the selector. For our purposes we'll show the element if it's hidden, and hide it if it's visible. This type of logic—where we flip between two states—is called a **toggle** and is a very useful construct.

Toggling elements between two states is so common that many jQuery functions have a version that allows for toggling. The toggle version of `show`/`hide` is simply called `toggle`, and works like this:

chapter_02/15_toggle_2/script.js *(excerpt)*

```
$('#toggleButton').click(function() {
  $('#disclaimer').toggle();
});
```

Every time you click the button, the element toggles between visible and hidden.

It would be nice, however, if the button was labeled with a more useful word than "toggle," which might be confusing to our users. What if you want to toggle the text of the button as well? As is often the case when working with jQuery, there are a few ways we could approach this problem. Here's one:

chapter_02/16_toggle_text/script.js *(excerpt)*

```
$('#toggleButton').click(function() {
  $('#disclaimer').toggle();

  if ($('#disclaimer').is(':visible')) {
    $(this).val('Hide');
  } else {
    $(this).val('Show');
  }
});
```

There's a lot in this code that will be new to you. We'll save most of the details for later, but have a look at it and see if you can figure it out yourself. (Hint: remember that the selector `$(this)` refers to the element that caused the event to fire—in this case, the button.)

Progressive Enhancement

Our disclaimer functionality is working perfectly—and our client will doubtlessly be impressed with it. However, there's one subtle aspect of our solution that we should be aware of: if a user came to our site using a browser lacking support for JavaScript, they'd see a button on the page that would do nothing when they clicked it. This would lead to a very confused user, who might even abandon our site.

"No support for JavaScript?" you might snort. "What kind of browser is unable to run JavaScript?!"

There might be more people than you think browsing the Web without JavaScript: users on very old computers or limited devices (like mobile phones); people with visual impairments who require screen readers to use the Web; and those who worry that JavaScript is an unnecessary security risk and so choose to disable it.

Depending on your site's demographic, anywhere between 5% and 10% of your users might be browsing without JavaScript capabilities, and nobody wants to alienate 10% of their customers! The solution is to provide an acceptable experience to these users—and beef it up for everyone else. This practice is known as progressive enhancement.

For our disclaimer functionality, we might settle on this compromise: we want the disclaimer to be visible to all users, so we place it in our HTML. Then, we add the ability to hide it for users with JavaScript. That said, we'd prefer to avoid displaying the show/hide button to users who'll be unable to make use of it.

One way of accomplishing this might be to hide our button with CSS, and only show it via a jQuery `css` statement. The problem with this trick is that it will fail if the user's browser also lacks support for CSS. What we'd really like to do is add the button to the page via jQuery; that way, only users with JavaScript will see the button at all. Perfect!

Adding New Elements

So far we've seen the jQuery function used for selecting, but it does have another function of equal importance: creating new elements. In fact, any valid HTML string you put inside the jQuery function will be created and made ready for you to stick on the page. Here's how we might create a simple paragraph element:

```
$('<p>A new paragraph!</p>')
```

jQuery performs several useful actions when you write this code: it parses the HTML into a DOM fragment and selects it—just as an ordinary jQuery selector does. That means it's instantly ready for further jQuery processing. For example, to add a `class` to our newly created element, we can simply write:

```
$('<p>A new paragraph!</p>').addClass('new');
```

The new paragraph will now be given the `class` `new`. Using this method you can create any new elements you need via jQuery itself, rather than defining them in your HTML markup. This way, we can complete our goal of progressively enhancing our page.

innerHTML

Internally, the HTML string is parsed by creating a simple element (such as a `div`) and setting the `innerHTML` property of that `div` to the markup you provide. Some content you pass in is unable to convert quite as easily—so it's best to keep the HTML fragments as simple as possible.

Once we've created our new elements, we need a way to insert in the page where we'd like them to go. There are several jQuery functions available for this purpose. The first one we'll look at is the `insertAfter` function. `insertAfter` will take our current jQuery selection (in this case, our newly created elements) and insert it after another selected element, which we pass as a parameter to the function.

An example is the easiest way to show how this works. This is how we'd create the toggle button using jQuery:

chapter_02/17_insert_after/script.js *(excerpt)*

```
$('<input type="button" value="toggle" id="toggleButton">')
  .insertAfter('#disclaimer');
$('#toggleButton').click(function() {
  $('#disclaimer').toggle();
});
```

As shown in Figure 2.5, the button is inserted into our page after the disclaimer, just as if we'd put it there in our HTML file.

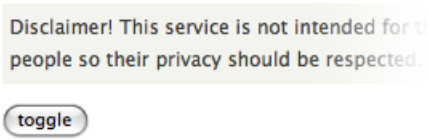

Figure 2.5. A button created and inserted with jQuery

The `insertAfter` function adds the new element as a sibling directly after the disclaimer element. If you want the button to appear *before* the disclaimer element,

you could either target the element before the disclaimer and use `insertAfter`, or, more logically, use the `insertBefore` method. `insertBefore` will also place the new element as a sibling to the existing element, but it will appear immediately before it:

chapter_02/18_insert_before/script.js *(excerpt)*

```
$('<input type="button" value="toggle" id="toggleButton">')
  .insertBefore('#disclaimer');
```

A quick refresher: when we talk about the DOM, *siblings* refer to elements on the same level in the DOM hierarchy. If you have a `div` that contains two `span` elements, the `span` elements are siblings.

If you want to add your new element as a *child* of an existing element (that is, if you want the new element to appear *inside* the existing element) then you can use the `prependTo` or `appendTo` functions:

chapter_02/19_prepend_append/script.js *(excerpt)*

```
$('<strong>START!</strong>').prependTo('#disclaimer');
$('<strong>END!</strong>').appendTo('#disclaimer');
```

As you can see in Figure 2.6, our new elements have been added to the start and the end of the actual disclaimer `div`, rather than before or after it. There are more actions for inserting and removing elements, but as they're unneeded in this round of changes, we'll address them later on.

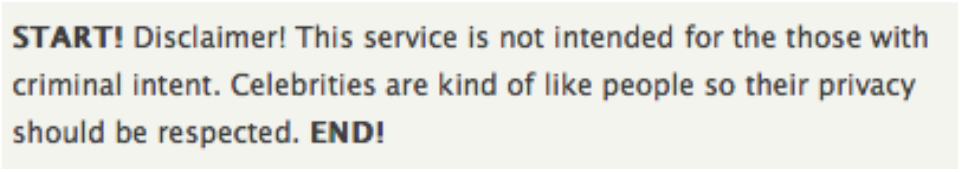

Figure 2.6. `prependTo` and `appendTo` in action

Inserting Multiple Elements

A new item is inserted *once for each element that's matched with the selector*. If your selector matches every paragraph tag, for example, the `insertAfter` action will add a new element after *every* paragraph tag. Which makes it a fairly powerful function!

Removing Existing Elements

We informed the client that up to 10% of his users might lack JavaScript capabilities and would therefore miss out on some of the advanced features we're building. He asked if we could add a message explaining that JavaScript was recommended for those people. Obviously the message should be hidden from those who *do* have JavaScript.

This seems like a perfect opportunity to learn how to remove HTML elements from a page using jQuery. We'll put the message in our HTML and remove it with jQuery; that way, only those visitors without JavaScript will see it.

Let's go ahead and add the new warning to our HTML page:

chapter_02/20_removing_elements/index.html *(excerpt)*

```
<p id="no-script">
  We recommend that you have JavaScript enabled!
</p>
```

Now we need to run our code to remove the element from the page. If a user has JavaScript disabled, our jQuery statements will fail to run and the message will remain on the screen. To remove elements in jQuery, you first select them (as usual) with a selector, and then call the `remove` method:

chapter_02/20_removing_elements/script.js *(excerpt)*

```
$('#no-script').remove();
```

The `remove` action will remove all of the selected elements from the DOM, and will also remove any event handlers or data attached to those elements. The `remove` action does not require any parameters, though you can also specify an expression to refine the selection further. Try this example:

chapter_02/21_removing_with_selector/script.js *(excerpt)*

```
$('#celebs tr').remove(':contains("Singer")');
```

Rather than removing every `tr` in the `#celebs div`, this code will remove only those rows which contain the text "Singer." This will come in handy when we look at some advanced effects in the next chapter.

Thanks to these changes, our page will work nicely for the 10% of our users without JavaScript, and even better for the remaining 90%! This is a very simple example of progressive enhancement, but it gives you a good understanding of the fundamental idea: rather than using jQuery as the underpinnings of your UI, use it to add some sugar to an already functioning experience. That way, you know no one's left behind.

In the interests of keeping our sample code small and focused, we'll stop short of delving much further into the topic. But go off and research it for yourself—it's the kind of best practice that makes you a better web developer.

Modifying Content

We can do just about anything we want to our elements now: show them, hide them, add new ones, remove old ones, style them however we like … but what if we want to change the actual content of an element? Again, jQuery provides a couple of methods for just this purpose: `text` and `html`.

The `text` and `html` actions are quite similar, as both set the content for the elements we've selected. We simply pass a string to either function:

chapter_02/22_modifying_content/script.js *(excerpt)*

```
$('p').html('good bye, cruel paragraphs!');
$('h2').text('All your titles are belong to us');
```

In both these examples the matched elements' contents will change to the string we've provided: every paragraph and `h2` tag on the page will be overwritten with our new content. The difference between `text` and `html` can be seen if we try adding some HTML to the content string:

chapter_02/23_text_vs_html/script.js *(excerpt)*

```
$('p').html('<strong>Warning!</strong> Text has been replaced … ');
$('h2').text('<strong>Warning!</strong> Title elements can be …');
```

In this case, our paragraphs will contain bold-faced text, but our `h2` tags will contain the entire content string exactly as defined, including the `<strong>` tags. The action you use to modify content will depend on your requirements: `text` for plain text or `html` for HTML.

You might wonder, "Can these new actions only *set* content?" At this stage it should be no surprise to you that we can also fetch content from our jQuery selections using the same actions:

chapter_02/24_get_content/script.js *(excerpt)*

```
alert($('h2:first').text());
```

We use the `text` action supplying no parameters, which returns the text content of the first `h2` tag on the page ("Welcome!"). Like other actions that retrieve values, this can be particularly useful for conditional statements, and it can also be great for adding essential information to our user interactions.

Basic Animation: Hiding and Revealing with Flair

All this showing and hiding and changing is useful, though visually it's somewhat unimpressive. It's time to move on to some jQuery techniques that are a bit more, shall we say, *animated*.

The core jQuery library includes a handful of basic effects that we can use to spice up our pages. And once you've had enough of these, mosey on over to the jQuery plugin repository, where you'll find hundreds more crazy effects.

Keep It Sensible

When dealing with effects and animation on the Web, it's probably a wise idea to proceed with your good taste sensors engaged. Remember, at one time the `<blink>` tag was considered perfectly sensible!

Fading In and Out

One of the most common (and timeless) effects in jQuery is the built-in fade effect. To use fading in its simplest form, just replace `show` with `fadeIn` or `hide` with `fadeOut`:

chapter_02/25_fade_in_out/script.js *(excerpt)*

```
$('#hideButton').click(function() {
  $('#disclaimer').fadeOut();
});
```

There are also a few optional parameters we can use to modify the effect, the first of which is used to control the time it takes for the fade to complete. Many jQuery effects and animations accept the time parameter—which can be passed either as a string or an integer.

We can specify the time span as a string using one of the following predefined words: `slow`, `fast`, or `normal`. For example: `fadeIn('fast')`. If you'd rather have more fine-grained control over the duration of the animation, you can also specify the time in milliseconds, as in: `fadeIn(1000)`.

Toggling Effects and Animations

Although jQuery has no specific action for toggling using fades, here's a little secret: our original `toggle` action has a few more tricks up its sleeve than we first thought. If we pass it a time span parameter, we'll see that `toggle` has the ability to animate:

chapter_02/26_toggle_fade/script.js *(excerpt)*

```
$('#toggleButton').click(function() {
  $('#disclaimer').toggle('slow');
});
```

You can see that the width, height, and opacity of the entire element are animated. If this is a bit much for you, there's another core jQuery animation effect that *does* include built-in toggle actions: sliding.

Sliding eases an element into and out of view, as if it were sliding out from a hidden compartment. It's implemented in the same manner as our fade, but with the

`slideDown`, `slideUp`, and `slideToggle` actions. As with the fade effect, we can also specify a time span:

chapter_02/27_slide_toggle/script.js *(excerpt)*

```
$('#toggleButton').click(function() {
  $('#disclaimer').slideToggle('slow');
});
```

Callback Functions

Many effects (including our slide and fade effects) accept a special parameter known as a **callback function**. Callbacks specify code that needs to run after the effect has finished doing whatever it needs to do. In our case, when the slide has finished sliding it will run our callback code:

chapter_02/28_callback_functions/script.js *(excerpt)*

```
$('#disclaimer').slideToggle('slow', function() {
  alert('The slide has finished sliding!')
});
```

The callback function is simply passed in as a second parameter to the effect action, as an anonymous function, much in the same way we provide functions as parameters to event handlers.

Anonymous Functions

In JavaScript, functions that are defined inline (such as our callbacks and event handlers) are called **anonymous functions**. They are referred to as "anonymous" simply because they don't have a name! You use anonymous functions when you only require the function to be run from one particular location.

