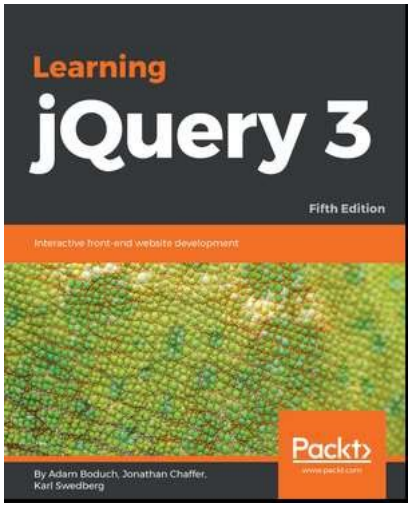


Learning jQuery 3 Fifth Edition

A step-by-step, practical tutorial on creating efficient and smart web apps and high-performance interactive pages with jQuery 3.0. Create a fully featured and responsive client-side app using jQuery. Explore jQuery 3.0 features and code examples updated to reflect modern JS environments.

Chapters
My Bookmarks
Add Bookmark

Table of Contents

My Bookmarks

Learning jQuery 3 Fifth Edition

Authors: Adam Boduch, Jonathan Chaffer, Karl Swedberg

Build interesting, interactive sites using jQuery by automating common tasks and simplifying the complicated ones

BIRMINGHAM - MUMBAI

Copyright

First published : July 2007
Second edition: Feburary 2009
Third edition: September 2011
Fourth edition: June 2013
Fifth edition: May 2017
Production reference: 1240517
Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78588-298-2

www.packtpub.com

Credits

- **Authors:** Adam Boduch, Jonathan Chaffer, Karl Swedberg
- **Copy Editor:** Charlotte Carneiro
- **Reviewer:** Andrew Kurz
- **Project Coordinator:** Devanshi Doshi
- **Commissioning Editor:** Amarabha Banerjee
- **Proofreader:** Safis Editing
- **Acquisition Editor:** Smeet Thakkar
- **Indexer:** Tejal Daruwale Soni
- **Content Development Editor:** Onkar Wani
- **Graphics:** Jason Monteiro
- **Technical Editor:** Rashil Shah
- **Production Coordinator:** Nilesh Mohite

About the Authors

Adam Boduch has been involved with large-scale JavaScript development for nearly 10 years. Before moving to the frontend, he worked on several large-scale cloud computing products, using Python and Linux. No stranger to complexity, Adam has practical experience with real-world software systems, and the scaling challenges pose.

He is the author of several JavaScript books, including React and React Native, and is passionate about innovative user experiences and high performance.

> I'd like to thank John Resig for creating jQuery, and I'd like to thank the collective jQuery community for making such a positive impact on web development.

Jonathan Chaffer is a member of Rapid Development Group, a web development firm located in Grand Rapids, Michigan. His work there includes overseeing and implementing projects in a wide variety of technologies, with an emphasis on PHP, MySQL, and JavaScript. In the open source community, he has been very active in the Drupal CMS project, which has adopted jQuery as its JavaScript framework of choice. He is the creator of the Content Construction Kit, now a part of the Drupal core used for managing structured content. He is also responsible for major overhauls of Drupal's menu system and developer API reference. In his spare time, he designs board and card games for the hobby market. He lives in Grand Rapids with his wife, Jennifer.

Karl Swedberg is a web developer at Fusionary Media in Grand Rapids, Michigan, where he spends much of his time writing both client-side and server-side JavaScript. When he isn't coding, he likes to hang out with his family, roast coffee in his garage, and exercise at the local gym.

About the Reviewer

Andrew Kurz is a UI/UX designer and developer with over 12 years of experience designing and building websites and online applications. He has worked for small start-ups, large corporations, and everything in between. He enjoys learning new technology and appreciates attractive, easy-to-use applications. He lives in Atlanta, GA, with his wife and three children. You can view his portfolio and contact him at www.kurzstudio.com.

For support files and downloads related to your book, please visit www.PacktPub.com.

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at https://www.amazon.com/dp/1785882988.

For Melissa, Jason, Simon, and Kevin. Thanks for all the love and support.

Preface

I started using jQuery in 2007, and I'm still using it today. Granted, a lot has happened between now and then: new JavaScript libraries, more consistency across browsers, and enhancements to JavaScript itself. The one thing that hasn't changed in 10 years is the expressiveness and conciseness of jQuery. Even with all the new hotness out there today, jQuery remains the go-to tool of choice for getting work done quickly, and efficiently.

This book has a long history behind it, and it remains intact in its fifth edition. It has been a successful book because it is straight to the point and easy to follow. I've done my best to preserve what has worked so well for this book. My goal is to modernize learning jQuery for the current web-development landscape.

What this book covers

Chapter 1, *Getting Started*, gets your feet wet with the jQuery JavaScript library. The chapter begins with a description of jQuery and what it can do for you. It then walks you through downloading and setting up the library as well as writing your first script.

Chapter 2, *Selecting Elements*, teaches you how to use jQuery's selector expressions and DOM-traversal methods to find elements on the page, wherever they may be. You'll use jQuery to apply styling to a diverse set of page elements, sometimes in a way that pure CSS cannot.

Chapter 3, *Handling Events*, walks you through jQuery's event-handling mechanism to fire off behaviors when browser events occur. You'll see how jQuery makes it easy to attach events to elements unobtrusively, even before the page finishes loading. Also, you'll get an overview of deeper topics, such as event bubbling, delegation, and namespacing.

Chapter 4, *Styling and Animating*, introduces you to jQuery's animation techniques and how to hide, show, and move page elements with effects that are both useful and pleasing to the eye.

Chapter 5, *Manipulating the DOM*, teaches you how to change your page on command. This chapter will also teach you how to alter the very structure of an HTML document as well as adding to its content on the fly.

Chapter 6, *Sending Data with Ajax*, walks you through many ways in which jQuery makes it easy to access server-side functionality without resorting to clunky page refreshes. With the basic components of the library well in hand, you will be ready to explore how the library can expand to fit your needs.

Chapter 7, *Using Plugins*, shows you how to find, install, and use plugins, including the powerful jQuery UI and jQuery Mobile plugin libraries.

Chapter 8, *Developing Plugins*, teaches you how to take advantage of jQuery's impressive extension capabilities to develop your own plugins from the ground up. You'll create your own utility functions, add jQuery object methods, and discover the jQuery UI widget factory. Next, you'll take a second tour through jQuery's building blocks, learning more advanced techniques.

Chapter 9, *Advanced Selectors and Traversing*, refines your knowledge of selectors and traversals, gaining the ability to optimize selectors for performance, manipulating the DOM element stack, and writing plugins that expand selecting and traversing capabilities.

Chapter 10, *Advanced Events*, dives further into techniques such as delegation and throttling that can greatly improve event-handling performance. You'll also create custom and special events that add even more capabilities to the jQuery library.

Chapter 11, *Advanced Effects*, shows you how to fine-tune the visual effects of jQuery that can be provided by crafting custom-easing functions and reacting to each step of an animation. You'll gain the ability to manipulate animations as they occur and schedule actions with custom queuing.

Chapter 12, *Advanced DOM Manipulation*, provides you with more practice modifying the DOM with techniques such as attaching arbitrary data to elements. You'll also learn how to extend the way jQuery processes CSS properties on elements.

Chapter 13, *Advanced Ajax*, helps you achieve a greater understanding of Ajax transactions, including the jQuery deferred object system for handling data that may become available at a later time.

Appendix A, *Testing JavaScript with QUnit*, teaches you about the QUnit library, which is used for the unit testing JavaScript programs. This library will be a great addition to your toolkit for developing and maintaining highly sophisticated web applications.

Appendix B, *Quick Reference*, provides a glimpse of the entire jQuery library, including every one of its methods and selector expressions. Its easy-to-scan format is perfect for those moments when you know what you want to do, but you're just unsure about the right method name or selector.

What you need for this book

In order to run the example code demonstrated in this book, you need a modern web browser, such as Google Chrome, Mozilla Firefox, Apple Safari, or Microsoft Edge.

To experiment with the examples and to work on the chapter-ending exercises, you will also need the following:

- A basic text editor
- Web development tools for the browser, such as Chrome Developer Tools or Firebug (as described in the *Using development tools* section of Chapter 1, *Getting Started*)
- The full code package for each chapter, which includes a copy of the jQuery library (seen in the *Downloading the example code* section)

Additionally, to run some of the Ajax examples in Chapter 6, *Sending Data with Ajax* and beyond, you will need Node.js.

Who this book is for

This book is ideal for client-side JavaScript developers. You do not need to have any previous experience with jQuery, although basic JavaScript programming knowledge is necessary.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "When we instruct jQuery to find all elements with the class `collapsible` and hide them, there is no need to loop through each returned element."

A block of code is set as follows:

```
body {
  background-color: #fff;
  color: #000;
  font-family: Helvetica, Arial, sans-serif;
}
h1, h2, h3 {
  margin-bottom: .2em;
}
.poem {
  margin: 0 2em;
}
.highlight {
  background-color: #ccc;
  border: 1px solid #888;
  font-style: italic;
  margin: 0.5em 0;
  padding: 0.5em;
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "The `Sources` tab allows us to view the contents of all loaded scripts on the page."

Tips, Notes, and Warnings

Hey, You!

Tips provide 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 highlight any gotchas that are likely to trip you up along the way.

Github

This example has a code repository available at Github.com.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the `SUPPORT` tab at the top.
3. Click on `Code Downloads & Errata`.
4. Enter the name of the book in the `Search` box.

5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on `Code Download`.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Learning-jQuery-3. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the `Errata Submission Form` link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the `Errata` section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

NEXT :Getting Started❯
Chapter 1

Getting Started

Adam Boduch,
Jonathan Chaffer,
Karl Swedberg

Today's **World Wide Web** (**WWW**) is a dynamic environment and its users set a high bar for both the style and function of sites. To build interesting and interactive sites, developers are turning to JavaScript libraries, such as jQuery, to automate common tasks and to simplify complicated ones. One reason the jQuery library is a popular choice is its ability to assist in a wide range of tasks.

It can seem challenging to know where to begin because jQuery performs so many different functions. Yet, there is a coherence and symmetry to the design of the library; many of its concepts are borrowed from the structure of **HTML** and **Cascading Style Sheets** (**CSS**). The library's design lends itself to a quick start for designers with little programming experience, since many have more experience with these technologies than they do with JavaScript. In fact, in this opening chapter, we'll write a functioning jQuery program in just three lines of code. On the other hand, experienced programmers will also appreciate this conceptual consistency.

In this chapter, we will cover:

- The primary features of jQuery
- Setting up a jQuery code environment
- A simple working jQuery script example
- Reasons to choose jQuery over plain JavaScript
- Common JavaScript development tools

What jQuery does?

The jQuery library provides a general-purpose abstraction layer for common web scripting, and it is therefore useful in almost every scripting situation. Its extensible nature means that we could never cover all the possible uses and functions in a single book, as plugins are constantly being developed to add new abilities. The core features, though, assist us in accomplishing the following tasks:

- **Access elements in a document**: Without a JavaScript library, web developers often need to write many lines of code to traverse the **Document Object Model** (**DOM**) tree and locate specific portions of an HTML document's structure. With jQuery, developers have a robust and efficient selector mechanism at their disposal, making it easy to retrieve the exact piece of the document that needs to be inspected or manipulated.

```
$('div.content').find('p');
```

- **Modify the appearance of a web page**: CSS offers a powerful method of influencing the way a document is rendered, but it falls short when not all web browsers support the same standards. With jQuery, developers can bridge this gap, relying on the same standards support across all browsers. In addition, jQuery can change the classes or individual style properties applied to a portion of the document even after the page has been rendered.

```
$('ul > li:first').addClass('active');
```

- **Alter the content of a document**: Not limited to mere cosmetic changes, jQuery can modify the content of a document itself with a few keystrokes. Text can be changed, images can be inserted or swapped, lists can be reordered, or the entire structure of the HTML can be rewritten and extended--all with a single easy-to-use **Application Programming Interface** (**API**).

```
$('#container').append('<a href="more.html">more</a>');
```

- **Respond to a user's interaction**: Even the most elaborate and powerful behaviors are not useful if we can't control when they take place. The jQuery library offers an elegant way to intercept a wide variety of events, such as a user clicking on a link, without the need to clutter the HTML code itself with event handlers.

```
$('button.show-details').click(() => {
  $('div.details').show();
});
```

- **Animate changes being made to a document**: To effectively implement such interactive behaviors, a designer must also provide visual feedback to the user. The jQuery library facilitates this by providing an array of effects such as fades and wipes, as well as a toolkit for crafting new ones.

```
$('div.details').slideDown();
```

- **Retrieve information from a server without refreshing a page**: This pattern is known as **Ajax**, which originally stood for **Asynchronous JavaScript and XML**, but has since come to represent a much greater set of technologies for communicating between the client and the server. The jQuery library removes the browser-specific complexity from this process, allowing developers to focus on the server-side functionality.