In any situation where we're using anonymous functions, it's also possible to pass a function name yet define the function elsewhere. This is best done when the same function needs to be called in several different places. In simple cases like our examples, this can make the code a bit harder to follow, so we'll stick with anonymous functions for the moment.

Let's put our callback functions to practical use. If we want to hide our button after the disclaimer has finished sliding out of view:

chapter_02/29_callback_functions_2/script.js *(excerpt)*

```
$('#disclaimer').slideUp('slow', function() {
  $('#hideButton').fadeOut();
});
```

The disclaimer will slide up, and only once that animation is complete will the button fade from view.

A Few Tricks

Now that we've struck a few high priority requests off the client's to-do list, let's be a bit more showy and add some extra sizzle to the site. We'll add a few effects and visual highlights by building on what we've learned so far. There'll be some new constructs and actions introduced, so it's worth working through them if this is your first venture into the world of jQuery.

Highlighting When Hovering

The client is really keen about the zebra-striping usability issue. He's requested that, as well as changing the row colors, there should be an additional highlight that occurs when the user runs the mouse over the `table`.

We could implement this effect by adding event handlers to the `table` that deal with both the `mouseover` and `mouseout` events. Then we could add or remove a CSS `class` containing a background color specific to elements over which the mouse is hovering. This is much the same way we'd do it in plain old JavaScript too:

chapter_02/30_hover_highlight/script.css *(excerpt)*

```
$('#celebs tr').mouseover(function() {
  $(this).addClass('zebraHover');
});
$('#celebs tr').mouseout(function() {
  $(this).removeClass('zebraHover');
});
```

Remember that `$(this)` refers to the selected object—so we're adding and removing the `zebraHover` class to each row as the user hovers the mouse over it. Now we simply need to add a style rule to our CSS file:

chapter_02/30_hover_highlight/zebra.css *(excerpt)*

```
tr.zebraHover {
  background-color: #FFFACD;
}
```

Try this out in your browser and you'll see how great it works. However, it turns out there's an even simpler way of achieving the same result: jQuery includes a `hover` action, which combines `mouseover` and `mouseout` into a single handler:

chapter_02/31_hover_action/script.js*(excerpt)*

```
$('#celebs tbody tr').hover(function() {
  $(this).addClass('zebraHover');
}, function() {
  $(this).removeClass('zebraHover');
});
```

Notice something odd about the `hover` event handler? Instead of one, it requires two functions as parameters: one to handle the `mouseover` event, and one to handle the `mouseout` event.

How Many Callbacks?

Some event handlers require a different number of functions. For example, the toggle event handler can accept any number of functions; it will simply cycle through each callback one by one each time it fires.

We're becoming handy at adding and removing `class` attributes, so it's probably a good time to point out another helpful `class`-related action: `toggleClass`. You can guess what it does. It's an incredibly useful action that adds a `class` if the element doesn't already have it, and removes it if it does.

For example, say we wanted users to be able to select multiple rows from our table. Clicking once on a table row should highlight it, and clicking again should remove the highlight. This is easy to implement with our new jQuery skills:

chapter_02/32_toggle_class/script.js *(excerpt)*

```
$('#celebs tbody tr').click(function() {
  $(this).toggleClass('zebraHover');
});
```

Try clicking on the table rows. Cool, huh?

Spoiler Revealer

The latest news section of the StarTrackr! site provides up-to-the-minute juicy gossip about a range of popular celebrities. The news is a real drawcard on the site—most users return every day to catch the latest update. The client would like to build on the hype it's generating and add to the excitement, so he's asked for our help. We've suggested a spoiler revealer: the user can try to guess which celebrity the news is about, before clicking to find the answer.

This kind of functionality would also make a great addition to a site containing movie reviews, for example. You could hide any parts of the review that give away details of the movie's story, but allow users to reveal them if they've already seen the film.

To set up our spoiler revealer, we need to add a new element to the news section of the site. Any "secrets" that should be hidden by default will be wrapped in a `span` element with the `class` `spoiler` attached to it:

chapter_02/33_spoiler_revealer/index.html *(excerpt)*

```
Who lost their recording contract today?
<span class='spoiler'>The Zaxntines!</span>
```

Let's break down what our script needs to do: first, we need to hide the answers and add a new element that enables them to be revealed if the user desires. When that element is clicked, we need to disclose the answer. Hiding? Adding? Handling clicks? We know how to do all of that:

chapter_02/33_spoiler_revealer/script.js *(excerpt)*

```
$('.spoiler').hide();
$('<span class="revealer">Tell me!</span>')
  .insertBefore('.spoiler');
$('.revealer').click(function() {
  $(this).hide();
  $(this).next().fadeIn();
});
```

There's a lot going on here, some of it new, but if you read through the lines one at a time, you'll make sense of it. First, we instantly hide all the spoiler elements, and use the `insertBefore` action to add a new button before each of them. At this point the page will display the new "Tell Me!" buttons, and the original spoiler spans will be hidden.

Next, we select the new buttons we just added and attach `click` event handlers to them. When one of the buttons is clicked, we remove the new revealer element (which we find with `$(this)`), and fade in the spoiler element that's next to it. `next` is an action we've yet to cover. It's used for traversing the DOM, and unsurprisingly gives us access to an element's next sibling (that is, the next element inside the same container).

If we look at our modified DOM shown in Figure 2.7, we can see that the spoiler `span` is the next element after the revealer button. The `next` action simply moves our selection to that element. jQuery also gives us access to a `previous` action that moves the selection to the element before the one that's currently selected.

Figure 2.7. The modified DOM

In fact, jQuery has about a dozen different actions you can use to move around the DOM; `previous` and `next` are just two particularly useful ones. We'll discover more

of them as we proceed through the book, or you can consult the jQuery API documentation[2] to see them all.

With the hidden spoiler element now under jQuery's control, we can simply call `fadeIn` to reveal the spoiler with a smooth transition.

Before We Move On

We've covered so much in the initial chapters that you should now be gaining a sense of jQuery's structure and power. With any luck, you've already hatched plans for using it in your current projects. Please do! Whether you're using it to solve a pernicious problem or just to add a bell here and a whistle there, dirtying your hands is by far the best way to cement your knowledge.

One small word of warning—remember the old saying: "When the only tool you have is a hammer, everything looks like a nail." jQuery is a great tool, but may be inappropriate in some instances. If a problem is better solved with simple changes to your CSS or HTML, that's what should be done. Of course, while you're learning, feel free to do *everything* with jQuery; just remember that when the time comes to put your skills into practice, you should always use the best tool for the job.

In the pages that follow, we'll take the simple jQuery building blocks we've learned here and use them to construct some very cool widgets, effects, and user interaction that you can start using immediately.

[2] http://docs.jquery.com/Traversing

Chapter 7

Forms, Controls, and Dialogs

In its infancy, the Web was a read-only medium. Discontent with a nearly infinite collection of linked documents, early web developers wanted more; specifically, they didn't just want people to *read* their web pages about their cats—they wanted them to sign their guest books and tell them how great their cats were. HTML forms gave us a feedback mechanism that would eventually give rise to the enormous and complex web-based applications that we have today.

JavaScript stepped in to help simple HTML form elements emulate many of the more sophisticated and interactive input controls found in desktop applications, but the code has often been unwieldy and bloated. jQuery allows us to simplify control creation and lets us concentrate on turning our ideas into functioning controls quickly and elegantly.

And it's lucky for us that it's quick! Our client is keen to build on the fancy Ajax controls we've built for him. Now that he has his buzzword-compliant features, he concedes that he probably should have first fixed up some of the forms on the site, which now look painfully 1999 in comparison. He wants "some inline editing, fancy form validation messages, cool dialog boxes, and everything—*everything*—should

be drag and droppable, like it's a web site from the future!" Fortunately for us, jQuery lets us build web sites from the future.

Forms

HTML forms are old. And a bit clunky. And browsers vary wildly in how they deal with them. Yet, thanks to JavaScript, these forms have become the basis for some amazingly cool web applications. As always, if JavaScript can do it, jQuery can make it fun!

We know the drill by now: form elements are DOM elements, so jQuery is great at manipulating them. But form elements aren't your typical DOM elements, so there are a handful of special jQuery tricks for dealing with them more specifically. We've seen quite a few of them throughout the book so far—but now it's time to focus on them a little more closely.

Simple Form Validation

Form validation is essential, even if it often seems boring. However, proper, well-designed and implemented forms can make or break how your users perceive your site. Who hasn't had the experience of giving up on a web site because of a particularly frustrating form?

Server-side Form Validation

Client-side form validation with jQuery should *only* be used to assist your users in filling out a form, and should *never* be relied upon to prevent certain types of data being sent to the server. Users with JavaScript disabled will be unhindered by your jQuery validation, so they can submit any values they want. Because of this, if there's any security risk from users submitting malicious data through your forms, that data needs to be thoroughly validated on the server side.

Although jQuery avoids dealing with the nitty-gritty of form validation, it does provide some convenient methods for accessing and setting form values—and that's half the battle! You can select form fields like any other element, but there are some extra filters to make your code more efficient and readable.

The `:input` filter, for example, selects all elements that are inputs, select boxes, textareas, or buttons. You'd use it as you would any filter. Here's how we'd give all of our form elements a lovely lemon chiffon background:

```
$('#myForm:input').css('background-color', 'lemonchiffon')
```

If you want to be more choosy about which elements you're selecting, there are a number of more specific form element filters: `:text`, `:password`, `:radio`, `:checkbox`, `:submit`, `:button`, `:image` (for image buttons), and `:file`. And remember, you're free to apply multiple filters in a single selection.

Furthermore, there are some additional filters that let you select form elements based on their state and value. The `:enabled` and `:disabled` filters will fetch elements based on their `disabled` attribute, and `:checked` and `:selected` help you find radio buttons, select box items, and checkboxes that are checked or selected.

`:checked` and `:selected` in Conditional Logic

These filters are particularly helpful when you need to perform different actions depending on the checked or selected state of a checkbox or radio button. For example, you can check to see if a box is checked with `if($(this).is(':checked'))`.

After you've selected your elements, it's time to find their values so you can validate them against your requirements. We've already used the `val` function enough to know what it does: it returns the value of a form field. We can now perform some simple validation—let's test to see if any text boxes in a form are empty:

chapter_07/01_simple_validation/script.js *(excerpt)*

```
$(':submit').click(function(e) {
  $(':text').each(function() {
    if ($(this).val().length == 0) {
      $(this).css('border', '2px solid red');
    }
  });
  e.preventDefault();
});
```

Fill out one or two of the text inputs, and try submitting the form; any input you leave blank will be highlighted in red.

The `val` action works for select boxes and radio buttons too. As an example, let's alert the radio button value when the user changes the selection:

chapter_07/02_radio_buttons/script.js *(excerpt)*

```
$(':radio[name=sex]').change(function() {
  alert($(this).val());
});
```

This `change` event is fired whenever a value in a form has changed. For checkboxes, select boxes, and radio buttons, this occurs whenever the value changes from its current value. For a text `input` or `textarea`, it fires whenever a user changes the element's value—but only when the focus is moved away from the element. This is a great way to implement some simple inline validation.

Let's revisit our simple validation example, except that this time we'll test for empty fields whenever the user moves to the next field. For this, we'll need to capture the `blur` event, which fires whenever a form field loses focus. This is perfect for inline validation:

chapter_07/03_simple_inline_validation/script.js *(excerpt)*

```
$(':input').blur(function() {
  if ($(this).val().length == 0) {
    $(this)
      .addClass('error')
      .after('<span class="error">This field must … </span>');
  }
});
$(':input').focus(function() {
  $(this)
    .removeClass('error')
    .next('span')
    .remove();
});
```

We're just checking that the fields are filled in, but any type of validation can be implemented in this way. You can check for a minimum or maximum number of

characters, or a specific format using regular expressions, or check that a password confirmation field matches the original password field.

Avoid Over-validating!

One important point to consider when designing form validation: keep it simple! The more rules you add, the more likely you'll have forgotten an edge case, and wind up frustrating some of your users. Offer hints, sample inputs, and guidance, instead of rules that will prevent users from submitting the form if their postal code is formatted differently to what you expected!

The `submit` Event

We also can hook into the `submit` event, which is fired when the form's submitted. This is a better technique than listening for a click event on the submit button, as it will also fire if the user submits the form by pressing the **Enter** key. If you return `false` from the `submit` event handler, the form will not be submitted. In our example below, we'll check all of the text boxes in the form. If any are left empty, we'll pop up a message, and focus on the offending element:

chapter_07/04_submit_event/script.js *(excerpt)*

```
$("form").submit(function() {
  var error = false;
  $(this).find(":text").each(function() {
    if ($(this).val().length == 0) {
      alert("Textboxes must have a value!");
      $(this).focus();
      error = true;
      return false; // Only exits the “each” loop
    }
  });
  if (error) {
    return false;
  }
  return true;
});
```

With all of these raw, form-based tools at your disposal you can easily add validation to your forms on a page-by-page basis. If you plan your forms carefully and develop a consistent naming standard, you can use jQuery to generalize your validation so that it can apply to many forms.

But—as we've already seen—there are an enormous number of edge cases to consider when designing form validation. If you need really bulletproof validation and would rather spend your time designing the user interaction, perhaps you should consider the Validation Plugin.

Form Validation with the Validation Plugin

Building your own inline validation system can be a daunting endeavor; you need to know regular expressions to be able to verify that an email address or phone number is valid, for example. The Validation plugin solves a lot of these problems for you, and lets you add sophisticated and customizable inline validation to most forms with minimal effort.

We'll stop short of going over every option available for use with this plugin here (that would fill a whole chapter!), but we'll look at the most common ones.

Let's start with the form. To illustrate as many of the different validation options, we'll go with a sign-up form that includes password and password confirmation fields:

chapter_07/05_validation_plugin/index.html *(excerpt)*

```
<div id="signup">
  <h2>Sign up</h2>
  <form action="">
    <div>
      <label for="name">Name:</label>
      <input name="name" id="name" type="text"/>
    </div>
    <div>
      <label for="email">Email:</label>
      <input name="email" id="email" type="text"/>
    </div>
    <div>
      <label for="website">Web site URL:</label>
      <input name="website" id="website" type="text" />
    </div>
    <div>
      <label for="password">Password:</label>
      <input name="password" id="password" type="password" />
    </div>
    <div>
```

```
      <label for="passconf">Confirm Password:</label>
      <input name="passconf" id="passconf" type="password" />
    </div>
    <input type="submit" value="Submit!" />
  </form>
</div>
```

To use the Validation Plugin, we simply need to call `validate` on a selection of our form, passing it any options we want to use. The most important option is `rules`, which is where you need to define rules used to validate the users' input:

chapter_07/05_validation_plugin/script.js *(excerpt)*

```
$('#signup form').validate({
  rules: {
    name: {
      required: true,
    },
    email: {
      required: true,
      email: true
    },
    website: {
      url: true
    },
    password: {
      minlength: 6,
      required: true
    },
    passconf: {
      equalTo: "#password"
    }
  },
  success: function(label) {
    label.text('OK!').addClass('valid');
  }
});
```

There are a considerable number of predefined validation rules available, and of course you can define your own. You'll need to consult the documentation to learn about all of them. Here we've used `required`, `email`, `url`, `minlength`, and `equalTo`.

`required` marks a field as required, so it will be flagged as an error if it's empty. `email` and `url` validate the format of the field; emails must contain an @, URLs must begin with http://, and so on. Inside the `rules` object, we define an object for each form field, named after the field's `id`. `minlength` is self-explanatory (and, as you'd expect, there's a corresponding `maxlength`). Finally, `equalTo` allows us to specify a jQuery selector pointing at another form field, the contents of which will be checked against the current field to see if they're the same.

The Validation plugin will add a new `label` element after each form field to contain the error message; by default this will have a `class` of `error`, so you're free to style it in as stern a fashion as you'd like.

By default, the plugin will only display a message if a field's value is invalid. User research has shown, however, that users complete forms more quickly and confidently if they're also provided with feedback for *correct* entries. That's why we're using the `success` callback to set the value of the message `label`, and giving it a `class` to style it with a nice green check mark. `success` is passed the message element itself, so you can manipulate it in any way you'd like. Our sample form is illustrated mid-completion in Figure 7.1.

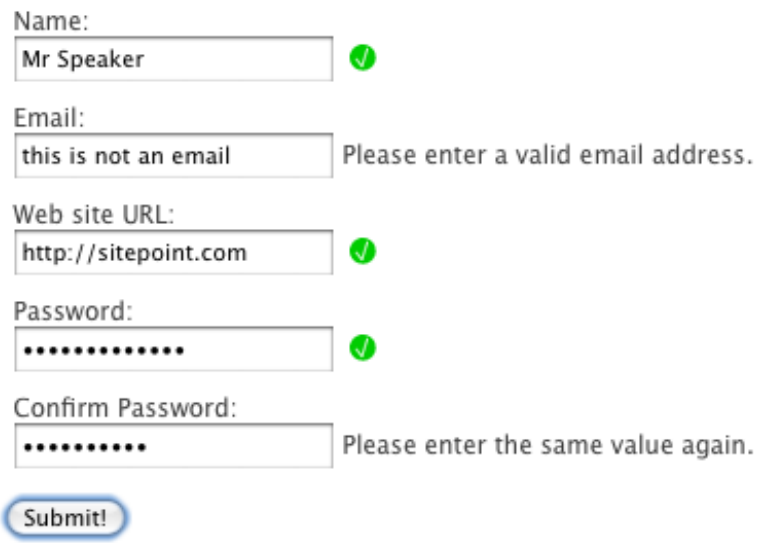

Figure 7.1. Inline validation with the Validation plugin

It's also possible to customize the error messages themselves, and it's worth noting that there are a number of localized variants in the **localization** folder of the plugin directory.

This example is just the beginning of what's possible with the Validation plugin. Make sure you consult the documentation and the examples in the plugin's demo folder to explore all the available features.

Maximum Length Indicator

Our client wants to limit the feedback form content field to 130 characters. "Like Twitter?" you ask. "Why would you want to do that?" He rambles off a spiel about targeted feedback and attention span and … but we know he just wants to copy Twitter. The "remaining characters" count is another feature making a comeback these days, though the idea of setting a limit on the length of input is as old as computers themselves.

By displaying the remaining characters next to the form field, users have clear expectations of how much they can type.

We'll set a `class` of `maxlength` on the `textarea` we want to target with this effect. Then, in our script, we append a `span` after it and add a new kind of event handler:

chapter_07/06_max_length_indicator/script.js *(excerpt)*

```
$('.maxlength')
  .after("<span></span>")
  .next()
  .hide()
  .end()
  .keypress(function(e) {
    // handle key presses;
  });
```

After we append the span, the `textarea` is still the selected element. We want to modify the new `span`, so we move to it with the `next` action. Then we hide the `span`, but now we need to go back to our form element to add an event handler, so we use the `end` action. The `end` action moves the jQuery selection back to where it was before the last time you changed it. In our example, `hide` doesn't change the selection, but `next` does. So when we call `end`, the selection moves back to the state it was in before we called `next`.

Now that we're back on the form element, we attach a `keypress` event handler. As you might expect, this event fires whenever a key is pressed. Here we can check whether another character is still allowed—and prevent the user from adding more characters if it's not:

chapter_07/06_max_length_indicator/script.js *(excerpt)*

```
var current = $(this).val().length;
if (current >= 130) {
  if (e.which != 0 && e.which != 8) {
    e.preventDefault();
  }
}
```

Now comes the meat of the effect: we grab the value of the element and use the JavaScript `length` property to give us its length. If the current number of characters is greater than the maximum length, we'll prevent the key press from registering by using the `preventDefault` action of the event.

When handling a `keypress` event, the event has a `which` property corresponding to the ASCII code of the key pressed. Note that we've allowed the **delete** (ASCII code 0) and **backspace** (ASCII code 8) keys to function regardless of the number of characters. If we didn't do this, the user could paste in a response that exceeded the limit—yet be unable to delete any characters to make it fit:

chapter_07/06_max_length_indicator/script.js *(excerpt)*

```
$(this).next().show().text(130 - current);
```

The last task to do is display the number of remaining characters in the `span` we created. We move to the next element, make sure it's visible, and display the results of our simple calculation to indicate how many more characters are allowed.

Form Hints

A nice trick to decrease the amount of space a form takes up on the page is to move the label for a form field *inside* the input itself. When users move their focus to the field, the label magically vanishes, allowing them to start typing. If they leave the field empty and move away, the original label text appears back in its place.

This technique is only appropriate for short and simple forms. In larger forms, it's too easy for users to lose track of what each particular field is for in the absence of visible labels. This can be a problem if they need to revisit or change values they've already entered.

That said, for simple forms like login or search forms, where most users are very familiar with what each field is for, it can be a great way to save space and streamline your interface. Looking at Figure 7.2, you could probably come up with a good guess of how to implement the effect yourself. The only tricky part is how to return the default value to the input when the user moves on without entering anything into it.

Figure 7.2. Form hints

If you guessed that we'd do it using the `data` action, you'd be correct. We'll store the default value in the data for each clearable item—and if the value is still empty when the user leaves, we'll restore it from there:

chapter_07/07_form_hints/script.js *(excerpt)*

```
$('input.clear').each(function() {
  $(this)
    .data('default', $(this).val())
    .addClass('inactive')
    .focus(function() {
      $(this).removeClass('inactive');
      if ($(this).val() == $(this).data('default') || '') {
        $(this).val('');
      }
    })
    .blur(function() {
      var default_val = $(this).data('default');
      if ($(this).val() == '') {
        $(this).addClass('inactive');
        $(this).val($(this).data('default'));
      }
    });
});
```

We need to go through each element and save the default value when the document loads. Then we keep track of the `focus` and `blur` events that will fire whenever the user moves into or out of our `inputs`. On `focus`, we test if the value of the text box is the same as our default text; if it is, we clear the box in preparation for the user's input.

On the way out, we check to see if the text box is empty, and if it is we put the original value back in. We add and remove a class as we go; this allows us to style the form fields differently when they're displaying the hint. In our example, we've simply made the text color a little lighter.

Check All Checkboxes

With text inputs firmly under our control, it's time to move on to other form controls. We'll start off with a bugbear of StarTrackr's users: there's too much checkbox ticking required when filling in the various celebrity information forms. This is resulting in skewed data, bored users, and inaccurate reports on celebrities. Our client has asked that each category of statistic have a "check all" box, so that the user can toggle all of the checkboxes off or on at once.

Knowing the jQuery form filters makes this task a walk in the park. We just have to select all checkboxes in the same group, and check or clear them. The way we group checkboxes together in HTML forms is by giving all of the related items the same name:

chapter_07/08_check_all/index.html *(excerpt)*

```
<div class="stats">
  <span class="title">Reason for Celebrity</span>
  <input name="reason"
    type="checkbox" value="net" />Famous on the internet<br/>
  <input name="reason"
    type="checkbox" value="crim" />Committed a crime<br />
  <input name="reason"
    type="checkbox" value="model" />Dates a super model<br />
  <input name="reason"
    type="checkbox" value="tv" />Hosts a TV show<br />
  <input name="reason"
    type="checkbox" value="japan" />Big in Japan<br />
  <hr />
  <input class="check-all"
    name="reason" type="checkbox" /><span>Check all</span>
</div>
```

We've given the last checkbox the special `class` of `check-all`. This box will act as our master checkbox: when it is checked or unchecked, our code springs to life. First, we construct a selector string that will select all of the checkboxes with the same name as the master checkbox. This requires gluing a few strings together, to end up creating a selector that looks like `:checkbox[name=reason]`.

We then set all of the related checkboxes to have the same checked value as our master checkbox. Because our code is running after the user has changed the value, the `checked` property will reflect the new state of the checkbox—causing all of the related items to be either selected or deselected accordingly:

chapter_07/08_check_all/script.js *(excerpt)*

```
$('.check-all:checkbox').change(function() {
  var group = ':checkbox[name=' + $(this).attr('name') + ']';
  $(group).attr('checked', $(this).attr('checked'));
});
```

Performance Issues

If your page is large, trawling through every DOM node looking for checkboxes can be slow. If you're noticing pages becoming unresponsive, you might want to investigate the `context` parameter of the jQuery selector, which limits where jQuery will hunt for your selections. We'll cover the `context` parameter in Chapter 8.

Inline Editing

Inline editing (aka edit in place) was one of the first effects that truly showed Ajax's power to create naturally helpful controls. The first time you used an inline edit box you were amazed; every time after that it was unnoticeable—it just worked like it should work.

There are a number of ways you can recreate the effect. The easiest way is to disguise your form fields as labels: remove the borders, give them the same background color as your page, and add borders back in when the users focuses on it! This is a great cheat, and means your form acts just like a regular one (because it is). However, this can be tricky to accomplish, and require a lot of extra markup and styles if you want many different parts of the page to be editable.

As a result, a more common approach is to allow the editing of non-form elements: paragraph tags and title tags, for example. When the user clicks on the tag, the contents are replaced with a text box or `textarea` that the user can interact with. When the task is complete, the original tags are replaced with the new content.

We'll use classes to mark content as being editable. For simple one-liners, we'll use `input` elements (by assigning the class `editable`), and for longer passages we'll use textareas (which we'll give the `class` name `editable-area`). We'll also be sure to assign each element a unique `id`. This is so we can send the data to the server for updating in the database, and reload the new data on the next pageload:

chapter_07/09_inline_editing/index.html *(excerpt)*

```
<h3 id="celeb-143-name" class="editable">Glendatronix</h3>
<p id="celeb-143-intro" class="editable-area">
  Glendatronix floated onto the scene with her incredible debut …
</p>
```

To make it work, we need to capture a few events. The first is the `hover` event, to add an effect so the user can see that the element is editable (we'll go with a tried and tested yellow background color).