```
$('div.details').load('more.html #content');
```

Why jQuery works well?

With the resurgence of interest in dynamic HTML comes a proliferation of JavaScript frameworks. Some are specialized, focusing on just one or two of the tasks previously mentioned. Others attempt to catalog every possible behavior and animation and serves these up prepackaged. To maintain the wide range of features outlined earlier while remaining relatively compact, jQuery employs several strategies:

- **Leverage knowledge of CSS**: By basing the mechanism for locating page elements on CSS selectors, jQuery inherits a terse yet legible way of expressing a document's structure. The jQuery library becomes an entry point for designers who want to add behaviors to their pages because a prerequisite for doing professional web development is knowledge of CSS syntax.
- **Support extensions**: In order to avoid "feature creep", jQuery relegates special-case uses to plugins. The method for creating new plugins is simple and well documented, which has spurred the development of a wide variety of inventive and useful modules. Even most of the features in the basic jQuery download are internally realized through the plugin architecture and can be removed if desired, yielding an even smaller library.
- **Abstract away browser quirks**: An unfortunate reality of web development is that each browser has its own set of deviations from published standards. A significant portion of any web application can be relegated to handling features differently on each platform. While the ever-evolving browser landscape makes a perfectly browser-neutral codebase impossible for some advanced features, jQuery adds an abstraction layer that normalizes the common tasks, reducing the size of code while tremendously simplifying it.
- **Always work with sets**: When we instruct jQuery to find all elements with the class `collapsible` and hide them, there is no need to loop through each returned element. Instead, methods such as `.hide()` are designed to automatically work on sets of objects instead of individual ones. This technique, called *implicit iteration*, means that many looping constructs become unnecessary, shortening code considerably.
- **Allow multiple actions in one line**: To avoid overuse of temporary variables or wasteful repetition, jQuery employs a programming pattern called *chaining* for the majority of its methods. This means that the result of most operations on an object is the object itself, ready for the next action to be applied to it.

These strategies keep the file size of the jQuery package small, while at the same time providing techniques for keeping our custom code that uses the library compact as well.

The elegance of the library comes about partly by design and partly due to the evolutionary process spurred by the vibrant community that has sprung up around the project. Users of jQuery gather to discuss not only the development of plugins but also enhancements to the core library. The users and developers also assist in continually improving the official project documentation, which can be found at http://api.jquery.com.

Despite all the efforts required to engineer such a flexible and robust system, the end product is free for all to use. This open source project is licensed under the MIT License to permit free use of jQuery on any site and facilitate its use within proprietary software. If a project requires it, developers can relicense jQuery under the GNU Public License for inclusion in other GNU-licensed open source projects.

What's new in jQuery 3?

The changes introduced in jQuery 3 are quite subtle compared to the changes introduced in jQuery 2. Most of what's changed is under the hood. Let's take a brief look at some changes and how they're likely to impact an existing jQuery project. You can review the fine-grained details (https://jquery.com/upgrade-guide/3.0) while reading this book.

Browser support

The biggest change with browser support in jQuery 3 is Internet Explorer. Having to support older versions of this browser is the bane of any web developer's existence. jQuery 3 has taken a big step forward by only supporting IE9+. The support policy for other browsers is the current version and the previous version.

Note

The days of Internet Explorer are numbered. Microsoft has released the successor to IE called Edge. This browser is a completely separate project from IE and isn't burdened by the issues that have plagued IE. Additionally, recent versions of Microsoft Windows actually push for Edge as the default browser, and updates are regular and predictable. Goodbye and good riddance IE.

Deferred objects

The `Deferred` object was introduced in jQuery 1.5 as a means to better manage asynchronous behavior. They were kind of like ES2015 promises, but different enough that they weren't interchangeable. Now that the ES2015 version of JavaScript is commonplace in modern browsers, the `Deferred` object is fully compatible with native `Promise` objects. This means that quite a lot has changed with the old `Deferred` implementation.

Asynchronous document-ready

The idea that the document-ready callback function is executed asynchronously might seem counterintuitive at first. There are a couple of reasons this is the case in jQuery 3. First, the `$(() => {})` expression returns a `Deferred` instance, and these now behave like native promises. The second reason is that there's a `jQuery.ready` promise that resolves when the document is ready. As you'll see later on in this book, you can use this promise alongside other promises to perform other asynchronous tasks before the DOM is ready to render.

All the rest

There are a number of other breaking changes to the API that were introduced in jQuery 3 that we won't dwell on here. The upgrade guide that I mentioned earlier goes into detail about each of these changes and how to deal with them. However, I'll point out functionality that's new or different in jQuery 3 as we make our way through this book.

Making our first jQuery-powered web page

Now that we have covered the range of features available to us with jQuery, we can examine how to put the library into action. To get started, we need to download a copy of jQuery.

Downloading jQuery

No installation is required. To use jQuery, we just need a publicly available copy of the file, no matter whether that copy is on an external site or our own. Since JavaScript is an interpreted language, there is no compilation or build phase to worry about. Whenever we need a page to have jQuery available, we will simply refer to the file's location from a `<script>` element in the HTML document.

The official jQuery website (http://jquery.com/) always has the most up-to-date stable version of the library, which can be downloaded right from the home page of the site. Several versions of jQuery may be available at any given moment; the most appropriate for us as site developers will be the latest uncompressed version of the library. This can be replaced with a compressed version in production environments.

As jQuery's popularity has grown, companies have made the file freely available through their **Content Delivery Networks** (**CDNs**). Most notably, Google (https://developers.google.com/speed/libraries/devguide), Microsoft (http://www.asp.net/ajaxlibrary/cdn.ashx), and the jQuery project itself (http://code.jquery.com) offer the file on powerful, low-latency servers distributed around the world for fast download, regardless of the user's location. While a CDN-hosted copy of jQuery has speed advantages due to server distribution and caching, using a local copy can be convenient during development. Throughout this book, we'll use a copy of the file stored on our own system, which will allow us to run our code whether we're connected to the Internet or not.

Tip

To avoid unexpected bugs, always use a specific version of jQuery. For example, 3.1.1. Some CDNs allow you to link to the latest version of the library. Similarly, if you're using `npm` to install jQuery, always make sure that your `package.json` requires a specific version.

Setting up jQuery in an HTML document

There are three pieces to most examples of jQuery usage: the HTML document, CSS files to style it, and JavaScript files to act on it. For our first example, we'll use a page with a book excerpt that has a number of classes applied to portions of it. This page includes a reference to the latest version of the jQuery library, which we have downloaded, renamed `jquery.js`, and placed in our local project directory:

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Through the Looking-Glass</title>

    <link rel="stylesheet" href="01.css">

    <script src="jquery.js"></script>
    <script src="01.js"></script>
  </head>

  <body>
    <h1>Through the Looking-Glass</h1>
    <div class="author">by Lewis Carroll</div>

    <div class="chapter" id="chapter-1">
      <h2 class="chapter-title">1. Looking-Glass House</h2>
      <p>There was a book lying near Alice on the table,
        and while she sat watching the White King (for she
        was still a little anxious about him, and had the
        ink all ready to throw over him, in case he fainted
        again), she turned over the leaves, to find some
        part that she could read, <span class="spoken">
        "&amp;mdash;for it's all in some language I don't know,"
        </span> she said to herself.</p>
      <p>It was like this.</p>
      <div class="poem">
        <h3 class="poem-title">YKCOWREBBAJ</h3>
        <div class="poem-stanza">
          <div>sevot yhtils eht dna ,gillirb sawT'</div>
          <div>;ebaw eht ni elbmig dna eryg diD</div>
          <div>,sevogorob eht erew ysmim llA</div>
          <div>.ebargtuo shtar emom eht dnA</div>
        </div>
      </div>
      <p>She puzzled over this for some time, but at last
        a bright thought struck her. <span class="spoken">
        "Why, it's a Looking-glass book, of course! And if
        I hold it up to a glass, the words will all go the
```

```
        right way again."</span></p>
      <p>This was the poem that Alice read.</p>
      <div class="poem">
        <h3 class="poem-title">JABBERWOCKY</h3>
        <div class="poem-stanza">
          <div>'Twas brillig, and the slithy toves</div>
          <div>Did gyre and gimble in the wabe;</div>
          <div>All mimsy were the borogoves,</div>
          <div>And the mome raths outgrabe.</div>
        </div>
      </div>
    </div>
  </body>
</html>
```

Immediately following the normal HTML preamble, the stylesheet is loaded. For this example, we'll use a simple one:

```
body {
  background-color: #fff;
  color: #000;
  font-family: Helvetica, Arial, sans-serif;
}
h1, h2, h3 {
  margin-bottom: .2em;
}
.poem {
  margin: 0 2em;
}
.highlight {
  background-color: #ccc;
  border: 1px solid #888;
  font-style: italic;
  margin: 0.5em 0;
  padding: 0.5em;
}
```

Getting the example code

You can access the example code from the following GitHub repository: https://github.com/PacktPublishing/Learning-jQuery-3.

After the stylesheet is referenced, the JavaScript files are included. It is important that the `script` tag for the jQuery library be placed before the tag for our custom scripts; otherwise, the jQuery framework will not be available when our code attempts to reference it.

Github

Throughout the rest of this book, only the relevant portions of HTML and CSS files will be printed. The files in their entirety are available from the book's companion code examples: https://github.com/PacktPublishing/Learning-jQuery-3.

Now, we have a page that looks like this:

We will use jQuery to apply a new style to the poem text.

Note

This example is to demonstrate a simple use of jQuery. In real-world situations, this type of styling could be performed purely with CSS.

Adding our jQuery code

Our custom code will go in the second, currently empty, JavaScript file, which we included from the HTML using `<script src="01.js"></script>`. For this example, we only need three lines of code:

```
$(() => {
  $('div.poem-stanza').addClass('highlight')
});
```

Note

I'll be using newer ES2015 **arrow function** syntax for most callback functions throughout the book. The only reason is that it's more concise than having the `function` keyword all over the place. However, if you're more comfortable with the `function() {}` syntax, by all means, use it.

Now let's step through this script piece by piece to see how it works.

Finding the poem text

The fundamental operation in jQuery is selecting a part of the document. This is done with the `$()` function. Typically, it takes a string as a parameter, which can contain any CSS selector expression. In this case, we wish to find all of the `<div>` elements in the document that have the `poem-stanza` class applied to them, so the selector is very simple. However, we will cover much more sophisticated options through the course of the book. We will walk through many ways of locating parts of a document in Chapter 2, *Selecting Elements*.

When called, the `$()` function returns a new jQuery object instance, which is the basic building block we will be working with from now on. This object encapsulates zero or more DOM elements and allows us to interact with them in many different ways. In this case, we wish to modify the appearance of these parts of the page and we will accomplish this by changing the classes applied to the poem text.

Injecting the new class

The `.addClass()` method, like most jQuery methods, is named self descriptively; it applies a CSS class to the part of the page that we have selected. Its only parameter is the name of the class to add. This method, and its counterpart, `.removeClass()`, will allow us to easily observe jQuery in action as we explore the different selector expressions available to us. For now, our example simply adds the `highlight` class, which our stylesheet has defined as italicized text with a gray background and a border.

Note

Note that no iteration is necessary to add the class to all the poem stanzas. As we discussed, jQuery uses implicit iteration within methods such as `.addClass()`, so a single function call is all it takes to alter all the selected parts of the document.

Executing the code

Taken together, `$()` and `.addClass()` are enough for us to accomplish our goal of changing the appearance of the poem text. However, if this line of code is inserted alone in the document header, it will have no effect. JavaScript code is run as soon as it is encountered in the browser, and at the time the header is being processed, no HTML is yet present to style. We need to delay the execution of the code until after the DOM is available for our use.

With the `$(() => {})` construct (passing a function instead of a selector expression), jQuery allows us to schedule function calls for firing once the DOM is loaded, without necessarily waiting for images to fully render. While this event scheduling is possible without the aid of jQuery, `$(() => {})` provides an especially elegant cross-browser solution that includes the following features:

- It uses the browser's native DOM-ready implementations when available and adds a `window.onload` event handler as a safety net
- It executes functions passed to `$()` even if it is called after the browser event has already occurred
- It handles the event scheduling asynchronously to allow scripts to delay if necessary

The `$()` function's parameter can accept a reference to an already defined function, as shown in the following code snippet:

```
function addHighlightClass()  {
  $('div.poem-stanza').addClass('highlight');
}

$(addHighlightClass);
```

Listing 1.1

However, as demonstrated in the original version of the script and repeated in *Listing 1.2*, the method can also accept an anonymous function:

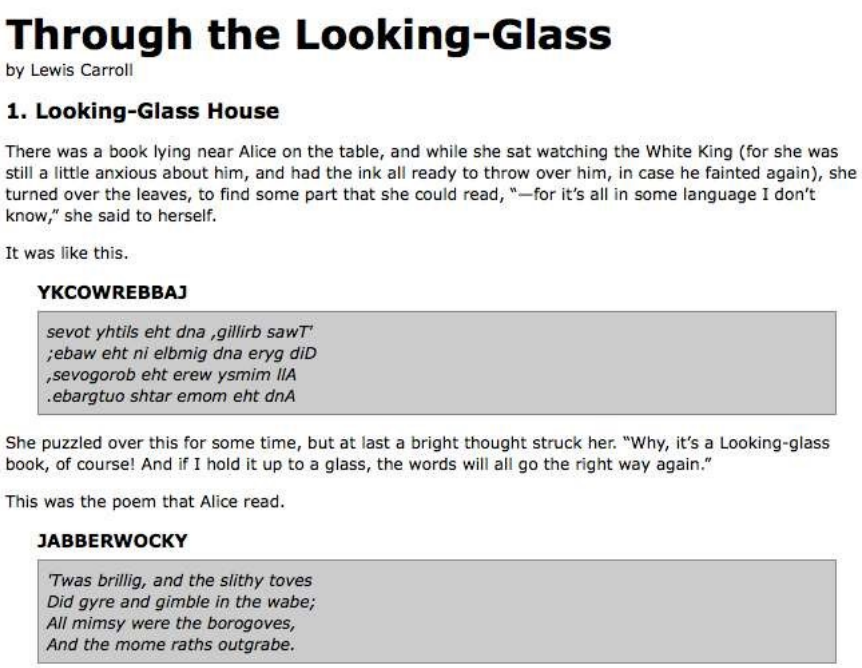
```
$(() =>
  $('div.poem-stanza').addClass('highlight')
);
```

Listing 1.2

This anonymous function idiom is convenient in jQuery code for methods that take a function as an argument when that function isn't reusable. Moreover, the closure it creates can be an advanced and powerful tool. If you're using arrow functions, you also get lexically bound `this` as a context, which avoids having to bind functions. It may also have unintended consequences and ramifications of memory use, however, if not dealt with carefully.