We also want to capture the `click` event—which will fire when the user clicks on the editable content—and the `blur` event, which signifies the end of editing:

chapter_07/09_inline_editing/script.js *(excerpt)*

```
$(".editable, .editable-area")
  .hover(function() {
    $(this).toggleClass("over-inline");
  })
  .click(function(e) {
    // Start the inline editing
  }).blur(function(e) {
    // End the inline editing
  });
```

When the user clicks an editable area, our code kicks in. First, we grab a reference to the element that was clicked and store it in the `$editable` variable (to prevent us having to reselect it every time). We'll also check for the `active-inline` `class` with `hasClass`. If the element already has the `active-inline` `class`, it's already an edit box. We'd rather not replace the edit box with another edit box:

chapter_07/09_inline_editing/script.js *(excerpt)*

```
// Start the inline editing
var $editable = $(this);
if ($editable.hasClass('active-inline')) {
  return;
}
```

Next up, we want to grab the contents of the element—and then remove it. To obtain the contents we'll just save the `html` data to a variable … but we'll also use the `$.trim` method to remove whitespace from the start and end of the content string. This is necessary because, depending on how your HTML is laid out, the string could have extra carriage returns and line spaces that we want to prevent showing up in the text box.

Then we add our `active class`, which serves the dual purpose of indicating that editing is in process, and providing a hook for our styles. Finally, we clear out the current element with the `empty` action. This command is similar to `remove`, except that calling `empty` on an element will result in all of its *children* being removed, rather than the element itself:

chapter_07/09_inline_editing/script.js *(excerpt)*

```
var contents = $.trim($editable.html());
$editable
  .addClass("active-inline")
  .empty();
```

Chaining with `empty` and `remove`

It's important to remember that any jQuery actions you chain after a `remove` or `empty` command will be applied to the removed selection and *not* the selection that you had before you removed the elements. The reasoning behind this is that if you simply threw away the elements, they'd be lost forever. This way you have the option to keep them around, process them, or store them for future use.

Finally, it's time to insert our brand-new text box or `textarea`. We will check for the `editable class` to determine which kind of form element we need to append (remember that we indicated multiline content with `editable-area`). We set the new element's value with the contents of the elements we removed, and append it to the target element:

chapter_07/09_inline_editing/script.js *(excerpt)*

```
// Determine what kind of form element we need
var editElement = $editable.hasClass('editable') ?
  '<input type="text" />' : '<textarea></textarea>';

// Replace the target with the form element
$(editElement)
  .val(contents)
  .appendTo($editable)
  .focus()
  .blur(function(e) {
    $editable.trigger('blur');
  });
```

You might be curious about the use of the `trigger` function. It's simply a different way to cause an event to fire (so, in this example, we could also have used the `$editable.blur()` syntax we've already seen). The trigger action is more flexible than its shorter counterpart—but for now we'll just stick with the basic usage.

`trigger` is being used in this example for clarity: to show whoever is reading the code that we want to manually fire an event. In this case we're just passing on the event; the `input` was blurred, so we tell the original element that it's time to finish editing. We could manage all of this inside the `input` box's `blur` event handler, but by delegating the event like this, we avoid nesting our code another level (which would make it harder to read). It also makes sense to let the original element deal with its own logic.

The counterpart to `trigger` is `bind`. `bind` lets us add event handlers to an object. Sound familiar? So far we've been binding events by using shorthand convenience methods like `click`, `hover`, `ready`, and so on. But if you pop the hood, you'll see that internally they all rely on `bind`.

The `bind` action takes a string containing the name of the event to bind, and a callback function to run. You can also bind multiple events to an item in a single call by wrapping them in an object. For example, our code attached three separate events to `.editable` and `.editable-area` elements: `click`, `hover`, and `blur`. If you wanted to, you could rewrite that with the `bind` syntax:

```
$('.editable, .editable-area').bind({
  hover: function(e) {
    // Hover event handler
  },
  click: function(e) {
    // Click event handler
  },
  blur: function(e) {
    // Blur event handler
  }
});
```

Let's return to our example; with the editing over, we can go back to our default state. We'll grab the value from the form element, and send it to the server with `$.post`, putting a "Saving …" message in place as we do so. When the `POST` is done, we eliminate the message and replace it with the updated value. As with the Ajax

functionality we saw in the previous chapter, we're faking a server-side response with an empty **save** file. In a real world application, you'd want to check that the changes had actually been saved to the database by checking the response data:

chapter_07/09_inline_editing/script.js *(excerpt)*

```
.blur(function(e) {
  // end the inline editing
  var $editable = $(this);

  var contents = $editable.find(':first-child:input').val();
  $editable
    .contents()
    .replaceWith('<em class="ajax">Saving … </em>');

  // post the new value to the server along with its ID
  $.post('save',
    {id: $editable.attr('id'), value: contents},
    function(data) {
      $editable
        .removeClass('active-inline')
        .contents()
        .replaceWith(contents);
    });
});
```

There are two new jQuery functions in this block of code, but both of them are fairly self-explanatory. `contents()` returns the entire contents of a DOM node, which can include other DOM elements and/or raw text, and `replaceWith()` swaps whatever you've selected with whatever you pass to it. Be careful when using the latter method; in our example we know that `contents()` will only return one element—but if it returned multiple elements, each of those elements would be replaced with the same loading message!

Autocomplete

We've appeased the client—he's having a blast playing with the inline edit fields over in the corner. While we have a few minutes up our sleeves until his next request, let's really impress him by having the "last known whereabouts" field of the celebrity form autocomplete from a list of major cities. The resulting functionality is illustrated in Figure 7.3.

We'll use the Autocomplete plugin from the jQuery plugin repository. It's a full-featured and stable plugin that provides exactly the functionality we need, with minimum weight.

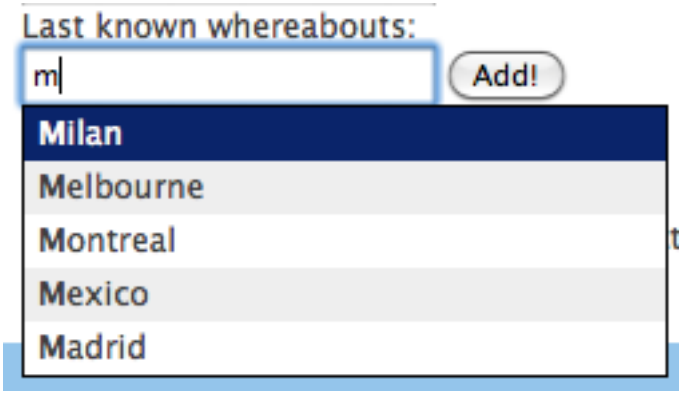

Figure 7.3. Autocompleting "last known whereabouts" field

Firstly, we need the plugin. Head over to the repository and grab it,[1] have a quick look at the examples, then include it your page.

We'll also need to set some CSS styles. There's an example CSS file included with the plugin, so you can gain some idea of the `classes` that are added. We've used several of these styles to give our drop-down suggestion list a standard appearance.

The Autocomplete plugin attaches itself to a `select` box. We're applying it to the location field in our simple form:

chapter_07/10_autocomplete/index.html *(excerpt)*

```
<label for="location">Last known whereabouts:</label>
<input type="text" id="location"/>
```

Now let's see what the Autocomplete plugin can do for us. By default, it requires a local collection of data stored in an array; this is perfect for us, as we want to source our data from an HTML list on the page:

[1] http://docs.jquery.com/Plugins/Autocomplete

chapter_07/10_autocomplete/script.js *(excerpt)*

```
var cities = ['New York', 'Melbourne', 'Montreal', 'London' … ];
$('#location').autocomplete(cities,{
  autoFill: true,
  selectFirst: true,
  width: '240px'
});
```

We've simply passed in a JavaScript array, but Autocomplete also allows us to pass in a URL, in which case it will retrieve the list of potential values via Ajax. Autocomplete will expect a plain-text return comprising one value per line, which should be easy to obtain after a quick chat with your back-end developers!

The above code is enough to get it up and running, but we can also specify a bunch of options. `autoFill` gives us a nice type-ahead effect (filling out the text box with the currently suggested completion), `matchContains` will cause it to match substrings of words, rather than just the first letters, and so on. There's a lot you can fine-tune, so it's worth having a quick study of the options list.

The Autocomplete plugin also fires the `result` event when the user chooses an option. It will give us the name of the tag that was selected as the second parameter passed to our event handler (after the event object). For example, this would alert the selected option when it's selected:

```
$('#location')
  .autocomplete(cities)
  .result(function(event, selection) {
    alert(selection);
  });
```

Very simple, but very funky. And the client is still playing with the last toy we built for him! Perhaps we're a bit too good at playing with form elements, and better return to the to-do list!

Star Rating Control

Building a large celebrity-stalking community is our client's primary goal; he's starting to realize that the users of his site are becoming his product—a product he can start to sell to advertisers. Keen to explore this possibility, he wants to increase

user engagement, and help his users feel important. He has to look after his product, after all. We've thought about this a bit, and tossed him a star rating idea—after all, people love nothing more than to express their feelings through the assignment of gold stars. Our control will appear as shown in Figure 7.4.

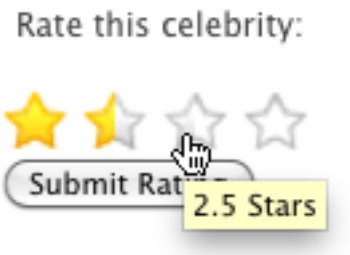

Figure 7.4. Star rating control

The basis for our star rating control is a radio button group; it's perfect, as the browser enforces a single selection from the group. You can select the range that you want the user to choose from, simply by adding the correct number of buttons:

chapter_07/11_star_ratings/index.html *(excerpt)*

```
<div class="stars">
  <label><input id="rating-1" name="rating" type="radio" value="1"/>
➥1 Star</label>
  <label><input id="rating-2" name="rating" type="radio" value="2"/>
➥2 Stars</label>
  <label><input id="rating-3" name="rating" type="radio" value="3"/>
➥3 Stars</label>
  <label><input id="rating-4" name="rating" type="radio" value="4"/>
➥2 Stars</label>
</div>
```

The hard part, of course, is replacing these radio buttons with our star control. You can try to grapple with styling the HTML controls with only CSS, but it will be much easier and more flexible if you split the control into two parts: the underlying model that stores the data, and the shiny view with stars. The model, in this case, is the original HTML radio button group. Our plan of attack is to hide the radio buttons, and display a list of links that we've added via jQuery, styled to look like stars. Interacting with the links will switch the selected radio button. Users without JavaScript will simply see the radio buttons themselves, which is fine by us.

For the stars themselves, we will again rely on CSS sprites. This way our control will only be reliant on a single image (shown in Figure 7.5), which saves on HTTP requests and makes it easier for our graphic designers to edit.

Figure 7.5. Star CSS sprite image

Our CSS will apply the CSS sprite image to the links we create that represent half-stars. To move between the different image states, we just need to update the `background-position` property:

chapter_07/11_star_ratings/stars.css *(excerpt)*

```
.stars div a {
  background: transparent url(../../css/images/sprite_rate.png)
➥0 0 no-repeat;
  display: inline-block;
  height: 23px;
  width: 12px;
  text-indent: -999em;
  overflow: hidden;
}

.stars a.rating-right {
  background-position: 0 -23px;
  padding-right: 6px;
}

.stars a.rating-over { background-position: 0 -46px; }

.stars a.rating-over.rating-right { background-position: 0 -69px; }
```

```
.stars a.rating { background-position: 0 -92px; }

.stars a.rating.rating-right { background-position: 0 -115px; }
```

We've decided to make the user select a rating out of four stars, rather than the usual five. Why? User psychology! Offer a person a middle road and they'll take it; by having no middle we make the users think a bit more about their selection. We achieve better results, and we'll be better able to present users with the best content (as chosen by them)!

But four is a limited scale—that's why we want to allow for half-star ratings. This is implemented via an optical illusion—you probably noticed that our star images are chopped in half. Our HTML will contain eight radio buttons, and they'll each be worth half a star. There's two parts to the code for transforming our eight radio buttons into four stars. First, the `createStars` function will take a container with radio buttons and replace it with star links. Each star will then be supplemented with the proper event handlers (in the `addHandlers` method) to let the user interact with the control. Here's the skeleton of our `starRating` object:

chapter_07/11_star_ratings/script.js *(excerpt)*

```
var starRating = {
  create: function(selector) {
    $(selector).each(function() {
      // Hide radio buttons and add star links
    });
  },

  addHandlers: function(item) {
    $(item).click(function(e) {
      // Handle star click
    })
    .hover(function() {
      // Handle star hover over
    },function() {
      // Handle star hover out
    });
  }
}
```

The `create` method iterates through each container matching the selector we pass in, and creates a list of links that act as a proxy for the radio buttons. These links are what we'll style to look like stars. It will also hide the original form elements, so the user only sees our lovely stars:

chapter_07/11_star_ratings/script.js *(excerpt)*

```
$(selector).each(function() {
  var $list = $('<div></div>');
  // loop over every radio button in each container
  $(this)
    .find('input:radio')
    .each(function(i) {
      var rating = $(this).parent().text();
      var $item = $('<a href="#"></a>')
        .attr('title', rating)
        .addClass(i % 2 == 1 ? 'rating-right' : '')
        .text(rating);

      starRating.addHandlers($item);
      $list.append($item);

      if ($(this).is(':checked')) {
        $item.prevAll().andSelf().addClass('rating');
      }
    });
```

We start by creating a container for the new links (a `div` element); we'll be creating one new link for each of the radio buttons we're replacing. After selecting all the radio buttons with the `:radio` selector filter, we take each item's rating and use it to create a hyperlink element.

Conditional Assignment with Modulus

For the `addClass` action, we're specifying the class name conditionally with a ternary operator (see the section called "Fading Slideshow" in Chapter 4), based on a bit of math. As we've done earlier in the book, we're using the modulus (`%`) operator to determine whether the index is even or odd. If the index is odd, we'll add the `rating-right class`; this makes the link look like the right side of a star.

Once our link is ready, we pass it to the `addHandlers` method—this is where we'll attach the events it needs to work. Then, we append it to the list container. Once it's in the container, we see if the current radio button is selected (we use the `:checked` form filter). If it's checked, we want to add the `rating` `class` to this half-star, and to all of the half-stars before it. Any link with the `rating` `class` attached will be assigned the gold star image (which will allow users to see the current rating).

To select all of the elements we need to turn gold, we're going to enlist the help of a couple of new jQuery actions: `prevAll` and `andSelf`. The `prevAll` action selects *every* sibling before the current selection (unlike the `prev` action, which only selects the immediately previous sibling). For our example, we want to add the `class` to the previous siblings *and* the current selection. To do so, we apply the `andSelf` action, which simply includes the original selection in the current selection. Now we have all of the links that will be gold, so we can add the `class`.

Other Traversal Methods

You might have guessed that, along with `prevAll`, jQuery provides us with a `nextAll` method, which grabs all of an element's siblings occurring *after* it in the same container. jQuery 1.4 has also introduced two new companion methods: `prevUntil` and `nextUntil`. These are called with a selector, and will scan an element's siblings (forwards or backwards, depending on which one you're using) until they hit an element that matches the selector.

So, for example, `$('h2:first').nextUntil('h2');` will give you all the elements sitting between the first `h2` on the page and the following `h2` in the same container element.

If you pass a second parameter to `prevUntil` or `nextUntil`, it will be used as a selector to filter the returned elements. So, for example, if we had written `nextUntil('h2', 'div')`, it would only return `div` elements between our current selection and the next `h2`.

After doing all this hard work, we can now add the new list of links to the control, and get rid of the original buttons. Now the user will only interact with the stars:

chapter_07/11_star_ratings/script.js *(excerpt)*

```
// Hide the original radio buttons
$(this).append($list).find('input:radio').hide();
```

The control looks like it's taking shape now—but it doesn't do anything yet. We need to attach some event handlers and add some behavior. We're interested in three events. When users hover over a star, we want to update the CSS sprite to show the hover state; when they move away, we want to revert the CSS sprite to its original state; and when they click, we want to make the selection gold:

chapter_07/11_star_ratings/script.js *(excerpt)*

```
$(item).click(function(e) {
  // React to star click
})
.hover(function() {
  $(this).prevAll().andSelf().addClass('rating-over');
},function() {
  $(this).siblings().andSelf().removeClass('rating-over');
});
```

The `hover` function is the easiest: when hovering over, we select all of the half-stars before—including the current half-star—and give them the `rating-over` `class` using `prevAll` and `andSelf`, just like we did in the setup. When hovering out, we cover our bases and remove the `rating-over` class from all of the links. That's hovering taken care of.

Now for the click:

chapter_07/11_star_ratings/script.js *(excerpt)*

```
// Handle Star click
var $star = $(this);
var $allLinks = $(this).parent();

// Set the radio button value
$allLinks
  .parent()
  .find('input:radio[value=' + $star.text() + ']')
  .attr('checked', true);

// Set the ratings
$allLinks.children().removeClass('rating');
$star.prevAll().andSelf().addClass('rating');

// prevent default link click
e.preventDefault();
```

The important part of handling the `click` event is to update the underlying radio button model. We do this by selecting the correct radio button with the `:radio` filter and an attribute selector, which searches for the radio button whose value matches the current link's text.

With the model updated, we can return to messing around with the CSS sprites. First, we clear the `rating class` from any links that currently have it, then add it to all of the links on and before the one the user selected. The last touch is to cancel the link's default action, so clicking the star doesn't cause it to fire a location change.

Controls

That takes care of our client's primary concern: form usability. Now we can start doing some of the really fun stuff. jQuery and jQuery UI are the perfect tools for moving beyond the primitive HTML form controls we all know and accept. Once we leave the stuffy confines of the Web's ancient history behind, we find that the ability to create amazing new controls is limited only by our imagination. After all, there should be more ways to interact with a web site than entering some text in a box!

Date Picker

Our client wants to add a "CelebSpotter" section to the site, where his users will be able to report celebrity spottings. Of course, they'll need to report the date and time of the spotting. Early tests of this functionality showed that users were often confused by the date format they were required to enter. This problem was partially offset by adding sample data and format hinting, but the client wants to take it further and add a fancy date picker to the form.

If you've ever sat down and created a reasonably functional date picker in JavaScript, you'd be inclined to avoid ever doing it again. It's a lot of hard work for a control that's, in the end, just a date picker. Mind you, date pickers are crucially important controls that can be insanely frustrating when done wrong. The problem is that because they're so involved, there are a lot of places for them to go wrong. Fortunately for our sanity, jQuery UI contains a highly customizable and fully-featured date picker control that lets us avoid many of the potential pitfalls of building one ourselves. An example of this control is shown in Figure 7.6.

Figure 7.6. jQuery UI date picker control

We'll start with the input field currently being used for the date:

chapter_07/12_date_picker/index.html *(excerpt)*

```
<input type="text" id="date" />
```

If you're just looking for the basic date picker, the jQuery code can be no more complicated than a single line:

```
$("#date").datepicker();
```

The date picker is triggered when the `input` box receives `focus`, and slides into view with the current month and day selected. When the text box loses `focus`, or when a date is selected, it disappears. Sure, it looks very nice, and works with the jQuery smoothness we expect—but what does it offer us over and beyond competing date pickers? (Remember, just because you're using jQuery, it doesn't mean you should ignore other suitable JavaScript components.)

The date picker component in jQuery UI is feature-packed. Packed! It is fully localizable, can handle any date formats, lets you display multiple months at once, has a nifty date range mechanism, allows configurable buttons, is keyboard navigable (you can move around with **ctrl** + arrow keys), and more.

All told, there are over 50 options and events available to you to tweak—almost every tiny aspect of the date picker! To make the calendar you see in Figure 7.6, we've used just a few of them:

chapter_07/12_date_picker/script.js *(excerpt)*

```
$('#date').datepicker({
  showOn: 'button',
  buttonText: 'Choose a date',
  buttonImage: 'calendar.png',
  buttonImageOnly: true,
  numberOfMonths: 2,
  maxDate: '0d',
  minDate: '-1m -1w',
  showButtonPanel: true
});
```

The `showOn` lets us choose when the calendar will pop up. The available options are `'focus'` (when the text box receives focus), `'button'` (a button is added next to the text box, which users can click to open the calendar), or `'both'` (which allows for both options). To use an icon for the button, we've specified a `buttonImage`. We also set `buttonImageOnly` to `true`; this means that only the image will be displayed, rather than a standard form `button`.

Next up, we've set the `numberOfMonths` to `2`—this means the user will see two months worth of days at the same time. You can even specify an array instead of an integer; for example, `[3, 3]` will show a 3x3 grid of months!

The `maxDate` and `minDate` options let you set the range within which the user can select a date. You can specify a JavaScript date object, or you can use a string to dictate relative dates. The latter option is usually easier, and that's what we've done here. We've set the `maxDate` as `0`—which means today's date. The `minDate` we've set as -1m -1w so the user can only select a date that is up to one month and one week in the past. You can plus or minus as much time as you need: `y` for year, `m` for month, `w` for week, and `d` for day.

Date Validation

You may have set a maximum date for the date picker, but users are still able to select a date outside of that range—they can enter it into the text box manually. If you must ensure that dates are within a given range, you need to be performing validation on the server side! The date ranges you specify in the date picker options are to assist your users in picking valid options; that way, they avoid submitting a form that contains frustrating errors.

Date Picker Utilities

The jQuery UI library also provides a few date picker utilities for globally configuring the date pickers, as well as making it easy to play with dates.

The `$.datepicker.setDefaults` method accepts an object made up of date picker settings. Any settings that you specify will be applied to all date pickers on the page (unless you manually override the defaults). For example, if you want every date picker to show two months at a time:

```
$.datepicker.setDefaults({
  numberOfMonths: 2
});
```

The remaining utility functions are for manipulating or assisting with dates and date formats. The `$.datepicker.iso8601Week` function accepts a date and returns the week of the year it's in, from 1 to 53. The `$.datepicker.parseDate` function extracts a date from a given string; you need to pass it a string and a date format (for example, `"mm-dd-yy"`), and you'll receive a JavaScript date object back. Finally, the `$.datepicker.formatDate` does the opposite. It will format a date object based according to the format you specify—which is great for displaying dates on screen.

Sliders

Our client wants his visitors to be able to find the celebrities they're looking for quickly and easily. He also recognizes that many of his clients will be looking for celebrities whose location information falls in a particular price range, so he wants us to add a price range filter to the site. This is a perfect opportunity to introduce another great jQuery UI component: `slider`!

We'll start with a basic form, consisting of two `select` boxes: one for the maximum price, and one for the minimum price. Then we'll call on jQuery UI to add a fancy slider to control the values of those boxes. The end result is illustrated in Figure 7.7.

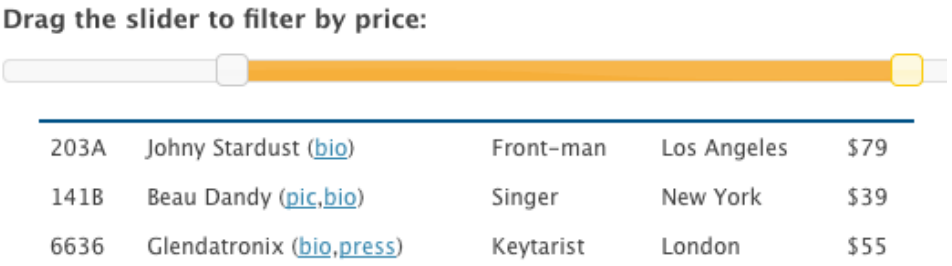

Figure 7.7. A jQuery UI slider

Let's have a look at the basic markup:

chapter_07/13_sliders/index.html *(excerpt)*

```
<div id="price-range">
  <form>
    <label for="min">Minimum Price:</label>
    <select id="min">
      <option value="0">0</option>
      <option value="10">10</option>
      <option value="20">20</option>
      ⋮
      <option value="80">80</option>
      <option value="90">90</option>
    </select>
    <br/>
    <label for="max">Maximum Price:</label>
    <select id="max">
      <option value="10">10</option>
      <option value="20">20</option>
      <option value="30">30</option>
      ⋮
      <option value="100">100</option>
    </select>
  </form>
</div>
```

Now for a look at the code. When the page loads, we first grab the current maximum and minimum values in the `select` boxes. Then we initiate our slider by calling the `slider` method on a newly created empty `div`:

chapter_07/13_sliders/script.js *(excerpt)*

```
var max = $('#max').val();
var min = $('#min').val();

var rangeSlider = $('<div></div>')
  .slider({
    min: 0,
    max: 100,
    step: 10,
    values: [min, max],
    range: true,
    animate: true,
    slide: function(e,ui) {
      $('#min')
        .val(ui.values[0]);
      $('#max')
        .val(ui.values[1]);
      showCelebs();
    }
  })
  .before('<h3>Drag the slider to filter by price:</h3>');

$('#price-range').after(rangeSlider).hide();
```

Whoa! That's a lot of options. Let's see if we can break them down: `min` and `max` are the minimum and maximum values of the slider, respectively. `step` is the amount by which the slider increments. `values` is used for specifying the default value of the slider. Because we've specified an array of two values, the slider bar will have two handles, each with a separate value. Here we're using the values from the `select` lists that we grabbed earlier, so that the slider will always match up with the data in those boxes.

`range` and `animate` are helpful options when creating a slider with more than one handle, as we're doing here: `range` indicates that the area between the handles should be styled differently, usually with a shadow or a different color. This option can also be set to `min` (in which case the area between the minimum and the first handle will be shaded) or `max` (which will shade the area between the last handle and the maximum). `animate` simply tells jQuery to animate the handle's position smoothly if the user clicks elsewhere on the bar, rather than simply jumping there.

Finally, `slide` allows you to specify an event handler that will run whenever the user moves the handles on the slider. The event handler can accept an optional `ui` parameter that allows you to access some of the slider's properties; here we're using the `values` property to adjust the values of our select boxes. We also call `showCelebs`, a custom method in which we'll show or hide celebrities, depending on whether their prices fall within the chosen range.

It's also possible to capture the `change` event, which is very similar to the `slide` event, except that it will also fire if the slider's values are modified programmatically (`slide` only fires when the user interacts directly with the slider).

The jQuery UI slider component will create a horizontal slider by default—but if you want a vertical one you can specify `orientation: 'vertical'`.

We've used `before` and `after` to add a title to our slider and affix it to the page, and we've also hidden the original `select` boxes. Try this now, and you'll see a nicely themed slider that you can play with, and which will adjust the values of the `select` boxes.