The finished product

Now that our JavaScript is in place, the page looks like this:



Through the Looking-Glass

by Lewis Carroll

1. Looking-Glass House

There was a book lying near Alice on the table, and while she sat watching the White King (for she was still a little anxious about him, and had the ink all ready to throw over him, in case he fainted again), she turned over the leaves, to find some part that she could read, "—for it's all in some language I don't know," she said to herself.

It was like this.

YKCOWREBBAJ

sevot yhtils eht dna ,gillirb sawT'
;ebaw eht ni elbmig dna eryg diD
,sevogorob eht erew ysmim llA
.ebargtuo shtar emom eht dnA

She puzzled over this for some time, but at last a bright thought struck her. "Why, it's a Looking-glass book, of course! And if I hold it up to a glass, the words will all go the right way again."

This was the poem that Alice read.

JABBERWOCKY

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

The poem stanzas are now italicized and enclosed in boxes, as specified by the `01.css` stylesheet, due to the insertion of the `highlight` class by the JavaScript code.

Plain JavaScript versus jQuery

Even a task as simple as this can be complicated without jQuery at our disposal. In plain JavaScript, we could add the `highlight` class this way:

```
window.onload = function() {
  const divs = document.getElementsByTagName('div');
  const hasClass = (elem, cls) =>
    new RegExp(` ${cls} `).test(` ${elem.className} `);

  for (let div of divs) {
    if (hasClass(div, 'poem-stanza') && !hasClass(div, 'highlight')) {
      div.className += ' highlight';
    }
  }
};
```

Listing 1.3

Despite its length, this solution does not handle many of the situations that jQuery takes care of for us in *Listing 1.2*, such as:

- Properly respecting other `window.onload` event handlers
- Acting as soon as the DOM is ready
- Optimizing element retrieval and other tasks with modern DOM methods

We can see that our jQuery-driven code is easier to write, simpler to read, and faster to execute than its plain JavaScript equivalent.

Using development tools

As this code comparison has shown, jQuery code is typically shorter and clearer than its basic JavaScript equivalent. However, this doesn't mean we will always write code that is free from bugs or that we will intuitively understand what is happening on our pages at all times. Our jQuery coding experience will be much smoother with the assistance of standard development tools.

High-quality development tools are available in all modern browsers. We can feel free to use the environment that is most comfortable to us. Options include the following:

- Microsoft Edge (https://developer.microsoft.com/en-us/microsoft-edge/platform/documentation/f12-devtools-guide/)
- Internet Explorer Developer Tools (http://msdn.microsoft.com/en-us/library/dd565628.aspx)
- Safari Web Development Tools (https://developer.apple.com/safari/tools/)
- Chrome Developer Tools (https://developer.chrome.com/devtools)
- Firefox Developer Tools (https://developer.mozilla.org/en-US/docs/Tools)

Each of these toolkits offers similar development features, including:

- Exploring and modifying aspects of the DOM
- Investigating the relationship between CSS and its effect on page presentation
- Convenient tracing of script execution through special methods
- Pausing execution of running scripts and inspecting variable values

While the details of these features vary from one tool to the next, the general concepts remain the same. In this book, some examples will require the use of one of these toolkits; we will use Chrome Developer Tools for these demonstrations, but development tools for other browsers are fine alternatives.

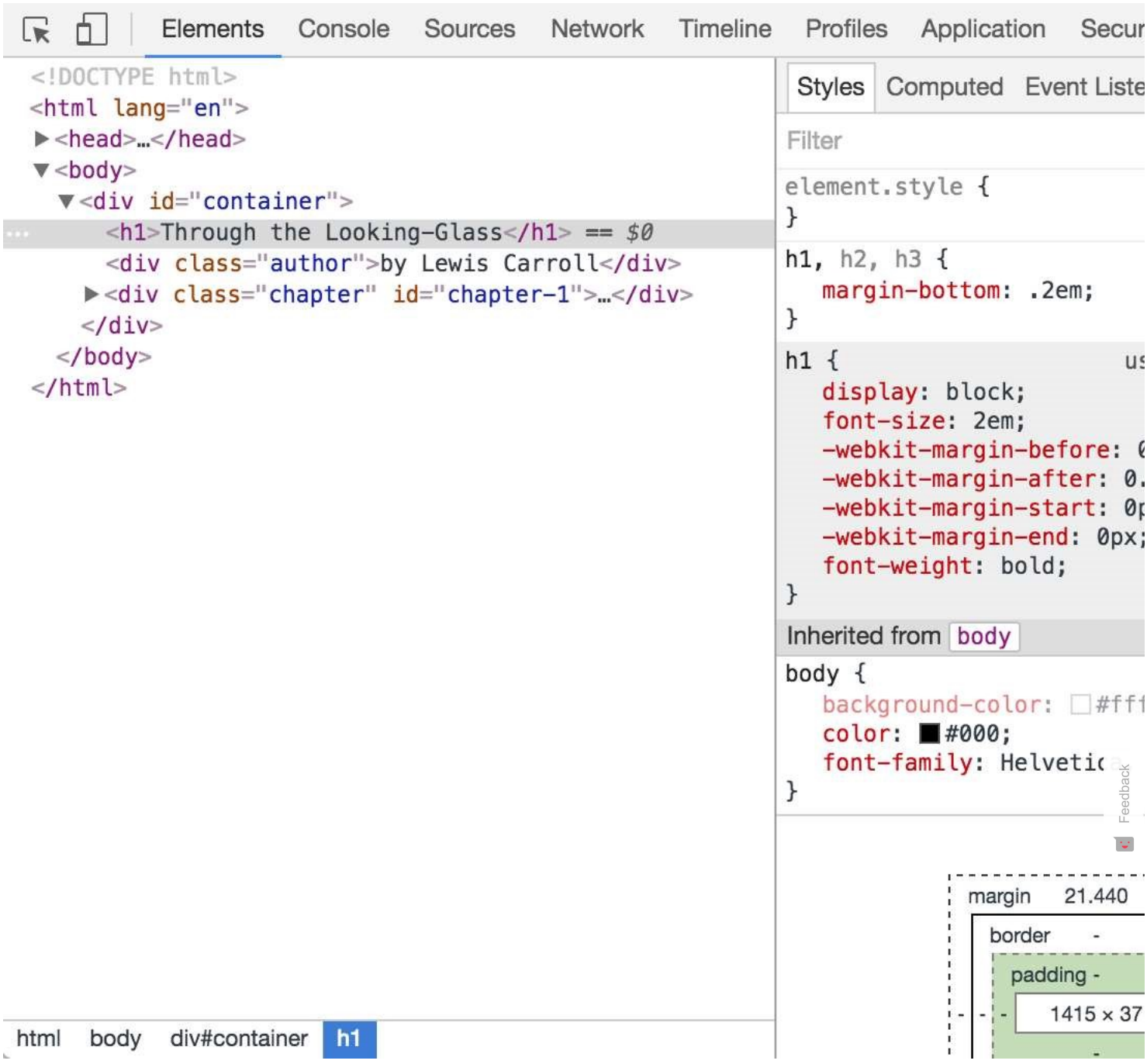
Chrome Developer Tools

Up-to-date instructions for accessing and using Chrome Developer Tools can be found on the project's documentation pages at https://developer.chrome.com/devtools. The tools are too involved to explore in great detail here, but a survey of some of the most relevant features will be useful to us.

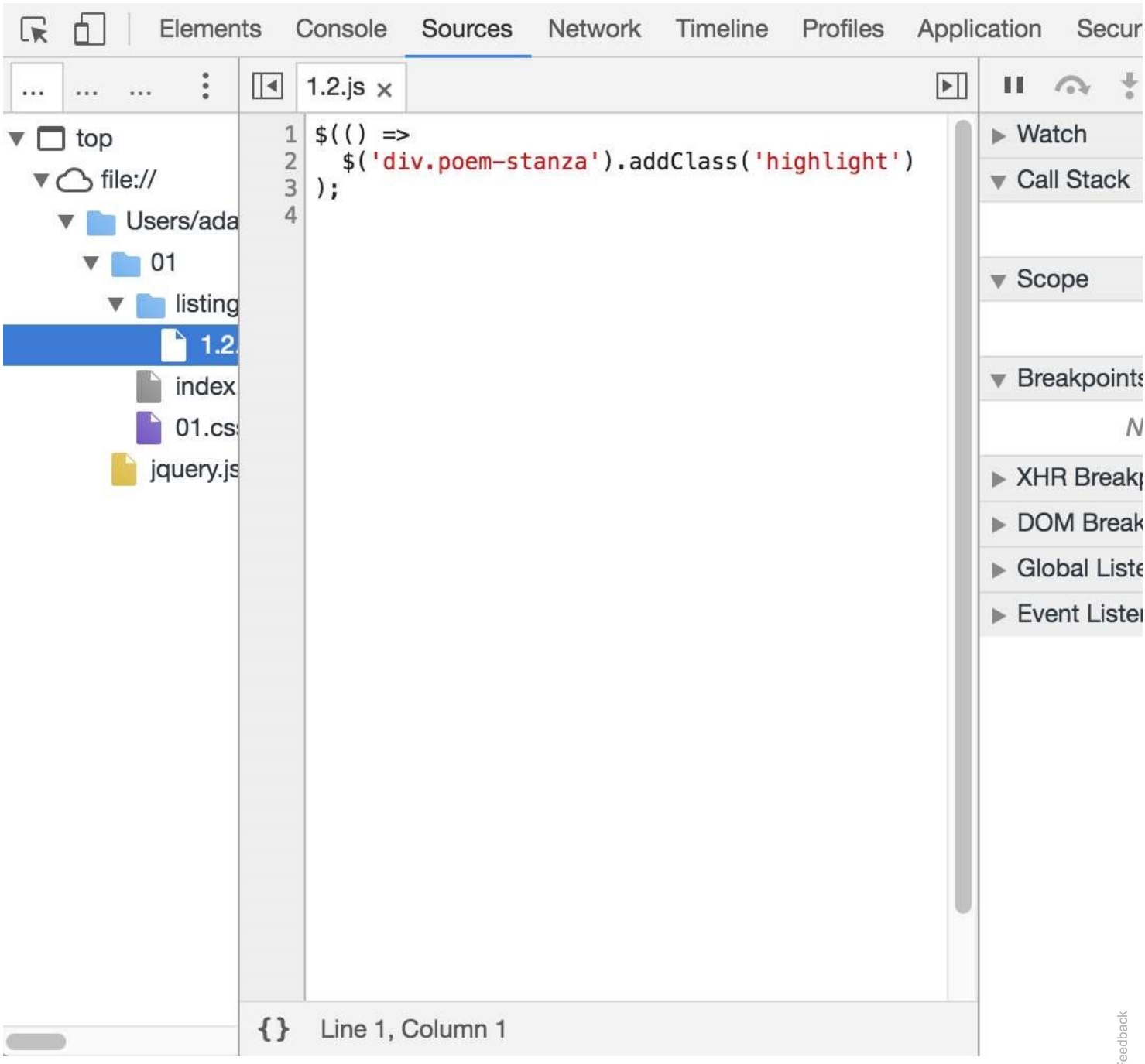
Tip

Understanding these screenshots Chrome Developer Tools is a quickly evolving project, so the following screenshots may not exactly match your environment.