In order to make it filter the celebrities, we simply need to implement the `showCelebs` method:

chapter_07/13_sliders/script.js *(excerpt)*

```
function showCelebs() {
  var min = $('#min').val();
  var max = $('#max').val();
  $('.data tr').each(function() {
    var price = parseInt($(this).find('td:last').text().
➥substring(1));
    if (price >= min && price <= max) {
      $(this).fadeIn();
    } else {
      $(this).fadeOut();
    }
  });
}
```

We extract the values of the select boxes, then cycle through each row in the celebrities table, and hide or show it depending on whether or not the price is within the selected range. The only tricky part here is a bit of JavaScript string

processing, required to extract the price from the row text; we use `substring(1)` to extract everything from the first character on (which will conveniently strip the prices of their dollar signs). Then we use `parseInt` to turn the string into a number.

We'll also call `showCelebs` on document-ready, so that the list is prefiltered based on the default values in the form.

This works entirely as expected, and allows users to easily and visually filter celebrities based on their desired price range. Sliders are a great UI widget precisely because they're so intuitive: users will know how to use them without being told. You can probably come up with a few other controls that could benefit from being *sliderized*!

Drag and Drop

Dragging and dropping is coming of age. It's always been there in the background, but has felt out of place (and therefore detrimental to a good user experience) next to the mundane text boxes and radio buttons that make up a typical form. But that was the olden days, with olden day forms. Today, if done well, drag and drop can augment forms in a highly usable way, providing a more natural experience that increases productivity. It also supplies a dash of coolness.

If there's one task that even beginner computer users know how to do, it's to drag an item to the trash. The metaphor is very satisfying—if you don't want it, throw it away! On the other hand, the standard web approach—click the checkbox and press delete—is also well known, but far less satisfying. Our client doesn't want to click checkboxes; he wants to drag stuff to their doom, and have them literally disappear in a puff of smoke to show that it's truly been destroyed.

Figure 7.8 shows an image thumbnail in mid-destruction. The user has selected a photo and dragged it out of the grid and into the trash. The grid of photos is nothing more than a set of `img` tags. You can choose any type of element to be draggable, just as long as you can make it look pretty and work well for your users. A nice touch is to set `cursor: move` on the draggable elements—that way users will see the "grabby hand" icon and know they can drag it.

Figure 7.8. Drag and destroy

As always, we'll start with the markup:

chapter_07/14_drag_drop/index.html *(excerpt)*

```
<div class="trash">
  <img src="trash.png" alt="trash"/>
  <span id="trash-title">Drag images here to delete</span>
</div>
<div id="photo-grid">
  <img src="../../images/fader_100.jpg" />
  <img src="../../images/beau_100.jpg" />
</div>
```

Progressive Enhancement

For the sake of illustration, we're including the `.trash div` in the markup here. However, this poses a problem for users with JavaScript disabled: they'll see a trash area, but will be unable to do anything with it! In a real-world app, you'd want to start with a fully functional, HTML form-based interface for deleting images (or whatever it is you intend to use drag and drop for). Then, you'd use all the methods we've seen throughout the book to remove all those interface elements from the page, and replace them with the above drag and drop markup.

Drag and drop can be a real pain to make work across browsers. Instead of reinventing the wheel, we'll look to our trusted jQuery companion, jQuery UI. It provides a couple of very handy interaction helpers—`draggable` and `droppable`—to handle smooth cross-browser drag and drop.

No Theme Required!

You'll need to include the jQuery UI library in your page as we've covered before, but this time no CSS theme file is required; `draggable` and `droppable` are behaviors, with no preset styling necessary. They do, however, give you some quite handy `class` names to apply your own styles to, which we'll be looking at very shortly.

Let's sketch out the basic structure of our interaction code:

chapter_07/14_drag_drop/script.js *(excerpt)*

```
$(document).ready(function() {
  $('#photo-grid > div').draggable({
    revert: 'invalid'
  });
  $('.trash').droppable({
    activeClass: 'highlight',
    hoverClass: 'highlight-accept',
    drop: function(event, ui) {
      puffRemove($(ui.draggable));
    }
  });
});

function puffRemove(which) {
  // Implement the "puff-of-smoke" effect
}
```

This is the skeleton of our interaction. There's still a lot we need to do to achieve a nice "puff" animation—but, incredibly, that's everything we need for drag and drop! Let's take a closer look at what jQuery UI has given us.

`draggable`

The `draggable` interaction helper makes whatever you select draggable with the mouse. Try this out for size: `$('p').draggable()`. It can make every `<p>` tag on the page draggable! Test it out—it's a lot of fun. Naturally, there are stacks of options and events to customize the behavior. Here are some of the more helpful ones:

```
$('p').draggable({axis: 'y', containment: 'parent'});
```

The `axis` option restricts the object to be draggable along either the X or Y axis only, and the `containment` option confines the object to a bounding box; acceptable values are `'parent'`, `'document'`, and `'window'` (to stay within the respective DOM elements), or an array of values to specify pixel boundaries in the form `[x1, y1, x2, y2]`. You can also use the `grid` option to confine dragging to a grid, by specifying a two element array (for example, `grid:[20,20]`).

Let's look at another example:

```
$('#dragIt').draggable({
  handle: 'p:first',
  opacity: 0.5,
  helper: 'clone'
});
```

For this next bunch of options, we're operating on a `div` called `dragIt`, which contains at least one `<p>` tag. We use the `handle` option to designate the first `p` element as the "handle" users can use to drag the draggable element around. We also specify the `helper` option, which allows you to specify an element to represent the node being dragged around. In this case we've set this option to `clone`. This causes the element to be duplicated, so that the original element will stay in place until you've finished dragging. The `opacity` applies to the helper element.

The other option worth noting is `revert`. If you set this to `invalid` (as we did in our photo dragging example), the element you drag will spring back to its original position if you drop it outside of a droppable target area.

There are also three events you can catch—`start`, `stop`, and `drag`—that fire when you start dragging, stop dragging, and are in mid-drag respectively. In our example we only need to react to drop, but you can easily conceive of situations where the other two events could be put to good use.

`droppable`

The Bonnie to `draggable`'s Clyde is the `droppable` behavior. Droppable elements are targets for draggable items. A droppable element has far fewer options than a draggable element; we've used the most important, `activeClass` and `hoverClass`, above. The `activeClass` is added to the droppable element when a draggable item is being dragged. Similarly, the `hoverClass` is added when a draggable item is hovering over the droppable element.

You can also specify a selector for the `accept` option, which restricts the draggables that can be dropped. This lets you have multiple drop points, where only certain draggable items can go. This can be great for list manipulation.

The events for a droppable element are similar to draggables. Instead of `start`, `stop`, and `drag` we have `over`, `out`, and `drop`. In our photo grid example, we've used the `drop` event to know when to destroy the draggable item.

Both the `draggable` and `droppable` behaviors are complex beasts. Once you're over the thrill of how easy they are to implement, you should have a further read through the advanced options in the documentation.

The "Puff" Effect

With dragging and dropping all taken care of, you can walk away knowing you've created a powerful yet cool control with just a few lines of code. But with all that time we saved by using the existing drag and drop functionality, rather than writing it ourselves, we might as well make this a little more impressive—and add the "puff of smoke" as the image is removed.

Instead of using jQuery's `animate` function, we'll need to roll our own animation solution. This is because we need to cycle through image frames—like creating cartoons. To do this we'll use a PNG image that has five same-sized frames of animation all stacked on top of each other, and then offset the image to show the correct frame. This means we'll need to change the position of the image in discrete chunks. If we were to use `animate` instead, it would change the background position gradually, resulting in chopped-off images halfway between frames:

chapter_07/14_drag_drop/script.js *(excerpt)*

```
// Implement the "puff-of-smoke" effect
var $this = $(which);
var image_width = 128;
var frame_count = 5;
```

To start off, we'll store our selection and set up a couple of constants. The `image_width` is the width in pixels of the animation image. The `frame_count` is the total number of frames in the animation (the total height of the image, therefore, should be `image_width * frame_count`). Of course, these will always be the same in our example, but this way, if you ever want to use a different animation image,

you can find the numbers you need to change right at the top of the script, instead of hunting through it to change them in multiple places.

We then set up a container to house the image. The container will be exactly the same size, and in exactly the same place as the element we're deleting:

chapter_07/14_drag_drop/script.js *(excerpt)*

```
// Create container
var $puff = $('<div class="puff"></div>')
  .css({
    height: $this.outerHeight(),
    left: $this.offset().left,
    top: $this.offset().top,
    width: $this.outerWidth(),
    position: 'absolute',
    overflow: 'hidden'
  })
  .appendTo('body');
```

With the container in place we can now append the animation image to it. Because the container has its `overflow` set to `hidden`, only a single frame of the image will ever be seen. To make the image fit the container (which is the same size as the element we're deleting), we need to scale it to fit. The scale is determined by dividing the width of the container by the width of the image:

chapter_07/14_drag_drop/script.js *(excerpt)*

```
var scale_factor = $this.outerWidth() / image_width;
var $image = $('<img class="puff" src="epuff.png"/>')
  .css({
    width: image_width * scale_factor,
    height: (frame_count * image_width) * scale_factor
  })
  .data('count', frame_count)
  .appendTo($puff);
```

Preloading the Image

If you have a lot of frames in your animation image, it could wind up being a fairly large file and take a while to load. If your user deletes an element before the image has loaded, the animation will be unable to display. A trick for preloading the image is to load it into a jQuery selector in the document-ready function: `$('<img src="puff.png"/>');`. This will load the image without displaying it, so it will be ready for your animation.

We also add a `count` property to the image via the `data` action. This contains the total number of frames left to show. With all of this in place, we can go ahead and delete the original element that was dropped:

chapter_07/14_drag_drop/script.js *(excerpt)*

```
// Remove the original element
$this.animate({
  opacity: 0
}, 'fast').remove();
```

While that's fading out, we want to initiate the animation. This is going to require a small amount of JavaScript-fu; we're going to set up a self-contained, self-executing loop that plays the animation through once:

chapter_07/14_drag_drop/script.js *(excerpt)*

```
// Animate the puff of smoke
(function animate() { 1

  var count = $image.data('count'); 2

  if (count) { 3
    var top = frame_count - count;
    var height = $image.height() / frame_count;
    $image.css({
      "top": - (top * height),
      'position': 'absolute'
    });
    $puff.css({
      'height': height
    })
    $image.data("count", count - 1); 4
```

```
    setTimeout(animate, 75); ❺
  } else {
    $image.parent().remove(); ❻
  }
})();
```

Inside this function, we're executing the animation. Here are the highlights:

❶ We've wrapped the function in the `(function myFunction(){})()` construct, which is a way to create and execute an anonymous function that can nonetheless refer to itself by name. This is an odd JavaScript construct, and one that you needn't worry about understanding completely; in this case it's handy as it allows us to create a self-contained piece of functionality that can call itself (this will be useful when we use the `setTimeout` method).

❷ We find out which frame we're up to by checking the `count` data.

❸ If there are still frames left to display, we calculate the offset of the image and move the correct frame into view. (We can use `if (count)` in this way because in JavaScript, the number `0` is equivalent to `false`.)

❹ We decrease the frame count so that the next time the loop runs it will display the next frame in the series.

❺ Finally, we call `setTimeout`, specifying our anonymous function as the callback. This way, after 75 milliseconds, the whole process will run again.

❻ When the count reaches zero and the animation concludes, we remove the puff container from the DOM.

Try it out. Drag an item to the trash, and watch it vanish in a cloud of smoke!

jQuery UI `sortable`

Another great feature of jQuery UI is the `sortable` behavior. An element that you declare as `sortable` becomes a droppable target to its children—and the children all become draggable. The result is that you can reorder the children as you see fit. While sortable allows us to order items within a container, it doesn't actually sort anything: the sorting is up to the user.

This makes it perfect for lists of elements where order needs to be managed. Rather than using the fiddly **move up the list** or **move down the list** buttons that we usually see next to lists, we can apply the sortable behavior to them and allow our users to reorder the list in a much more intuitive way.

On the front page of StarTrackr! there are two lists that show the ranking of the week's top celebrities. One is for the A-list celebrities, and the other for the B-list. This is the perfect opportunity to show our client a cool trick: let's make the lists reorderable by the users. They can move the celebs up and down the lists, and even swap them if they challenge their A/B list status. When they're happy with their reordering, they can click the **Accept** button and the changes will be submitted to the server.

Lists are the primary targets for the sortable behaviour. With a little extra work a `div` can also take up the challenge. For this example, we'll use the following markup:

chapter_07/15_sortables/index.html *(excerpt)*

```
<ul id="a-list" class="connected">
  <li><a href="#">Glendatronix</a></li>
  <li><a href="#">Baron von Jovi</a></li>
  ⋮
</ul>

<ul id="b-list" class="connected">
  <li><a href="#">Mr Speaker</a></li>
  ⋮
</ul>
```

Like `draggable` and `droppable`, establishing an element as sortable is straightforward:

```
$("#a-list, #b-list").sortable();
```

There's a raft of methods, events, and options that are available when an element becomes sortable, and we can combine them to control the interesting moments that occur during the course of the sorting:

chapter_07/15_sortables/script.js *(excerpt)*

```
$("#a-list, #b-list").sortable({
  connectWith: '.connected',
  placeholder: 'ui-state-highlight',
  receive: function(event, ui) { adopt(this) },
  remove: function(event, ui) { orphan(this) }
}).disableSelection();
```

We've specified two options and two methods to our sortables, and we'll build on those methods to make our actions a little more user-friendly. A nice touch we can exploit is that accessing `this` inside the callbacks (as we've done above) gives us a reference to the sortable element.