When Chrome Developer Tools is activated, a new panel appears offering information about the current page. In the default Elements tab of this panel, we can see a representation of the page structure on the left-hand side and details of the selected element (such as the CSS rules that apply to it) on the right-hand side. This tab is especially useful for investigating the structure of the page and debugging CSS issues:

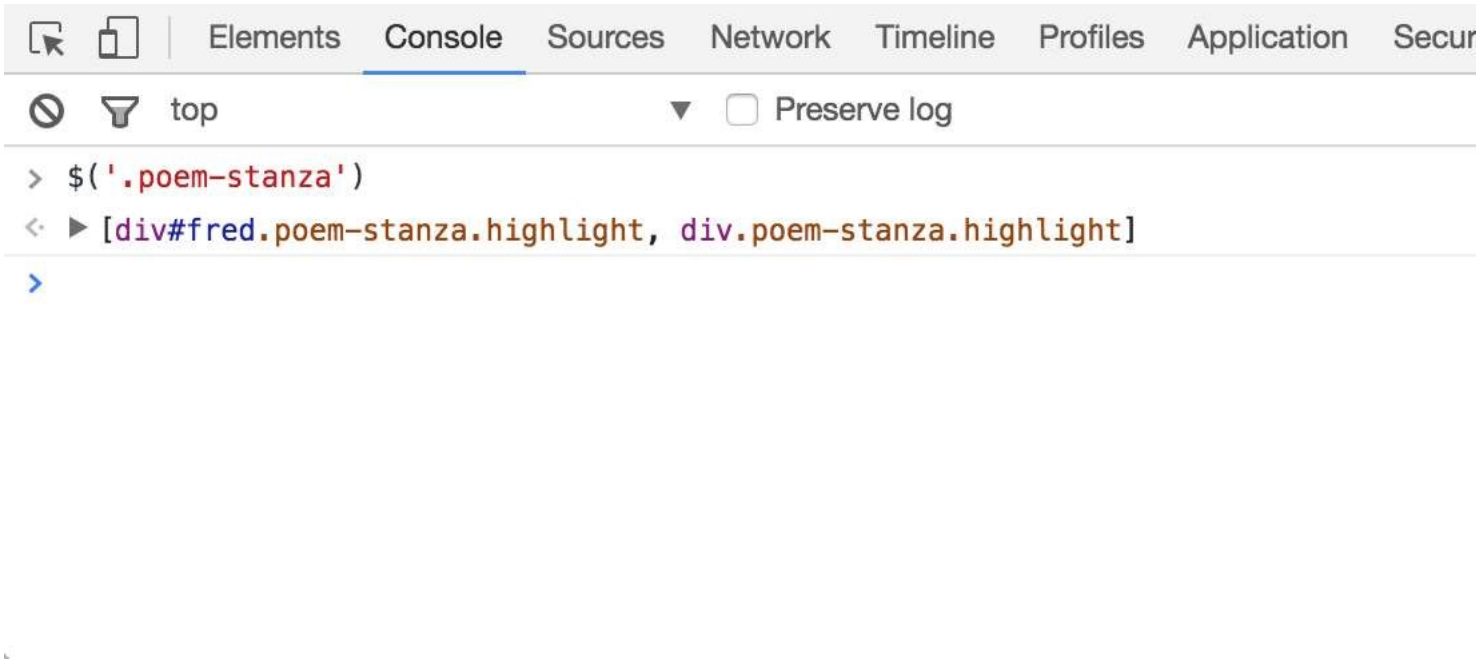


The Sources tab allows us to view the contents of all loaded scripts on the page. By right-clicking on a line number, we can set a breakpoint, set a conditional breakpoint, or have the script continue to that line after another breakpoint is reached. Breakpoints are effective ways to pause the execution of a script and examine what occurs in a step-by-step fashion. On the right-hand side of the page, we can enter a list of variables and expressions we wish to know the value of at any time:



The Console tab will be of most frequent use to us while learning jQuery. A field at the bottom of the panel allows us to enter any JavaScript statement, and the result the statement is then presented in the panel.

In this example, we perform the same jQuery selector as in *Listing 1.2*, but we are not performing any action on the selected elements. Even so, the statement gives us interesting information: we see that the result of the selector is a jQuery object pointing to the two `.poem-stanza` elements on the page. We can use this console feature to quickly try out jQuery code at any time, right from within the browser:

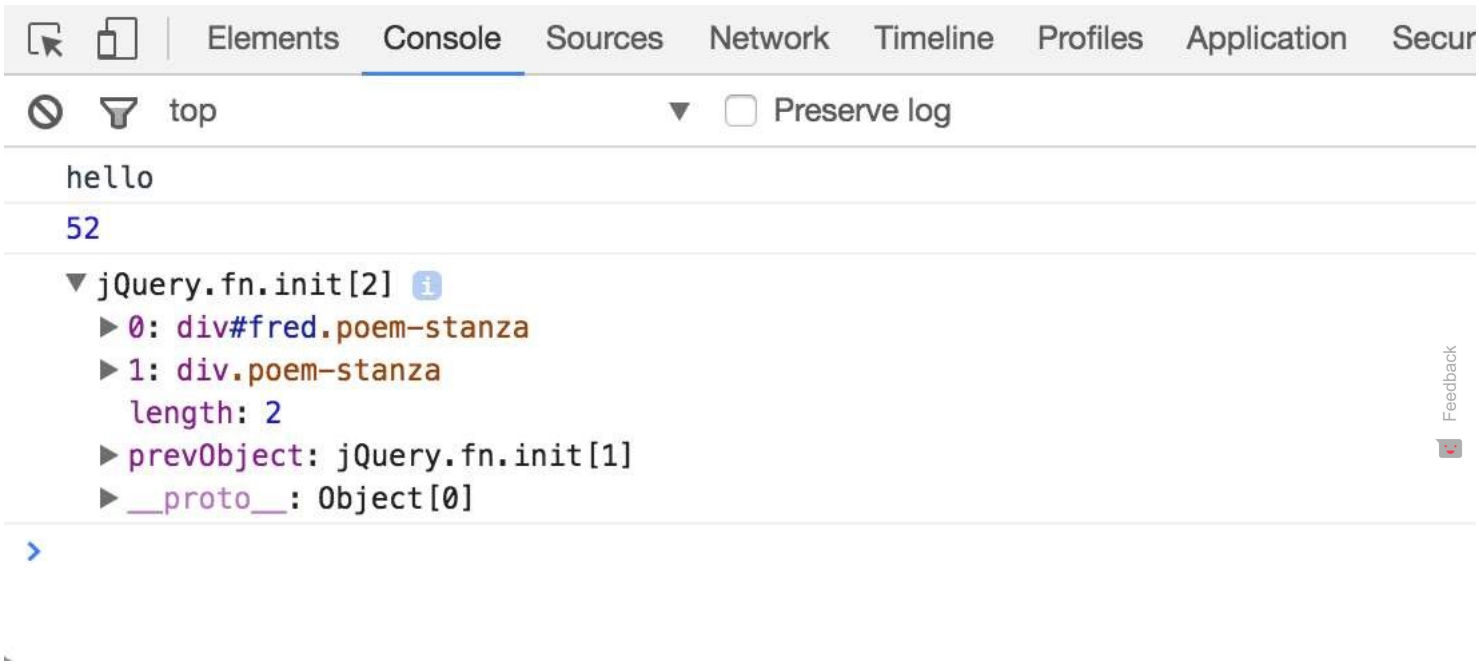


In addition, we can interact with this console directly from our code using the `console.log()` method:

```
$(() => {
  console.log('hello');
  console.log(52);
  console.log($('div.poem-stanza'));
});
```

Listing 1.4

This code illustrates that we can pass any kind of expression into the `console.log()` method. Simple values such as strings and numbers are printed directly, and more complicated values such as jQuery objects are nicely formatted for our inspection:



This `console.log()` function (which works in each of the browser developer tools we mentioned earlier) is a convenient alternative to the JavaScript `alert()` function, and will be very useful as we test our jQuery code.

Summary

In this chapter, we learned how to make jQuery available to JavaScript code on our web page, use the `$()` function to locate a part of the page that has a given class, call `.addClass()` to apply additional styling to this part of the page, and invoke `$(() => {})` to cause this function to execute upon loading the page. We have also explored the development tools we will be relying on when writing, testing, and debugging our jQuery code.

We now have an idea of why a developer would choose to use a JavaScript framework rather than writing all code from scratch, even for the most basic tasks. We have also seen some of the ways in which jQuery excels as a framework, why we might choose it over other options, and in general, which tasks jQuery makes easier.

The simple example we have been using demonstrates how jQuery works, but is not very useful in real-world situations. In the next chapter, we will expand on this code by exploring jQuery's sophisticated selector language, finding practical uses for this technique.

NEXT :Selecting Elements❯
Chapter 2

Selecting Elements

Adam Boduch,
Jonathan Chaffer,
Karl Swedberg

The jQuery library harnesses the power of **Cascading Style Sheets** (**CSS**) selectors to let us quickly and easily access elements or groups of elements in the **Document Object Model** (**DOM**).

In this chapter, we will cover:

- The structure of the elements on a web page
- How to use CSS selectors to find elements on the page
- What happens when the specificity of a CSS selector changes
- Custom jQuery extensions to the standard set of CSS selectors
- The DOM traversal methods, which provide greater flexibility for accessing elements on the page
- Using modern JavaScript language features to iterate over jQuery objects efficiently

Understanding the DOM

One of the most powerful aspects of jQuery is its ability to make selecting elements in the DOM easy. The DOM serves as the interface between JavaScript and a web page; it provides a representation of the source HTML as a network of objects rather than as plain text.

This network takes the form of a family tree of elements on the page. When we refer to the relationships that elements have with one another, we use the same terminology that we use when referring to family relationships: parents, children, siblings, and so on. A simple example can help us understand how the family tree metaphor applies to a document:

```
<html>
  <head>
    <title>the title</title>
  </head>
  <body>
    <div>
      <p>This is a paragraph.</p>
      <p>This is another paragraph.</p>
      <p>This is yet another paragraph.</p>
    </div>
  </body>
</html>
```

Here, `<html>` is the ancestor of all the other elements; in other words, all the other elements are descendants of `<html>`. The `<head>` and `<body>` elements are not only descendants, but children of `<html>` as well. Likewise, in addition to being the ancestor of `<head>` and `<body>`, `<html>` is also their parent. The `<p>` elements are children (and descendants) of `<div>`, descendants of `<body>` and `<html>`, and siblings of each other.

To help visualize the family tree structure of the DOM, we can use the browser's developer tools to inspect the DOM structure of any page. This is especially helpful when you're curious about how some other application works, and you want to implement something similar.

With this tree of elements at our disposal, we'll be able to use jQuery to efficiently locate any set of elements on the page. Our tools to achieve this are jQuery **selectors** and **traversal methods**.

Using the $() function

The resulting set of elements from jQuery's selectors and methods is always represented by a jQuery object. These objects are very easy to work with when we want to actually do something with the things that we find on a page. We can easily bind events to these objects and add visual effects to them, as well as chain multiple modifications or effects together.

Note

Note that jQuery objects are different from regular DOM elements or node lists, and as such do not necessarily provide the same methods and properties for some tasks. In the final part of this chapter, we will look at ways to directly access the DOM elements that are collected within a jQuery object.

In order to create a new jQuery object, we use the `$()` function. This function typically accepts a CSS selector as its sole parameter and serves as a factory, returning a new jQuery object pointing to the corresponding elements on the page. Just about anything that can be used in a stylesheet can also be passed as a string to this function, allowing us to apply jQuery methods to the matched set of elements.

Tip

Making jQuery play well with other JavaScript libraries In jQuery, the dollar sign (`$`) is simply an alias for `jQuery`. Because a `$()` function is very common in JavaScript libraries, conflicts could arise if more than one of these libraries were being used in a given page. We can avoid such conflicts by replacing every instance of `$` with `jQuery` in our custom jQuery code. Additional solutions to this problem are addressed in Chapter 10, *Advanced Events*. On the other hand, jQuery is so prominent in frontend development, that libraries tend to leave the `$` symbol alone.

The three primary building blocks of selectors are **tag name**, **ID**, and **class**. They can be used either on their own or in combination with others. The following simple examples illustrate how these three selectors appear in code:

Selector type	CSS	jQuery	What it does
Tag name	`p { }`	`$('p')`	This selects all paragraphs in the document.
ID	`#some-id { }`	`$('#some-id')`	This selects the single element in the document that has an ID of `some-id`.
Class	`.some-class { }`	`$('.some-class')`	This selects all elements in the document that have a class of `some-class`.

As mentioned in Chapter 1, *Getting Started*, when we call methods of a jQuery object, the elements referred by the selector we passed to `$()` are looped through automatically and implicitly. Therefore, we can usually avoid explicit iteration, such as a `for` loop, that is so often required in DOM scripting.

Now that we covered the basics, we're ready to start exploring some more powerful uses of selectors.

CSS selectors

The jQuery library supports nearly all the selectors included in CSS specifications 1 through 3, as outlined on the World Wide Web Consortium's site: http://www.w3.org/Style/CSS/specs. This support allows developers to enhance their websites without worrying about which browsers might not understand more advanced selectors, as long as the browsers have JavaScript enabled.

Progressive Enhancement

Responsible jQuery developers should always apply the concepts of progressive enhancement and graceful degradation to their code, ensuring that a page will render as accurately, even if not as beautifully, with JavaScript disabled as it does with JavaScript turned on. We will continue to explore these concepts throughout the book. More information on progressive enhancement can be found at http://en.wikipedia.org/wiki/Progressive_enhancement. Having said this, it's not very often that you'll encounter users with JavaScript disabled these days--even on mobile browsers.

To begin learning how jQuery works with CSS selectors, we'll use a structure that appears on many websites, often for navigation--the nested unordered list:

```
<ul id="selected-plays">
  <li>Comedies
    <ul>
      <li><a href="/asyoulikeit/">As You Like It</a></li>
      <li>All's Well That Ends Well</li>
      <li>A Midsummer Night's Dream</li>
      <li>Twelfth Night</li>
    </ul>
  </li>
  <li>Tragedies
    <ul>
      <li><a href="hamlet.pdf">Hamlet</a></li>
      <li>Macbeth</li>
      <li>Romeo and Juliet</li>
    </ul>
  </li>
  <li>Histories
    <ul>
      <li>Henry IV (<a href="mailto:henryiv@king.co.uk">email</a>)
         <ul>
           <li>Part I</li>
           <li>Part II</li>
         </ul>
      <li><a href="http://www.shakespeare.co.uk/henryv.htm">Henry V</a></li>
      <li>Richard II</li>
    </ul>
  </li>
</ul>
```

Downloadable code examples

You can access the example code from the following Github repository: https://github.com/PacktPublishing/Learning-jQuery-3.

Note that the first `<ul>` has an ID of `selecting-plays`, but none of the `<li>` tags have a class associated with them. Without any styles applied, the list looks like this:

The nested list appears as we would expect it to--a set of bulleted items arranged vertically and indented according to their level.

Styling list-item levels

Let's suppose that we want the top-level items, and only the top-level items--Comedies , Tragedies , and Histories --to be arranged horizontally. We can start by defining a `horizontal` class in the stylesheet:

```
.horizontal {
  float: left;
  list-style: none;
  margin: 10px;
}
```

The `horizontal` class floats the element to the left-hand side of the one following it, removes the bullet from it if it's a list item, and adds a 10-pixel margin on all sides of it.

Rather than attaching the `horizontal` class directly in our HTML, we'll add it dynamically to the top-level list items only, to demonstrate jQuery's use of selectors:

```
$(() => {
  $('#selected-plays > li')
    .addClass('horizontal');
});
```

Listing 2.1

As discussed in Chapter 1, *Getting Started*, we begin jQuery code by calling `$(() => {})`, which runs the function passed to it once the DOM has been loaded, but not before.

The second line uses the child combinator (`>`) to add the `horizontal` class to all the top-level items only. In effect, the selector inside the `$()` function is saying, "Find each list item (`li`) that is a child (`>`) of the element with an ID of `selected-plays` (`#selected-plays`)".

With the class now applied, the rules defined for that class in the stylesheet take effect, which in this case means that the list items are arranged horizontally rather than vertically. Now, our nested list looks like this:

Styling all the other items--those that are not in the top level--can be done in a number of ways. Since we have already applied the `horizontal` class to the top-level items, one way to select all sub-level items is to use a negation pseudo-class to identify all list items that do not have a class of `horizontal`:

```
$(() => {
  $('#selected-plays > li')
    .addClass('horizontal');
  $('#selected-plays li:not(.horizontal)')
    .addClass('sub-level');
});
```

Listing 2.2

This time we are selecting every list item (`<li>`) that:

- Is a descendant of the element with an ID of `selected-plays` (`#selected-plays`)
- Does not have a class of `horizontal` (`:not(.horizontal)`)

When we add the `sub-level` class to these items, they receive the shaded background defined in the stylesheet:

```
.sub-level {
  background: #ccc;
}
```

Now the nested list looks like this:

Selector specificity

Selectors in jQuery have a spectrum of specificity, from very general selectors, to very targeted selectors. The goal is to select the correct elements, otherwise your selector is broken. The tendency for jQuery beginners is to implement very specific selectors for everything. Perhaps through trial and error, they've fixed selector bugs by adding more specificity to a given selector. However, this isn't always the best solution.

Let's look at an example that increases the size of the first letter for top-level `<li>` text. Here's the style we want to apply:

```
.big-letter::first-letter {
   font-size: 1.4em;
 }
```

And here's what the list item text looks like:

As you see, Comedies , Tragedies , and Histories have the `big-letter` style applied to them as expected. In order to do this, we need a selector that's more specific than just `$('#selected-plays li')`, which would apply the style to every `<li>`, even the sub-elements. We can use change the specificity of the jQuery selector to make sure we're only getting what we expect:

```
$(() => {
  $('#selected-plays > li')
    .addClass('big-letter');

  $('#selected-plays li.horizontal')
    .addClass('big-letter');

  $('#selected-plays li:not(.sub-level)')
    .addClass('big-letter');
});
```

Listing 2.3

All three of these selectors do the same thing--apply the `big-letter` style to the top-level `<li>` elements in `#selected-plays`. The specificity is different in each of these selectors. Let's review how each of these work, and what their strengths are:

- `#selected-plays > li`: This finds `<li>` elements that are direct children of `#selected-plays`. This is easy to read, and semantically relevant to the DOM structure.

- `#selected-plays li.horizontal`: This finds `<li>` elements or sub-elements of `#selected-plays` with the `horizontal` class. This is also easy to read and enforces a particular DOM schema (applying the `horizontal` class).
- `#selected-plays li:not(.sub-level)`: This is difficult to read, inefficient, and doesn't reflect the actual DOM structure.

There are endless examples where selector-selector specificity comes up. Every application is unique, and as we just saw, there's no one correct way to implement selector specificity. What's important is that we exercise good judgement by considering the ramifications of selectors on the DOM structure, and consequently, the maintainability of our application or website.

Attribute selectors

Attribute selectors are a particularly helpful subset of CSS selectors. They allow us to specify an element by one of its HTML attributes, such as a link's `title` attribute or an image's `alt` attribute. For example, to select all images that have an `alt` attribute, we write the following:

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

Styling links

Attribute selectors accept a wildcard syntax inspired by regular expressions for identifying the value at the beginning (`^`) or end (`$`) of a string. They can also take an asterisk (`*`) to indicate the value at an arbitrary position within a string or an exclamation mark (`!`) to indicate a negated value.

Let's say we want to have different styles for different types of links. We first define the styles in our stylesheet:

```
a {
  color: #00c;
}
a.mailto {
  background: url(images/email.png) no-repeat right top;
  padding-right: 18px;
}
a.pdflink {
  background: url(images/pdf.png) no-repeat right top;
  padding-right: 18px;
}
a.henrylink {
  background-color: #fff;
  padding: 2px;
  border: 1px solid #000;
}
```

Then, we add the three classes--`mailto`, `pdflink`, and `henrylink`--to the appropriate links using jQuery.

To add a class for all e-mail links, we construct a selector that looks for all anchor elements (`a`) with an `href` attribute (`[href]`) that begins with `mailto:` (`^="mailto:"`), as follows:

```
$(() => {
  $('a[href^="mailto:"]')
    .addClass('mailto');
});
```

Listing 2.4

Because of the rules defined in the page's stylesheet, an envelope image appears after the mailto: link on the page.

To add a class for all the links to PDF files, we use the dollar sign rather than the caret symbol. This is because we're selecting links with an `href` attribute that ends with `.pdf`:

```
$(() => {
  $('a[href^="mailto:"]')
    .addClass('mailto');
  $('a[href$=".pdf"]')
    .addClass('pdflink');
});
```

Listing 2.5

The stylesheet rule for the newly added `pdflink` class causes an Adobe Acrobat icon to appear after each link to a PDF document, as shown in the following screenshot:

Attribute selectors can be combined as well. We can, for example, add the class `henrylink` to all links with an `href` value that both starts with `http` and contains `henry` anywhere:

```
$(() => {
  $('a[href^="mailto:"]')
    .addClass('mailto');
  $('a[href$=".pdf"]')
    .addClass('pdflink');
  $('a[href^="http"][href*="henry"]')
    .addClass('henrylink');
});
```

Listing 2.6

With the three classes applied to the three types of links, we should see the following:

Note the PDF icon to the right-hand side of the Hamlet link, the envelope icon next to the email link, and the white background and black border around the Henry V link.

Custom selectors

To the wide variety of CSS selectors, jQuery adds its own custom selectors. These custom selectors enhance the capabilities of CSS selectors to locate page elements in new ways.

Performance note

When possible, jQuery uses the native DOM selector engine of the browser to find elements. This extremely fast approach is not possible when custom jQuery selectors are used. For this reason, it is recommended to avoid frequent use of custom selectors when a native option is available.

Most of the custom selectors allow us to choose one or more elements from a collection of elements that we have already found. The custom selector syntax is the same as the CSS pseudo-class syntax, where the selector starts with a colon (`:`). For example, to select the second item from a set of `<div>` elements with a class of `horizontal`, we write this:

```
$('div.horizontal:eq(1)')
```

Note that `:eq(1)` selects the second item in the set because JavaScript array numbering is zero-based, meaning that it starts with zero. In contrast, CSS is one-based, so a CSS selector such as `$('div:nth-child(1)')` would select all `div` selectors that are the first child of their parent. Because it can be difficult to remember which selectors are zero based and which are one based, we should consult the jQuery API documentation at http://api.jquery.com/category/selectors/ when in doubt.

Styling alternate rows

Two very useful custom selectors in the jQuery library are `:odd` and `:even`. Let's take a look at how we can use one of them for basic table striping given the following tables:

```
<h2>Shakespeare's Plays</h2>
<table>
  <tr>
    <td>As You Like It</td>
    <td>Comedy</td>
    <td></td>
  </tr>
  <tr>
    <td>All's Well that Ends Well</td>
    <td>Comedy</td>
    <td>1601</td>
  </tr>
  <tr>
    <td>Hamlet</td>
    <td>Tragedy</td>
    <td>1604</td>
  </tr>
  <tr>
    <td>Macbeth</td>
    <td>Tragedy</td>
    <td>1606</td>
  </tr>
  <tr>
    <td>Romeo and Juliet</td>
    <td>Tragedy</td>
    <td>1595</td>
  </tr>
  <tr>
    <td>Henry IV, Part I</td>
    <td>History</td>
    <td>1596</td>
  </tr>
  <tr>
    <td>Henry V</td>
    <td>History</td>
    <td>1599</td>
  </tr>
</table>
<h2>Shakespeare's Sonnets</h2>
<table>
  <tr>
    <td>The Fair Youth</td>
    <td>1-126</td>
  </tr>
  <tr>
    <td>The Dark Lady</td>
    <td>127-152</td>
  </tr>
  <tr>
    <td>The Rival Poet</td>
    <td>78-86</td>
  </tr>
</table>
```

With minimal styles applied from our stylesheet, these headings and tables appear quite plain. The table has a solid white background, with no styling separating one row from the next, as shown in the following screenshot:

Now, we can add a style to the stylesheet for all the table rows and use an `alt` class for the odd rows:

```
tr {
  background-color: #fff;
```

```
}
.alt {
  background-color: #ccc;
}
```

Finally, we write our jQuery code, attaching the class to the odd-numbered table rows (`<tr>` tags):

```
$(() => {
  $('tr:even').addClass('alt');
});
```

Listing 2.7

But wait! Why use the `:even` selector for odd-numbered rows? Well, just as with the `:eq()` selector, the `:even` and `:odd` selectors use JavaScript's native zero-based numbering. Therefore, the first row counts as zero (even) and the second row counts as one (odd), and so on. With this in mind, we can expect our simple bit of code to produce tables that look like this:

Note that for the second table, this result may not be what we intend. Since the last row in the Plays table has the alternate gray background, the first row in the Sonnets table has the plain white background. One way to avoid this type of problem is to use the `:nth-child()` selector instead, which counts an element's position relative to its parent element rather than relative to all the elements selected so far. This selector can take a number, `odd` or `even` as its argument:

```
$(() => {
  $('tr:nth-child(odd)').addClass('alt');
});
```

Listing 2.8

As before, note that `:nth-child()` is the only jQuery selector that is one based. To achieve the same row striping as we did earlier--except with consistent behavior for the second table--we need to use `odd` rather than `even` as the argument. With this selector in place, both tables are now striped nicely, as shown in the following screenshot:

Note

The `:nth-child()` selector is a native CSS selector in modern browsers.

Finding elements based on textual content

For one final custom selector, let's suppose for some reason we want to highlight any table cell that referred to one of the Henry plays. All we have to do--after adding a class to the stylesheet to make the text bold and italicized (`.highlight {font-weight:bold; font-style: italic;}`)--is add a line to our jQuery code using the `:contains()` selector:

```
$(() => {
  $('tr:nth-child(odd)')
    .addClass('alt');
  $('td:contains(Henry)')
    .addClass('highlight');
});
```

Listing 2.9

So, now we can see our lovely striped table with the Henry plays prominently featured:

Note

It's important to note that the `:contains()` selector is case sensitive. Using `$('td:contains(henry)')` instead, without the uppercase "H", would select no cells. It's also important to note that `:contains()` can cause catastrophically bad performance, since the text of every element that matches the first part of the selector needs to be loaded and compared to our supplied argument. When `:contains()` has the potential to search hundreds of nodes for content, it's time to rethink our approach.

Admittedly, there are ways to achieve the row striping and text highlighting without jQuery--or any client-side programming, for that matter. Nevertheless, jQuery, along with CSS, is a great alternative for this type of styling in cases where the content is generated dynamically and we don't have access to either the HTML or server-side code.

Form selectors

The capabilities of custom selectors are not limited to locating elements based on their position. For example, when working with forms, jQuery's custom selectors and complementary CSS3 selectors can make short work of selecting just the elements we need. The following table describes a handful of these form selectors:

Selector	Match
`:input`	Input, text area, select, and button elements
`:button`	Button elements and input elements with a `type` attribute equal to `button`
`:enabled`	Form elements that are enabled

`:disabled` Form elements that are disabled

`:checked` Radio buttons or checkboxes that are checked

`:selected` Option elements that are selected

As with the other selectors, form selectors can be combined for greater specificity. We can, for example, select all checked radio buttons (but not checkboxes) with `$('input[type="radio"]:checked')` or select all password inputs and disabled text inputs with `$('input[type="password"], input[type="text"]:disabled')`. Even with custom selectors, we can use the same basic principles of CSS to build the list of matched elements.

Note

We have only scratched the surface of available selector expressions here. We will dive further into the topic in Chapter 9, *Advanced Selectors and Traversing*.

DOM traversal methods

The jQuery selectors that we have explored so far allow us to select a set of elements as we navigate across and down the DOM tree and filter the results. If this were the only way to select elements, our options would be somewhat limited. There are many occasions when selecting a parent or ancestor element is essential; that is where jQuery's DOM traversal methods come into play. With these methods, we can go up, down, and all around the DOM tree with ease.