`disableSelection`

Chained on the end of our `sortable` instantiation is a nifty action: `disableSelection`. `disableSelection`, and its reverse, `enableSelection`, are two really powerful methods in jQueryUI. Calling `disableSelection` makes it impossible for users to select text inside the target elements. It can be used to stop text from being selected when users are dragging—or sorting—the element, and prevents users from accidentally highlighting text when they just want to drag an item.

Let's look at the two methods we've assigned as event handlers:

chapter_07/15_sortables/script.js *(excerpt)*

```
function adopt(which) {
  if ($(which).hasClass('empty')) {
    $(which).removeClass('empty').find('.empty').remove();
  }
}

function orphan(which) {
  console.log(which);
  if ($(which).children().length == 0) {
    $(which)
      .append($('<li class="empty">empty</li>'))
      .addClass('empty');
  }
}
```

These methods allow us to add the text "empty" to a list when its last item is removed, and remove the text as soon as a new item is added. The `receive` event is fired when a sortable list receives an item from a connected list. We use it to call our custom `adopt` method, wherein we remove the "empty" text if it's found.

Removing a child from a sortable fires the `remove` event, which we use to call our `orphan` function. This method checks to see if the parent sortable has no children. Should it be empty, we give it a child `<li>` and assign it the `empty class`.

Progress Bar

Our client wants to implement a new feature he calls StarChirp, which will enable his users to communicate via short status messages (presumably about celebrities). We have no idea where he could have come up with this idea, but we're happy to have a go at it. He specifies that he wants to differentiate his product from other status update sites by displaying the remaining character count in the form of a progress bar. This makes sense: it'll display the percentage of how much room is left to type, so users can easily see if they're approaching their word limit.

A progress bar is one of the most recognizable messages a user can see. Thanks to countless bad movies, even the layperson understands that the progress bar is the ultimate technological ticking clock. A progress bar effectively shows how far through a long-running process or set of processes we are—and more importantly, how far we have to go.

The simplest way to simulate a progress bar is to include a block-level element inside another block-level element. The outside element's width is set to the length of the progress bar, and the inside element's width is set to the correct ratio in relation to the outer element. Give the inside element a bit of color and that's it!

As we've been using the jQuery UI library for our recent tasks, we might as well explore the whole gamut and see what the jQuery UI progress bar widget has to offer.

We've coded up a small form to hold the relevant elements, but for the progress bar all that's required is an empty `div`:

chapter_07/16_progress_bar/index.html *(excerpt)*

```
<form>
  <fieldset>
    <legend>StarChirp</legend>
    <textarea id="chirper" rows=""></textarea>
    <div id="console">
      <div id="bar"></div>
      <div id="count">0</div>
    </div>
    <input type="submit" value="Chirp!" />
  </fieldset>
</form>
```

Now we simply need to tell jQuery UI which element we'd like to transform:

chapter_07/16_progress_bar/script.js *(excerpt)*

```
$('#bar').progressbar();
```

That's it. The progress bar is ready! There's not much tweaking you can do. If you want the bar to default at a value other than 0%, you can specify it like this: `$('#bar').progressbar({value: 50})`.

For our StarChirp box, we'll monitor the user's key presses in much the same manner as we did for the maximum length indicator earlier in this chapter. This time, however, we need to update the progress bar as the user types:

chapter_07/16_progress_bar/script.js *(excerpt)*

```
$('#chirper')
  .val('')
  .keyup(function(e) {
    var characters = 140;
    var chirp = $('#chirper').val();
    var count = chirp.length;

    if (count <= characters) {
      $('#bar').progressbar('value', (count / characters) * 100);
      $('#count').text(count);
    } else {
      $('#chirper').val(chirp.substring(0,characters));
    }
  });
```

The important point to remember about the jQuery UI progress bar is that its range is from 0 to 100. It's a percentage, so you'll need to figure out the percentage to pass in. We'll divide the current number of characters by the total allowed, and multiply the result by 100. Now we have a valid value to pass to the progress bar via the `value` option.

If there are already more characters in the box than what's allowed, we'll use the JavaScript `substring` function to chop off the excess.

The effect is that every character we add will move the progress bar to the right, and every character we remove will move the progress bar to the left.

Dialogs and Notifications

In the olden days, there was little requirement for user messages on our brochure sites; perhaps just a "thanks for submitting the form," or a JavaScript popup dialog telling us we forgot to fill out an email field.

These days, as our Ajax-enabled web applications become more complex, the breadth of information that needs to be conveyed is growing: validation messages, status updates, error handling messages, and so on. Doing it in a way that avoids overwhelming or annoying the user can be quite an art form.

Simple Modal Dialog

Modal dialogs are notifications that pop up in the user's face and must be acted on if the user want to continue. It's quite an intrusion—people tend to dislike popups, so they should only be used if the interaction is essential. Our client informs us it's essential that users agree to an End User License Agreement (EULA) to use the StarTrackr! application. Not all modal dialogs are as disagreeable as our StarTrackr! EULA, however, so they're a useful control to learn to build.

What you might notice from the figure is that a modal dialog looks strikingly like a lightbox. It's a lightbox with some buttons! To supply the contents of a dialog, we'll embed the HTML in a hidden `div`. When we want to show it, we'll copy the contents into the dialog structure and fade it in. That way we can have multiple dialogs that use the same lightbox elements:

chapter_07/17_simple_modal_dialog/index.html *(excerpt)*

```
<div id="overlay">
  <div id="blanket"></div>
</div>
<!-- the dialog contents -->
<div id="eula" class="dialog">
  <h4>End User License Agreement</h4>
  ⋮
  <div class="buttons">
    <a href="#" class="ok">Agree</a>
    <a href="#" class="cancel">Disagree</a>
  </div>
</div>
```

You'll see that we've included a couple of button links in the bottom of the dialog. These are where we can hook in our events to process the user interaction. It's a fairly simple HTML base so, as you can imagine, CSS plays a big part in how effective the dialogs look. We want to stretch our structure and lightbox "blanket" over the entire screen. The modal dialog will appear to sit on top of it:

chapter_07/17_simple_modal_dialog/dialog.css *(excerpt)*

```
#overlay {
  display:none;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  margin-right: auto;
  margin-left: auto;
  position: fixed;
  width: 100%;
  z-index: 100;
}

#blanket {
  background-color: #000000;
  top: 0;
  bottom: 0;
  left: 0;
  display: block;
  opacity: 0.8;
  position: absolute;
  width: 100%;
}

.dialog {
  display: none;
  margin: 100px auto;
  position: relative;
  width: 500px;
  padding: 40px;
  background: white;
  -moz-border-radius: 10px;
}
```

Now to bring the dialog onscreen. We'll create an `openDialog` function that will be responsible for taking the dialog HTML, transporting it to the overlay structure and displaying it. The "transporting" part is achieved via the `clone` action, which creates a copy of the current jQuery selection, leaving the original in place. When we close the dialog we're going to remove the contents, so unless we cloned it each time, we'd only be able to open it once:

chapter_07/17_simple_modal_dialog/script.js *(excerpt)*

```
function openDialog(selector) {
  $(selector)
    .clone()
    .show()
    .appendTo('#overlay')
    .parent()
    .fadeIn('fast');
}
```

Because we've added the behavior to a function, we can call it whenever we need to open a dialog, and pass it the selector of the element we want to show:

chapter_07/17_simple_modal_dialog/script.js *(excerpt)*

```
$("#eulaOpen").click(function() {
  openDialog("#eula");
});
```

The second part is returning everything back to its initial state when the dialog is closed. This is achieved by finding the overlay, fading it out, and then removing the cloned dialog contents:

chapter_07/17_simple_modal_dialog/script.js *(excerpt)*

```
function closeDialog(selector) {
  $(selector)
    .parents("#overlay")
    .fadeOut('fast', function() {
      $(this)
        .find(".dialog")
        .remove();
    });
}
```

We need to call the `closeDialog` function from within the current dialog. But as well as closing it, the buttons in a dialog should have other effects. By adding extra buttons in the dialog's HTML, and hooking on to them in the document-ready part of your code, you can run any arbitrary number of event handlers and process them as you need:

chapter_07/17_simple_modal_dialog/script.js *(excerpt)*

```
$('#eula')
  .find('.ok, .cancel')
  .live('click', function() {
    closeDialog(this);
  })
  .end()
  .find('.ok')
  .live('click', function() {
    // Clicked Agree!
  })
  .end()
  .find('.cancel')
  .live('click', function() {
    // Clicked disagree!
});
```

The important part of this code is that we're using the `live` action. When we use `clone` to duplicate a DOM node, its event handlers get lost in the process—but `live` keeps everything in place no matter how often we clone and delete nodes!

This is a simple, but fairly crude way to handle the button events. In Chapter 9, we'll look at how we can set up a custom event handling system. The advantage of the method used here is that it's extremely lightweight and targeted to our particular needs. But manually creating buttons and handling the related events would become tiring fairly quickly if you have many complicated dialogs to look after, so you'll probably be interested in the jQuery UI Dialog widget.

jQuery UI Dialog

As you'd expect by now, the jQuery UI Dialog component is the complete bells and whistles version of a dialog box. Out of the box it is draggable and resizable, can be modal or non-modal, allows for various transition effects, and lets you specify the dialog buttons programmatically. A sample dialog, styled with the UI lightness theme, is shown in Figure 7.9.

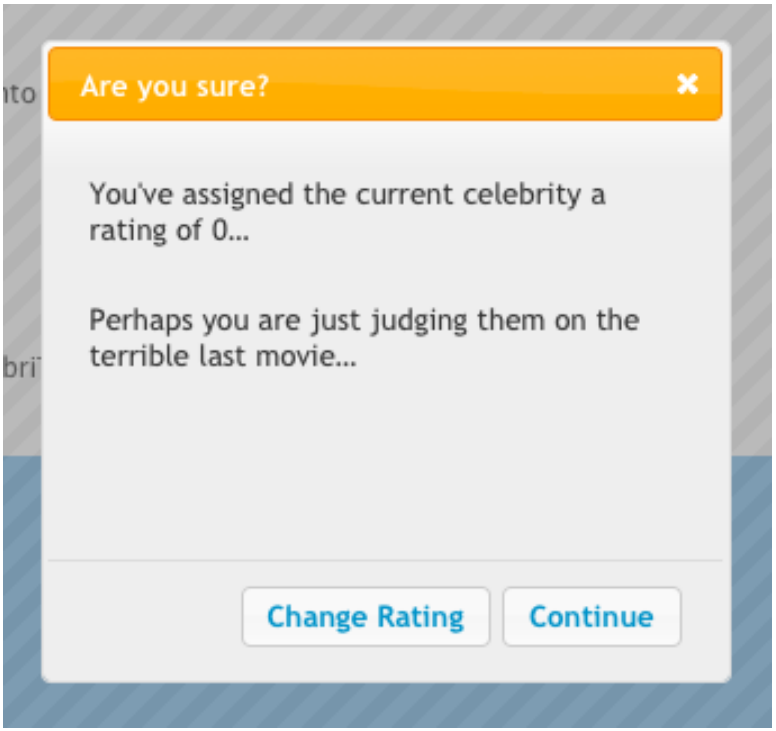

Figure 7.9. A jQuery UI dialog

Just like with our custom dialog box, the main contents are specified in the HTML itself, then hidden and displayed as necessary by the library. This way you can put whatever you like inside the dialog—including images, links, or forms:

chapter_07/18_jquery_ui_dialog/index.html *(excerpt)*

```
<div id="dialog" title="Are you sure?">
  <p>You've assigned the current celebrity a rating of 0…</p>
  <p>Perhaps you are just judging them on the terrible …</p>
</div>
```

We're using the UI lightness theme for CSS, as it matches up well with the StarTrackr! site—but the dialogs are fully skinnable, and as always you can make a custom theme with the ThemeRoller tool (more on this in the section called "Theme Rolling" in Chapter 9). As you can see from the HTML snippet, the `title` attribute specifies the text to be displayed in the title bar of the dialog. Other than that, there's little going on in our HTML … so where do those buttons come from? Let's have a look at the script:

chapter_07/18_jquery_ui_dialog/script.js *(excerpt)*

```
$('#dialog').dialog({
  autoOpen: false,
  height: 280,
  modal: true,
  resizable: false,
  buttons: {
    Continue: function() {
      $(this).dialog('close');
      // Submit Rating
    },
    'Change Rating': function() {
      $(this).dialog('close');
      // Update Rating
    }
  }
});
```

Aha, interesting! The buttons, including their text, are specified via the options passed to the `dialog` function.

The buttons are grouped together in an object and assigned to the `buttons` property of the dialog. To define a button, you need to create a named function inside the `buttons` object. The function code will execute whenever the user clicks the button—and the name of the function is the text that will be displayed on the button. If you want your button text to contain a space, you'll need to wrap the function name in quotes. The buttons are added to the dialog from right to left, so make sure you add them in the order you want them displayed. This is quite a neat way to package together the button functions with the dialog—unlike our custom dialog where the functionality was specified independently of the dialog code.