Some of the methods have a nearly identical counterpart among the selector expressions. For example, the line we first used to add the `alt` class, `$('tr:even').addClass('alt')`, could be rewritten with the `.filter()` method as follows:

```
$('tr')
  .filter(':even')
  .addClass('alt');
```

For the most part, however, the two ways of selecting elements complement each other. Also, the `.filter()` method in particular has enormous power because it can take a function as its argument. The function allows us to create complex tests for whether elements should be included in the matched set. Let's suppose, for example, that we want to add a class to all external links:

```
a.external {
  background: #fff url(images/external.png) no-repeat 100% 2px;
  padding-right: 16px;
}
```

jQuery has no selector for this sort of thing. Without a filter function, we'd be forced to explicitly loop through each element, testing each one separately. With the following filter function, however, we can still rely on jQuery's implicit iteration and keep our code compact:

```
$('a')
  .filter((i, a) =>
    a.hostname && a.hostname !== location.hostname
  )
  .addClass('external');
```

Listing 2.10

The supplied function filters the set of `<a>` elements by two criteria:

- They must have an `href` attribute with a domain name (`a.hostname`). We use this test to exclude mailto links, for instance.
- The domain name that they link to (again, `a.hostname`) must not match (`!==`) the domain name of the current page (`location.hostname`).

More precisely, the `.filter()` method iterates through the matched set of elements, calling the function once for each and testing the return value. If the function returns `false`, the element is removed from the matched set. If it returns `true`, the element is kept.

With the `.filter()` method in place, the Henry V link is styled to indicate it is external:

In the next section, we'll take another look at our striped table example to see what else is possible with traversal methods.

Styling specific cells

Earlier, we added a `highlight` class to all cells containing the text Henry . To instead style the cell next to each cell containing Henry , we can begin with the selector that we have already written and simply call the `.next()` method on the result:

```
$(() => {
  $('td:contains(Henry)')
    .next()
    .addClass('highlight');
});
```

Listing 2.11

The tables should now look like this:

The `.next()` method selects only the very next sibling element. To highlight all of the cells following the one containing Henry , we could use the `.nextAll()` method instead:

```
$(() => {
  $('td:contains(Henry)')
    .nextAll()
    .addClass('highlight');
});
```

Listing 2.12

Since the cells containing Henry are in the first column of the table, this code causes the rest of the cells in these rows to be highlighted:

As we might expect, the `.next()` and `.nextAll()` methods have counterparts: `.prev()` and `.prevAll()`. Additionally, `.siblings()` selects all other elements at the same DOM level, regardless of whether they come before or after the previously selected element.

To include the original cell (the one that contains Henry) along with the cells that follow, we can add the `.addBack()` method:

```
$(() => {
  $('td:contains(Henry)')
    .nextAll()
    .addBack()
    .addClass('highlight');
});
```

Listing 2.13

With this modification in place, all of the cells in the row get their styles from the `highlight` class:

There are a multitude of selector and traversal-method combinations by which we can select the same set of elements. Here, for example, is another way to select every cell in each row where at least one of the cells contains Henry :

```
$(() => {
  $('td:contains(Henry)')
    .parent()
    .children()
    .addClass('highlight');
});
```

Listing 2.14

Rather than traversing across to sibling elements, we travel up one level in the DOM to the `<tr>` tag with `.parent()` and then select all of the row's cells with `.children()`.

Chaining

The traversal method combinations that we have just explored illustrate jQuery's chaining capability. With jQuery, it is possible to select multiple sets of elements and do multiple things with them, all within a single line of code. This chaining not only helps keep jQuery code concise, but it can also improve a script's performance when the alternative is to respecify a selector.

Tip

How chaining works Almost all jQuery methods return a jQuery object and so can have more jQuery methods applied to the result. We will explore the inner workings of chaining in Chapter 8, *Developing Plugins*.

It is also possible to break a single line of code into multiple lines for greater readability, as we've been doing throughout this chapter so far. For example, a single chained sequence of methods could be written in one line:

```
$('td:contains(Henry)').parent().find('td:eq(1)')
    .addClass('highlight').end().find('td:eq(2)')
                            .addClass('highlight');
```

Listing 2.15

This same sequence of methods could also be written in seven lines:

```
$('td:contains(Henry)') // Find every cell containing "Henry"
  .parent() // Select its parent
  .find('td:eq(1)') // Find the 2nd descendant cell
  .addClass('highlight') // Add the "highlight" class
  .end() // Return to the parent of the cell containing "Henry"
  .find('td:eq(2)') // Find the 3rd descendant cell
  .addClass('highlight'); // Add the "highlight" class
```

Listing 2.16

The DOM traversal in this example is contrived and not recommended. There are clearly simpler, more direct methods at our disposal. The point of the example is simply to demonstrate the tremendous flexibility that chaining affords us, especially when many calls need to be made.

Chaining can be like speaking a whole paragraph's worth of words in a single breath--it gets the job done quickly, but it can be hard for someone else to understand. Breaking it up into multiple lines and adding judicious comments can save more time in the long run.

Iterating over jQuery objects

New in jQuery 3 is the ability to iterate over jQuery objects using a `for...of` loop. This by itself isn't a big deal. For one thing, it's rare that we need to explicitly iterate over jQuery objects, especially when the same result is possible by using implicit iteration in jQuery functions. But sometimes, explicit iteration can't be avoided. For example, imaging you need to reduce an array of elements (a jQuery object) to an array of string values. The `each()` function is a tool of choice here:

```
const eachText = [];

$('td')
  .each((i, td) => {
    if (td.textContent.startsWith('H')) {
      eachText.push(td.textContent);
    }
  });

console.log('each', eachText);
 // ["Hamlet", "Henry IV, Part I", "History", "Henry V", "History"]
```

Listing 2.17

We start off with an array of `<td>` elements, the result of our `$('td')` selector. We then reduce it to an array of strings by passing the `each()` function a callback that pushes each string that starts with "H" onto the `eachText` array. There's nothing wrong with this approach, but having callback functions for such a straightforward task seems like a bit much. Here's the same functionality using `for...of` syntax:

```
 const forText = [];

 for (let td of $('td')) {
   if (td.textContent.startsWith('H')) {
     forText.push(td.textContent);
   }
 }

 console.log('for', forText);
 // ["Hamlet", "Henry IV, Part I", "History", "Henry V", "History"]
```

Listing 2.18

We can now reduce jQuery objects using simple for loops and if statements. We'll revisit this `for...of` approach later on in the book for more advanced usage scenarios involving generators.

Accessing DOM elements

Every selector expression and most jQuery methods return a jQuery object. This is almost always what we want because of the implicit iteration and chaining capabilities that it affords.

Still, there may be points in our code when we need to access a DOM element directly. For example, we may need to make a resulting set of elements available to another JavaScript library, or we might need to access an element's tag name, which is available as a property of the DOM element. For these admittedly rare situations, jQuery provides the `.get()` method. To access the first DOM element referred to by a jQuery object, for example, we would use `.get(0)`. So, if we want to know the tag name of an element with an ID of `my-element`, we would write:

```
$('#my-element').get(0).tagName;
```

For even greater convenience, jQuery provides a shorthand for `.get()`. Instead of writing the previous line, we can use square brackets immediately following the selector:

```
$('#my-element')[0].tagName;
```

It's no accident that this syntax appears to treat the jQuery object as an array of DOM elements; using the square brackets is like peeling away the jQuery layer to get at the node list, and including the index (in this case, `0`) is like plucking out the DOM element itself.

Summary

With the techniques that we covered in this chapter, we should now be able to locate sets of elements on the page in a variety of ways. In particular, we learned how to style top-level and sub-level items in a nested list using basic CSS selectors, how to apply different styles to different types of links using attribute selectors, add rudimentary striping to a table using either the custom jQuery selectors `:odd` and `:even` or the advanced CSS selector `:nth-child()`, and highlight text within certain table cells by chaining jQuery methods.

So far, we have been using the `$(() => {})` document ready handler to add a class to a matched set of elements. In the next chapter, we'll explore ways in which to add a class in response to a variety of user-initiated events.

Further reading

The topic of selectors and traversal methods will be explored in more detail in Chapter 9, *Advanced Selectors and Traversing*. A complete list of jQuery's selectors and traversal methods is available in Appendix B of this book and in the official jQuery documentation at http://api.jquery.com/.

Exercises

Challenge exercises may require the use of the official jQuery documentation at http://api.jquery.com/:

1. Add a class of `special` to all of the `<li>` elements at the second level of the nested list.
2. Add a class of `year` to all the table cells in the third column of a table.
3. Add the class `special` to the first table row that has the word Tragedy in it.
4. Here's a challenge for you. Select all the list items (`<li>s`) containing a link (`<a>`). Add the class `afterlink` to the sibling list items that follow the ones selected.
5. Here's another challenge for you. Add the class `tragedy` to the closest ancestor `<ul>` of any `.pdf` link.

NEXT :Handling Events❯
Chapter 3

Handling Events

Adam Boduch,
Jonathan Chaffer,
Karl Swedberg

JavaScript has several built-in ways of reacting to user interaction and other events. To make a page dynamic and responsive, we need to harness this capability so that we can, at the appropriate times, use the jQuery techniques you learned so far and the other tricks you'll learn later. While we could do this with vanilla JavaScript, jQuery enhances and extends the basic event-handling mechanisms to give them a more elegant syntax while making them more powerful at the same time.

In this chapter, we will cover:

- Executing JavaScript code when the page is ready
- Handling user events, such as mouse clicks and keystrokes
- The flow of events through the document, and how to manipulate that flow
- Simulating events as if the user initiated them

Performing tasks on page load

We have already seen how to make jQuery react to the loading of a web page. The `$(() => {})` event handler can be used to run code that depends on HTML elements, but there's a bit more to be said about it.

Timing of code execution

In Chapter 1, *Getting Started*, we noted that `$(() => {})` was jQuery's primary way to perform tasks on page load. It is not, however, the only method at our disposal. The native `window.onload` event can do the same thing. While the two methods are similar, it is important to recognize their difference in timing, even though it can be quite subtle depending on the number of resources being loaded.

The `window.onload` event fires when a document is completely downloaded to the browser. This means that every element on the page is ready to be manipulated by JavaScript, which is a boon for writing feature-rich code without worrying about load order.

On the other hand, a handler registered using `$(() => {})` is invoked when the DOM is completely ready for use. This also means that all elements are accessible by our scripts, but does not mean that every associated file has been downloaded. As soon as the HTML file has been downloaded and parsed into a DOM tree, the code can run.

Style loading and code execution

To ensure that the page has also been styled before the JavaScript code executes, it is good practice to place the `<link rel="stylesheet">` and `<style>` tags prior to any `<script>` tags within the document's `<head>` element.

Consider, for example, a page that presents an image gallery; such a page may have many large images on it, which we can hide, show, move, and otherwise manipulate with jQuery. If we set up our interface using the `onload` event, users will have to wait until each and every image is completely downloaded before they can use those features. Even worse, if behaviors are not yet attached to elements that have default behaviors (such as links), user interactions could produce unintended outcomes. However, when we use `$(() => {})` for the setup, the interface is ready to be used earlier with the correct behavior.

What is loaded and what is not?

Using `$(() => {})` is almost always preferred over using an `onload` handler, but we need to keep in mind that, because supporting files may not have loaded, attributes such as image height and width are not necessarily available at this time. If these are needed, we may at times also choose to implement an `onload` handler; the two mechanisms can coexist peacefully.

Handling multiple scripts on one page

The traditional mechanism for registering event handlers through JavaScript (rather than adding handler attributes right in the HTML content) is to assign a function to the DOM element's corresponding property. For example, suppose we had defined the following function:

```
function doStuff() {
  // Perform a task...
}
```


We could then either assign it within our HTML markup:

```
<body onload="doStuff();">
```

Or, we could assign it from within JavaScript code:

```
window.onload = doStuff;
```

Both of these approaches will cause the function to execute when the page is loaded. The advantage of the second is that the behavior is cleanly separated from the markup.

Referencing versus calling functions

When we assign a function as a handler, we use the function name but omit the trailing parentheses. With the parentheses, the function is called immediately; without the parantheses, the name simply identifies, or *references*, the function, and can be used to call it later.

With one function, this strategy works quite well. However, suppose we have a second function as follows:

```
function doOtherStuff() {
  // Perform another task...
}
```

We could then attempt to assign this function to run on page load:

```
window.onload = doOtherStuff;
```

However, this assignment trumps the first one. The `.onload` attribute can only store one function reference at a time, so we can't add to the existing behavior.

The `$(() => {})` mechanism handles this situation gracefully. Each call adds the new function to an internal queue of behaviors; when the page is loaded, all of the functions will execute. The functions will run in the order in which they were registered.

Note

To be fair, jQuery doesn't have a monopoly on workarounds to this issue. We can write a JavaScript function that calls the existing `onload` handler, then calls a passed-in handler. This approach avoids conflicts between rival handlers like `$(() => {})` does, but lacks some of the other benefits we have discussed. In modern browsers, the `DOMContentLoaded` event can be triggered with the W3C standard `document.addEventListener()` method. However, the `$(() => {})` is more concise and elegant.

Passing an argument to the document ready callback

In some cases, it may prove useful to use more than one JavaScript library on the same page. Since many libraries make use of the `$` identifier (since it is short and convenient), we need a way to prevent collisions between libraries.

Fortunately, jQuery provides a method called `jQuery.noConflict()` to return control of the `$` identifier back to other libraries. Typical usage of `jQuery.noConflict()` follows the following pattern:

```
<script src="prototype.js"></script>
<script src="jquery.js"></script>
<script>
  jQuery.noConflict();
</script>
<script src="myscript.js"></script>
```

First, the other library (`prototype.js` in this example) is included. Then, `jquery.js` itself is included, taking over `$` for its own use. Next, a call to `.noConflict()` frees up `$`, so that control of it reverts to the first included library (`prototype.js`). Now in our custom script, we can use both libraries, but whenever we want to use a jQuery method, we need to write `jQuery` instead of `$` as an identifier.

The `$(() => {})` document ready handler has one more trick up its sleeve to help us in this situation. The callback function we pass to it can take a single parameter--the `jQuery` object itself. This allows us to effectively rename it without fear of conflicts using the following syntax:

```
jQuery(($) => {
  // In here, we can use $ like normal!
});
```

Handling simple events

There are other times, apart from the loading of the page, at which we might want to perform a task. Just as JavaScript allows us to intercept the page load event with `<body onload="">` or `window.onload`, it provides similar hooks for user-initiated events such as mouse clicks (`onclick`), form fields being modified (`onchange`), and windows changing size (`onresize`). When assigned directly to elements in the DOM, these hooks have similar drawbacks to the ones we outlined for `onload`. Therefore, jQuery offers an improved way of handling these events as well.

A simple style switcher

To illustrate some event handling techniques, suppose we wish to have a single page rendered in several different styles based on user input; we will present buttons that allow the user to toggle between a normal view, a view in which the text is constrained to a narrow column, and a view with large print for the content area.

Progressive enhancement

In a real-world example, a good web citizen will employ the principle of progressive enhancement here. In Chapter 5, *Manipulating the DOM*, you will learn how we can inject content like this style switcher right from our jQuery code, so that users without JavaScript available will not see nonfunctional controls.

The HTML markup for the style switcher is as follows:

```
<div id="switcher" class="switcher">
  <h3>Style Switcher</h3>
  <button id="switcher-default">
    Default
  </button>
  <button id="switcher-narrow">
    Narrow Column
  </button>
  <button id="switcher-large">
    Large Print
  </button>
</div>
```

Getting the example code

You can access the example code from the following GitHub repository: https://github.com/PacktPublishing/Learning-jQuery-3.

Combined with the rest of the page's HTML markup and some basic CSS, we get a page that looks like the following:

To begin with, we'll make the Large Print button operate. We need a bit of CSS to implement our alternative view of the page as follows:

```
body.large .chapter {
  font-size: 1.5em;
}
```

Our goal, then, is to apply the `large` class to the `<body>` tag. This will allow the stylesheet to reformat the page appropriately. Using what you learned in Chapter 2, *Selecting Elements*, we already know the statement needed to accomplish this:

```
$('body').addClass('large');
```

However, we want this to occur when the button is clicked, not when the page is loaded as we have seen so far. To do this, we'll introduce the `.on()` method. This method allows us to specify any DOM event and to attach a behavior to it. In this case, the event is called `click`, and the behavior is a function consisting of our previous one liner:

```
$(() => {
  $('#switcher-large')
    .on('click', () => {
      $('body').addClass('large');
    });
});
```

Listing 3.1

Now when the button gets clicked on, our code runs and the text is enlarged:

That's all there is to binding a behavior to an event. The advantages we discussed with the `$(() => {})` document ready handler apply here as well. Multiple calls to `.on()` coexist nicely, appending additional behaviors to the same event as necessary.

This isn't necessarily the most elegant or efficient way to accomplish this task. As we proceed through this chapter, we will extend and refine this code into something we can be proud of.

Enabling the other buttons

We now have a Large Print button that works as advertised, but we need to apply similar handling to the other two buttons (Default and Narrow Column) to make them perform their tasks. This is straightforward: we use `.on()` to add a `click` handler to each of them, removing and adding classes as necessary. The new code reads as follows:

```
$(() => {
  $('#switcher-default')
    .on('click', () => {
      $('body')
        .removeClass('narrow')
        .removeClass('large');
    });

  $('#switcher-narrow')
    .on('click', () => {
      $('body')
        .addClass('narrow')
        .removeClass('large');
    });

  $('#switcher-large')
    .on('click', () => {
      $('body')
        .removeClass('narrow')
        .addClass('large');
    });
});
```

Listing 3.2

This is combined with a CSS rule for the `narrow` class:

```
body.narrow .chapter {
  width: 250px;
}
```

Now, after clicking the Narrow Column button, its corresponding CSS is applied and the text gets laid out differently:

Clicking on Default removes both class names from the `<body>` tag, returning the page to its initial rendering.

Making use of event handler context

Our switcher is behaving correctly, but we are not giving the user any feedback about which button is currently active. Our approach for handling this will be to apply the `selected` class to the button when it is clicked, and to remove this class from the other buttons. The `selected` class simply makes the button's text bold:

```
.selected {
  font-weight: bold;
}
```

We could accomplish this class modification as we did previously by referring to each button by ID and applying or removing classes as necessary, but, instead, we'll explore a more elegant and scalable solution that exploits the context in which event handlers run.

When any event handler is triggered, the keyword `this` refers to the DOM element to which the behavior was attached. Earlier we noted that the `$()` function could take a DOM element as its argument; this is one of the key reasons why that facility is available. By writing `$(this)` within the event handler, we create a jQuery object corresponding to the element, and we can act on it just as if we had located it with a CSS selector.

With this in mind, we can write the following:

```
$(this).addClass('selected');
```

Placing this line in each of the three handlers will add the class when a button is clicked. To remove the class from the other buttons, we can take advantage of jQuery's implicit iteration feature, and write:

```
$('#switcher button').removeClass('selected');
```

This line removes the class from every button inside the style switcher.

We should also add the class to the Default button when the document is ready. So, placing these in the correct order, the code is as follows:

```
$(() => {
  $('#switcher-default')
    .addClass('selected')
    .on('click', function() {
      $('body')
        .removeClass('narrow');
        .removeClass('large');
      $('#switcher button')
        .removeClass('selected');
      $(this)
        .addClass('selected');
    });

  $('#switcher-narrow')
    .on('click', function() {
        $('body')
          .addClass('narrow')
          .removeClass('large');
        $('#switcher button')
          .removeClass('selected');
        $(this)
          .addClass('selected');
  });

  $('#switcher-large')
    .on('click', function() {
      $('body')
        .removeClass('narrow')
        .addClass('large');
      $('#switcher button')
        .removeClass('selected');
      $(this)
        .addClass('selected');
  });
});
```

Listing 3.3

Now the style switcher gives appropriate feedback.

Generalizing the statements by using the handler context allows us to be yet more efficient. We can factor the highlighting routine out into a separate handler, as shown in *Listing 3.4*, because it is the same for all three buttons:

```
$(() => {
  $('#switcher-default')
    .addClass('selected')
    .on('click', function() {
      $('body')
        .removeClass('narrow')
        .removeClass('large');
    });
  $('#switcher-narrow')
    .on('click', () => {
      $('body')
        .addClass('narrow')
        .removeClass('large');
    });

  $('#switcher-large')
    .on('click', () => {
      $('body')
        .removeClass('narrow')
        .addClass('large');
    });

  $('#switcher button')
    .on('click', function() {
      $('#switcher button')
        .removeClass('selected');
      $(this)
        .addClass('selected');
    });
});
```

Listing 3.4

This optimization takes advantage of three jQuery features we have already discussed. First, **implicit iteration** is once again useful when we bind the same `click` handler to each button with a single call to `.on()`. Second, **behavior queuing** allows us to bind two functions to the same click event without the second overwriting the first.

Note

When an event handler function references its context using `this`, you can't use an arrow function (`() => {}`). These functions have a **lexical context**. This means that when jQuery attempts to set the context as the element that triggered the event, it doesn't work.

Consolidating code using event context

The code optimization we've just completed is an example of **refactoring**--modifying existing code to perform the same task in a more efficient or elegant way. To explore further refactoring opportunities, let's look at the behaviors we have bound to each button. The `.removeClass()` method's parameter is optional; when omitted, it removes all classes from the element. We can streamline our code a bit by exploiting this as follows:

```
$(() => {
  $('#switcher-default')
    .addClass('selected')
    .on('click', () => {
      $('body').removeClass();
    });
  $('#switcher-narrow')
    .on('click', () => {
      $('body')
        .removeClass()
        .addClass('narrow');
    });

  $('#switcher-large')
    .on('click', () => {
      $('body')
        .removeClass()
        .addClass('large');
    });

  $('#switcher button')
    .on('click', function() {
      $('#switcher button')
        .removeClass('selected');
      $(this)
        .addClass('selected');
    });
});
```

Listing 3.5

Note that the order of operations has changed a bit to accommodate our more general class removal; we need to execute `.removeClass()` first so that it doesn't undo the call to `.addClass()`, which we perform in the same breath.

Tip

We can only safely remove all classes because we are in charge of the HTML in this case. When we are writing code for reuse (such as for a plugin), we need to respect any classes that might be present and leave them intact.

Now we are executing some of the same code in each of the button's handlers. This can be easily factored out into our general button `click` handler:

```
$(() => {
  $('#switcher-default')
    .addClass('selected');
  $('#switcher button')
    .on('click', function() {
      $('body')
        .removeClass();
      $('#switcher button')
        .removeClass('selected');
      $(this)
        .addClass('selected');
    });

  $('#switcher-narrow')
    .on('click', () => {
      $('body')
        .addClass('narrow');
    });

  $('#switcher-large')
    .on('click', () => {
      $('body')
        .addClass('large');
    });
});
```

Listing 3.6

Note that we need to move the general handler above the specific ones now. The `.removeClass()` call needs to happen before `.addClass()` executes, and we can count on this because jQuery always triggers event handlers in the order in which they were registered.

Finally, we can get rid of the specific handlers entirely by, once again, exploiting **event context**. Since the context keyword `this` gives us a DOM element rather than a jQuery object, we can use native DOM properties to determine the ID of the element that was clicked. We can thus bind the same handler to all the buttons, and within the handler perform different actions for each button:

```
$(() => {
  $('#switcher-default')
    .addClass('selected');
  $('#switcher button')
    .on('click', function() {
      const bodyClass = this.id.split('-')[1];
      $('body')
```

```
        .removeClass()
        .addClass(bodyClass);
      $('#switcher button')
        .removeClass('selected');
      $(this)
        .addClass('selected');
    });
});
```

Listing 3.7

The value of the `bodyClass` variable will be `default`, `narrow`, or `large`, depending on which button is clicked. Here, we are departing somewhat from our previous code; in that we are adding a `default` class to `<body>` when the user clicks on `<button id="switcher-default">`. While we do not need this class applied, it isn't causing any harm either, and the reduction of code complexity more than makes up for an unused class name.

Shorthand events

Binding a handler for an event (such as a simple `click` event) is such a common task that jQuery provides an even terser way to accomplish it; shorthand event methods work in the same way as their `.on()` counterparts with fewer keystrokes.

For example, our style switcher could be written using `.click()` instead of `.on()` as follows:

```
$(() => {
  $('#switcher-default')
    .addClass('selected');

  $('#switcher button')
    .click(function() {
      const bodyClass = this.id.split('-')[1];
      $('body')
        .removeClass()
        .addClass(bodyClass);
      $('#switcher button')
        .removeClass('selected');
      $(this)
        .addClass('selected');
  });
});
```

Listing 3.8

Shorthand event methods such as the previous one exist for the other standard DOM events such as `blur`, `keydown`, and `scroll` as well. Each shortcut method binds a handler to the event with the corresponding name.

Showing and hiding page elements

Suppose that we wanted to be able to hide our style switcher when it is not needed. One convenient way to hide page elements is to make them collapsible. We will allow one click on the label to hide the buttons, leaving the label alone. Another click on the label will restore the buttons. We need another class that will hide buttons:

```
.hidden {
  display: none;
}
```

We could implement this feature by storing the current state of the buttons in a variable and checking its value each time the label is clicked to know whether to add or remove the hidden class on the buttons. However, jQuery provides an easy way for us to add or remove a class depending on whether that class is already present--the `.toggleClass()` method:

```
$(() => {
  $('#switcher h3')
    .click(function() {
      $(this)
        .siblings('button')
        .toggleClass('hidden');
    });
});
```

Listing 3.9

After the first click, the buttons are all hidden:

A second click then returns them to visibility:

Once again, we rely on implicit iteration, this time to hide all the buttons - siblings of the `<h3>` - in one fell swoop.

Event propagation

In illustrating the ability of the `click` event to operate on normally non-clickable page elements, we have crafted an interface that doesn't indicate that the style switcher label--just an `<h3>` element-is actually a *live* part of the page awaiting user interaction. To remedy this, we can give it a rollover state, making it clear that it interacts in some way with the mouse:

```
.hover {
  cursor: pointer;
  background-color: #afa;
}
```

The CSS specification includes a pseudo-class called `:hover`, which allows a stylesheet to affect an element's appearance when the user's mouse cursor hovers over it. This would certainly solve our problem in this instance, but instead, we will take this opportunity to introduce jQuery's `.hover()` method, which allows us to use JavaScript to change an element's styling--and indeed, perform any arbitrary action--both when the mouse cursor enters the element and when it leaves the element.