Quotes

In the above example, the second button's name is in quotes, while the first one isn't. This is simply to illustrate the necessity of enclosing multiple-word buttons in quotes; in your code it might be preferable to put quotes around everything for consistency and simplicity.

By default, the dialog will pop up as soon as you define it. This makes it easy to create small and simple dialogs as you need them. For our example, though, we

want to set up the dialog first, and only have it pop up on a certain trigger (when the user gives the poor celebrity a zero rating). To prevent the dialog popping up immediately, we set the `autoOpen` property to `false`. Now, when the page is loaded, the dialog sits and waits for further instructions.

When the user clicks the `rating-0` link, we tell the dialog to display itself by passing the string `'open'` to the `dialog` method. This is a good way to communicate with the dialog after the initialization phase:

chapter_07/18_jquery_ui_dialog/script.js *(excerpt)*

```
$('#rating-0').click(function() {
  $('#dialog').dialog('open');
});
```

That's a nice looking dialog we have there! We can now execute any required code inside the dialog button functions. As part of the code we'll also have to tell the dialog when we want it to close. If you look back at the button definitions above, you can see we have the line `$(this).dialog('close')`. As you might suspect, the `close` command is the opposite of the `open` command. You can open and close the dialogs as many times as you need.

What else can the plugin do? Well, we've specified the option `modal` to be `true`; that's why we have the nice stripey background—but by default, `modal` will be `false`, which allows the user to continue working with the rest of the page while the dialog is open. Also, we've set `resizable` to `false` (and left the `draggable` option on default—which is `true`). These options make use of the jQuery UI `resizable` and `draggable` behaviors to add some desktop flavor to the dialog.

We specified the dialog's title text in HTML, but you can also do it in jQuery via the `title` property, just as you can set its width and height. One less obvious, but extremely useful alternative is the bgiframe option. If this option is set to `true`, the bgiframe plugin will be used to nfix an issue in Internet Explorer 6 where select boxes show on top of other elements.

In terms of events, you can utilize the dialog's `open`, `close`, and `focus` events if you need to do some processing unrelated to buttons. But there's also an extremely useful `beforeClose` event that occurs when a dialog is asked to close—before it actually does! This is a great place to handle any processes you'd have to do regard-

less of which button was clicked. It's also useful if you need to stop the dialog from closing unless certain conditions are satisfied.

By now, you're starting to appreciate the depth of the jQuery UI library. All of the controls are well thought out and feature-rich. As always, you need to weight the leaner custom option against the more bandwidth-intensive (but quick to implement and more fully featured) jQuery UI alternative. Which one you choose should depend on your project requirements.

Growl-style Notifications

Our client is worried that StarTrackr! is lagging behind competitors in the real-time web space. He wants to be able to communicate with users and keep them abreast of up-to-the-second information: new Twitter posts, news from the RSS feed … anything to show that StarTrackr! is buzzing with life.

The data is no problem—the back-end team can handle it … but how can we notify the user in a way that's both cool and helpful? Once again we'll look to the desktop for inspiration, and implement Growl-style notification bubbles (Growl is a popular notification system for the Mac OS X desktop).

When we have a message to share with the users, we'll add a bubble to the page. The bubble will be located at the bottom right-hand side of the screen. If we have more messages to share, they'll appear underneath the previous ones, in a kind of upside-down stack. Each bubble will have a close button, enabling users to close them after they've been read. The overall effect is shown in Figure 7.10.

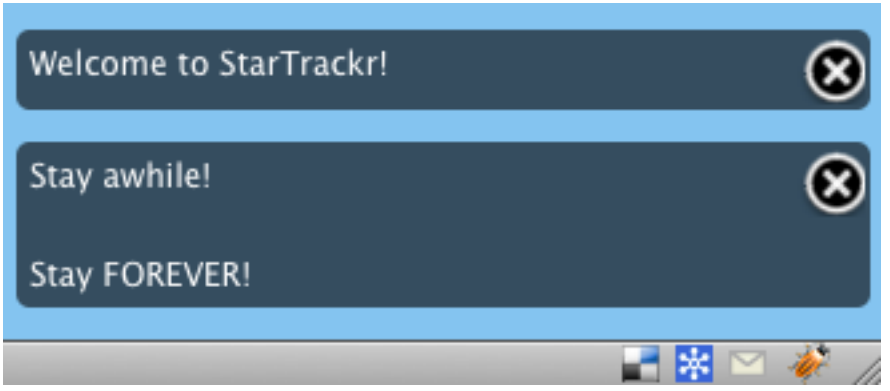

Figure 7.10. Growl-style notifications

The real trick to the bubbles is CSS. It takes care of all the tough stuff involved in positioning the dialogs and making them look really cool. In terms of HTML, all we need is a simple container:

chapter_07/19_growl_style_notifications/index.html *(excerpt)*

```
<div id="growl"></div>
```

It needs to be able to be positioned in the bottom corner to achieve the effect we're attempting. Placing it in the footer or outside of your page's main container element is common. Let's apply some basic CSS to handle the positioning:

chapter_07/19_growl_style_notifications/style.css *(excerpt)*

```
#growl {
  position: absolute;
  bottom: 0;
  right: 0;
  width: 320px;
  z-index: 10;
}
```

Now that the container is in place, we can start adding our message bubbles to it. We'll create a simple function that takes a message, wraps it in some structure, and appends it to our positioned bubble holder:

chapter_07/19_growl_style_notifications/script.js *(excerpt)*

```
function addNotice(notice) {
  $('<div class="notice"></div>')
    .append('<div class="skin"></div>')
    .append('<a href="#" class="close">close</a>')
    .append($('<div class="content"></div>').html($(notice)))
    .hide()
    .appendTo('#growl')
    .fadeIn(1000);
}
```

The structure we've added consists of a containing element with an extra `div` available for styling (we're using it to lay the visible message over a semi-opaque background), a close button, and a container for the message contents.

One other point to note about this function is that any HTML we pass to it is wrapped in the jQuery dollar selector. This means we can pass in either plain text, HTML, or jQuery objects, and they'll be displayed in the box. Again, you can style it all however suits your site—though you'll need to give the bubble container `position: relative`:

chapter_07/19_growl_style_notifications/style.css *(excerpt)*

```
.notice {
  position: relative;
}

.skin {
  position: absolute;
  background-color: #000000;
  bottom: 0;
  left: 0;
  opacity: 0.6;
  right: 0;
  top: 0;
  z-index: -1;
  -moz-border-radius: 5px; -webkit-border-radius: 5px;
}

.close {
  background: transparent url('button-close.png') 0 0 no-repeat;
}
```

This will position our bubbles correctly and give them some basic styles. Inside the document-ready function, just call the `addNotice` function with a message, and it will fade in at the bottom of the screen:

chapter_07/19_growl_style_notifications/script.js *(excerpt)*

```
addNotice("<p>Welcome to StarTrackr!</p>");
addNotice("<p>Stay awhile!</p><p>Stay FOREVER!</p>");
```

You can also pass in images, or indeed any HTML you like. Of course, most of the time you'll want to display the result of a user interaction, or an Ajax call—you just need to call `addNotice` whenever you want to display a message to the user. The only problem is … once the bubbles are there, they're unable to be removed—they just keep stacking up! Let's fix this:

chapter_07/19_growl_style_notifications/script.js *(excerpt)*

```
$('#growl')
  .find('.close')
  .live('click', function() {
    // Remove the bubble
  });
```

Instead of adding the `click` handler directly to the close button, we're using the `live` function to keep an eye on any new `.close` elements that are added. This helps us separate out the closing code and keep everything nice and readable. All that's left to do now is handle the actual removing:

chapter_07/19_growl_style_notifications/script.js *(excerpt)*

```
// Remove the bubble
$(this)
  .closest('.notice')
  .animate({
    border: 'none',
    height: 0,
    marginBottom: 0,
    marginTop: '-6px',
    opacity: 0,
    paddingBottom: 0,
    paddingTop: 0,
    queue: false
  }, 1000, function() {
    $(this).remove();
  });
```

The removal code goes looking for the nearest parent container via the `closest` action, and animates it to invisibility in an appealing way. Once it's invisible, the container is no longer needed, so we remove it from the DOM. The `closest` method is another one of jQuery's DOM traversing actions, and has the cool ability to locate the closest parent element that matches the selector you give it—including itself.

1-up Notification

It's Friday afternoon again, and the boss is out of the office. There's nothing left to do in this week's blitz, and there's still an hour left until office drinks. This seems like the perfect time to sneak a cool little feature onto the site. Throughout the book

we've concentrated on enlivening tried-and-true controls and recreating desktop effects, but jQuery's best asset is that it lets you try out new effects extremely quickly. We'll embrace the creative spirit and make a notification mechanism that comes straight out of 8-bit video gaming history: the 1-up notification.

The brainchild of web developer Jeremy Keith, 1-up notifications provide a non-modal feedback mechanism to show your user that an action happened. A small message (generally a single word) will appear at the point the action has taken place, then fade upwards and quickly away—exactly like the point scoring notifications in classic platform video games! Perhaps you'd think that this effect is only useful for novelty value—but it turns out to be a very satisfying and surprisingly subtle way to message your users.

As this is jQuery, there are many ways to put this together. Our approach will be to insert a new element that's initially hidden, positioned such that it sits directly centered and slightly above the element that triggers the action. For our triggers, we have some simple anchor tags that act as "Add to wishlist" links. When they're clicked, a notice saying "Adding" will appear above the link and rapidly fade out while moving upwards. Once the animation finishes, the button will change to "Added" and the interaction is complete:

chapter_07/20_1_up_notifications/index.html *(excerpt)*

```
<a class="wishlist" href="#">Add to wishlist</a>
```

The message elements we'll insert will have the `class adding`—so let's make sure that when we append them, they'll be invisible and properly located:

chapter_07/20_1_up_notifications/style.css *(excerpt)*

```
.adding{
  position:relative;
  left:-35px;
  top:-4px;
  display:none;
}
```

When the document is ready, we can then find all our targets and add the new message element to each of them. When a target (an element that has the `wishlist class`) is clicked, we call a custom function that sets our notification in motion.

The custom function takes a reference to the current object and a callback function to run when the interaction is complete. This function will move the selection to the link (via the `prev` action) and set its text to "Added":

chapter_07/20_1_up_notifications/script.js *(excerpt)*

```
$('<span>Adding</span>')
  .addClass('adding')
  .insertAfter('.wishlist');

$('.wishlist')
  .click(function(e) {
    doOneUp(this, function() {
      $(this).prev().text('Added');
    });
    e.preventDefault();
  })
```

Our custom function features nothing new to us at this point: it simply moves to the hidden `span` element and displays it. Now the message is visible to the end user. We then kick off an animation that adjusts the `span`'s `top` and `opacity` properties—to move it upwards and fade it out simultaneously:

chapter_07/20_1_up_notifications/script.js *(excerpt)*

```
function doOneUp(which, callback) {
  $(which)
    .next()
    .show()
    .animate({
      top:"-=50px",
      opacity:"toggle"
    },
    1000,
    function() {
      $(this)
        .css({top: ""})
        .hide(callback)
        .remove();
    });
}
```

Passing Callbacks

Notice the `callback` variable that's being passed around in the example? We supply a function as a parameter to our `doOneUp` code, but we don't do anything with it ourselves; we just pass it along as the callback to jQuery's `hide` action. When `hide` completes, it will run whatever code we gave it. In this case, it's the code to change the link text from "Add to wishlist" to "Added."

This effect is impressive, but it would be more useful if it were customizable, especially with respect to the positioning of the text message; at the moment it's hard-coded into the CSS. It would be good to make this an option in the code, and also provide options to select the distance the message travels and its speed. In short, this effect would be perfect as a plugin! You'll have to wait until (or skip over to) Chapter 9 to learn how to do that.

We're in Good Form

Building usable, accessible, and impressive forms and interface controls is hard work, and to tackle the task we have to use all of the tools we have at our disposal: HTML, CSS, JavaScript, and jQuery. It's a team effort, and as developers, we need to be aware which tool is the right one for the job. Once we've figured this out though, it's all bets off. Forms and controls are the core of application development on the Web—so it's an exciting area to be experimenting in. Striking a balance between impressive, novel, and usable interactions can be tricky, but if you get it right, you can have a significant impact on the way people use and perceive your site.

What's Next?

jQuery is rapidly becoming the norm in web design. Website visitors now expect a certain level of interactivity and animation in their web experience. Once you've read jQuery: Novice to Ninja, you'll have the ninja skills to create powerful UI widgets, impressing even the fussiest of clients.

With jQuery: Novice to Ninja, you have the best of both worlds. Not only are the fundamentals of jQuery covered in detail, you'll also be equipped with a big bunch of out-of-the-box solutions ready to use straight away.

If you're ready to create modern, effective websites that are feature-packed and dressed to impress, grab yourself a copy of jQuery: Novice to Ninja today.[1]

100% Satisfaction Guarantee

We want you to feel as confident as we do that this book will deliver the goods, so you have a full 30 days to play with it. If in that time you feel the book falls short, simply send it back and we'll give you a prompt refund of the full purchase price, minus shipping and handling.

So, for the cost of a new T-shirt, start your journey into jQuery today!

To find out more or to order your copy, visit http://www.sitepoint.com/launch/25534b.

[1] https://sitepoint.com/bookstore/go/170/25534b

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z