The `.hover()` method takes two function arguments, unlike the simple event methods we have so far encountered. The first function will be executed when the mouse cursor enters the selected element, and the second is fired when the cursor leaves. We can modify the classes applied to the buttons at these times to achieve a rollover effect:

```
$(() => {
  $('#switcher h3')
    .hover(function() {
      $(this).addClass('hover');
    }, function() {
      $(this).removeClass('hover');
    });
});
```

Listing 3.10

We once again use implicit iteration and event context for short and simple code. Now when hovering over the `<h3>` element, we see our class applied:

The use of `.hover()` also means we avoid headaches caused by event propagation in JavaScript. To understand this, we need to take a look at how JavaScript decides which element gets to handle a given event.

The journey of an event

When an event occurs on a page, an entire hierarchy of DOM elements gets a chance to handle the event. Consider a page model like the following:

```
<div class="foo">
  <span class="bar">
    <a href="http://www.example.com/">
      The quick brown fox jumps over the lazy dog.
    </a>
  </span>
  <p>
    How razorback-jumping frogs can level six piqued gymnasts!
  </p>
</div>
```

We then visualize the code as a set of nested elements:

For any event, there are multiple elements that could logically be responsible for reacting. When the link on this page is clicked, for example, the `<div>`, `<span>`, and `<a>` elements should all get the opportunity to respond to the click. After all, these three elements are all under the user's mouse cursor at the time. The `<p>` element, on the other hand, is not part of this interaction at all.

One strategy for allowing multiple elements to respond to a user interaction is called **event capturing**. With event capturing, the event is first given to the most all-encompassing element, and then to progressively more specific ones. In our example, this means that first the `<div>` element gets passed the event, then the `<span>` element, and finally the `<a>` element, as shown in the following figure:

The opposite strategy is called **event bubbling**. The event gets sent to the most specific element, and after this element has an opportunity to react, the event **bubbles up** to more general elements. In our example, the `<a>` element would be handed the event first, and then the `<span>` and `<div>` elements in that order, as shown in the following figure:

Unsurprisingly, different browser developers originally decided on different models for event propagation. The DOM standard that was eventually developed thus specified that both strategies should be used: first the event is captured from general elements to specific ones, and then the event bubbles back up to the top of the DC tree. Event handlers can be registered for either part of the process.

To provide consistent and easy-to-understand behavior, jQuery always registers event handlers for the bubbling phase of the model. We can always assume that the most specific element will get the first opportunity to respond to any event.

Side effects of event bubbling

Event bubbling can cause unexpected behavior, especially when the wrong element responds to a `mouseover` or `mouseout` event. Consider a `mouseout` event handler attached to the `<div>` element in our example. When the user's mouse cursor exits the `<div>` element, the `mouseout` handler is run as anticipated. Since this is at the top of the hierarchy, no other elements get the event. On the other hand, when the cursor exits the `<a>` element, a `mouseout` event is sent to that. This event will then bubble up to the `<span>` element and then to the `<div>` element, firing the same event handler. This bubbling sequence is unlikely to be desired.

The `mouseenter` and `mouseleave` events, either bound individually or combined in the `.hover()` method, are aware of these bubbling issues and, when we use them to attach events, we can ignore the problems caused by the wrong element getting a `mouseover` or `mouseout` event.

The `mouseout` scenario just described illustrates the need to constrain the scope of an event. While `.hover()` handles this specific case, we will encounter other situations in which we need to limit an event spatially (preventing the event from being sent to certain elements) or temporally (preventing the event from being sent at certain times).

Altering the journey - the event object

We have already seen one situation in which event bubbling can cause problems. To show a case in which `.hover()` does not help our cause, we'll alter the collapsing behavior that we implemented earlier.

Suppose we wish to expand the clickable area that triggers the collapsing or expanding of the style switcher. One way to do this is to move the event handler from the label, `<h3>`, to its containing `<div>` element. In *Listing 3.9*, we added a `click` handler to `#switcher h3`; we will attempt this change by attaching the handler to `#switcher` instead:

```
$(() => {
  $('#switcher')
    .click(() => {
      $('#switcher button').toggleClass('hidden');
    });
});
```

Listing 3.11

This alteration makes the entire area of the style switcher clickable to toggle its visibility. The downside is that clicking on a button also collapses the style switcher after the style on the content has been altered. This is due to event bubbling; the event is first handled by the buttons, then passed up through the DOM tree until it reaches the `<div id="switcher">` element, where our new handler is activated and hides the buttons.

To solve this problem, we need access to the `event` object. This is a DOM construct that is passed to each element's event handler when it is invoked. It provides information about the event, such as where the mouse cursor was at the time of the event. It also provides some methods that can be used to affect the progress of the event through the DOM.

Event object reference

For detailed information about jQuery's implementation of the event object and its properties, see http://api.jquery.com/category/events/event-object/.

To use the event object in our handlers, we only need to add a parameter to the function:

```
$(() => {
  $('#switcher')
    .click(function(event) {
      $('#switcher button').toggleClass('hidden');
    });
});
```

Note that we have named this parameter `event` because it is descriptive, not because we need to. Naming it `flapjacks` or anything else for that matter would work just as well.

Event targets

Now we have the event object available to us as `event` within our handler. The property `event.target` can be helpful in controlling *where* an event takes effect. This property is a part of the DOM API, but is not implemented in some older browser versions; jQuery extends the event object as necessary to provide the property in every browser. With `.target`, we can determine which element in the DOM was the first to receive the event. In the case of a `click` event, this will be the actual item clicked on. Remembering that `this` gives us the DOM element handling the event, we can write the following code:

```
$(() => {
  $('#switcher')
    .click(function(event) {
      if (event.target == this) {
        $(this)
          .children('button')
          .toggleClass('hidden');
      }
    });
});
```

Listing 3.12

This code ensures that the item clicked on was `<div id="switcher">`, not one of its sub-elements. Now, clicking on buttons will not collapse the style switcher, but clicking on the switcher's background *will*. However, clicking on the label, `<h3>`, now does nothing, because it, too, is a sub-element. Instead of placing this check here can modify the behavior of the buttons to achieve our goals.

Stopping event propagation

The event object provides the `.stopPropagation()` method, which can halt the bubbling process completely for the event. Like `.target`, this method is a basic DOM feature, but using the jQuery implementation will hide any browser inconsistencies from our code.

We'll remove the `event.target == this` check we just added, and instead add some code in our buttons' `click` handlers:

```
$(() => {
  $('#switcher')
    .click((e) => {
      $(e.currentTarget)
        .children('button')
        .toggleClass('hidden');
    });
});
$(() => {
  $('#switcher-default')
    .addClass('selected');
  $('#switcher button')
    .click((e) => {
      const bodyClass = e.target.id.split('-')[1];

      $('body')
        .removeClass()
        .addClass(bodyClass);
      $(e.target)
        .addClass('selected')
        .removeClass('selected');
```

```
      e.stopPropagation();
    });
});
```

Listing 3.13

As before, we need to add an event parameter to the function we're using as the `click` handler: `e`. Then, we simply call `e.stopPropagation()` to prevent any other DOM element from responding to the event. Now our click is handled by the buttons, and only the buttons; clicks anywhere else on the style switcher will collapse or expand it.

Preventing default actions

If our `click` event handler was registered on a link element (`<a>`) rather than a generic `<button>` element outside of a form, we would face another problem. When a user clicks on a link, the browser loads a new page. This behavior is not an event handler in the same sense as the ones we have been discussing; instead, this is the default action for a click on a link element. Similarly, when the *nter* key is pressed while the user is editing a form, the `submit` event may be triggered on the form, but then the form submission actually occurs after this.

If these default actions are undesired, calling `.stopPropagation()` on the event will not help. These actions don't occur in the normal flow of event propagation. Instead, the `.preventDefault()` method serves to stop the event in its tracks before the default action is triggered.

Tip

Calling `.preventDefault()` is often useful after we have done some tests on the environment of the event. For example, during a form submission, we might wish to check that required fields are filled in and prevent the default action only if they are not. With links, we can check if some precondition has been met before allowing the `href` to be followed, in essence, disabling the link under some circumstances.

Event propagation and default actions are independent mechanisms; either of them can be stopped while the other still occurs. If we wish to halt both, we can return `false` at the end of our event handler, which is a shortcut for calling both `.stopPropagation()` and `.preventDefault()` on the event.

Delegating events

Event bubbling isn't always a hindrance; we can often use it to great benefit. One great technique that exploits bubbling is called **event delegation**. With it, we can use an event handler on a single element to do the work of many.

In our example, there are just three `<button>` elements that have attached `click` handlers. But what if there were many more than three? This is more common than you might think. Consider, for example, a large table of information in which each row has an interactive item requiring a `click` handler. Implicit iteration makes assigning all of these `click` handlers easy, but performance can suffer because of the looping being done internally to jQuery, and because of the memory footprint of maintaining all the handlers.

Instead, we can assign a single `click` handler to an ancestor element in the DOM. An uninterrupted `click` event will eventually reach the ancestor due to event bubbling, and we can do our work there.

As an example, let's apply this technique to our style switcher (even though the number of items does not demand the approach). As seen in *Listing 3.12* previously, we can use the `e.target` property to check which element is under the mouse cursor when the `click` event occurs.

```
$(() => {
  $('#switcher')
    .click((e) => {
      if ($(event.target).is('button')) {
        const bodyClass = e.target.id.split('-')[1];

        $('body')
          .removeClass()
          .addClass(bodyClass);
        $(e.target)
          .addClass('selected')
          .removeClass('selected');

        e.stopPropagation();
      }
    });
});
```

Listing 3.14

We've used a new method here called `.is()`. This method accepts the selector expressions we investigated in the previous chapter and tests the current jQuery object against the selector. If at least one element in the set is matched by the selector, `.is()` returns `true`. In this case, `$(e.target).is('button')` asks whether the element clicked is a `<button>` element. If so, we proceed with the previous code, with one significant alteration: the keyword `this` now refers to `<div id="switcher">`, so every time we are interested in the clicked button, we must now refer to it with `e.target`.

Tip

.is() and .hasClass() We can test for the presence of a class on an element with `.hasClass()`. The `.is()` method is more flexible, however, and can test any selector expression.

We have an unintentional side-effect from this code, however. When a button is clicked now, the switcher collapses, as it did before we added the call to `.stopPropagation()`. The handler for the switcher visibility toggle is now bound to the same element as the handler for the buttons, so halting the event bubbling does not stop the toggle from being triggered. To sidestep this issue, we can remove the `.stopPropagation()` call and instead add another `.is()` test. Also, since we're making the entire switcher `<div>` element clickable, we ought to toggle the `hover` class while the user's mouse is over any part of it:

```
$(() => {
  const toggleHover = (e) => {
    $(e.target).toggleClass('hover');
  };

  $('#switcher')
```

```
    .hover(toggleHover, toggleHover);
});

$(() => {
  $('#switcher')
    .click((e) => {
      if (!$(e.target).is('button')) {
        $(e.currentTarget)
          .children('button')
          .toggleClass('hidden');
      }
    });
});

$(() => {
  $('#switcher-default')
    .addClass('selected');
  $('#switcher')
    .click((e) => {
      if ($(e.target).is('button')) {
        const bodyClass = e.target.id.split('-')[1];

        $('body')
          .removeClass()
          .addClass(bodyClass);
        $(e.target)
          .addClass('selected')
          .siblings('button')
          .removeClass('selected');
      }
  });
});
```

Listing 3.15

This example is a bit over complicated for its size, but as the number of elements with event handlers increases, so does event delegation's benefit. Also, we can avoid some of the code repetition by combining the two `click` handlers and using a single `if-else` statement for the `.is()` test:

```
$(() => {
  $('#switcher-default')
    .addClass('selected');
  $('#switcher')
    .click((e) => {
      if ($(e.target).is('button')) {
        const bodyClass = e.target.id.split('-')[1];
        $('body')
          .removeClass()
          .addClass(bodyClass);
        $(e.target)
          .addClass('selected')
          .removeClass('selected');
      } else {
        $(e.currentTarget)
          .children('button')
          .toggleClass('hidden');
      }
    });
});
```

Listing 3.16

While our code could still use some fine tuning, it is approaching a state at which we can feel comfortable using it for what we set out to do. Nevertheless, for the sake of learning more about jQuery's event handling, we'll back up to *Listing 3.15* and continue to modify that version of the code.

Tip

Event delegation is also useful in other situations we'll see later, such as when new elements are added by DOM manipulation methods (Chapter 5, *Manipulating the DOM*) or Ajax routines (Chapter 6, *Sending Data with Ajax*).

Using built-in event delegation capabilities

Because event delegation can be helpful in so many situations, jQuery includes a set of tools to aid developers in using this technique. The `.on()` method we have already discussed can perform event delegation when provided with appropriate parameters:

```
$(() => {
  $('#switcher-default')
    .addClass('selected');
  $('#switcher')
   .on('click', 'button', (e) => {
     const bodyClass = e.target.id.split('-')[1];

     $('body')
       .removeClass()
       .addClass(bodyClass);
     $(e.target)
       .addClass('selected')
       .siblings('button')
       .removeClass('selected');

     e.stopPropagation();
   })
   .on('click', (e) => {
     $(e.currentTarget)
```

```
      .children('button')
      .toggleClass('hidden');
  });
});
```

Listing 3.17

This is looking pretty good now. We have two really simple handlers for all click events in switcher feature. We added a selector expression to the `.on()` method as the second argument. Specifically, we want to make sure that any elements that bubble click events up to `#switch` are in fact button elements. This is better than writing a bunch of logic in the event handler to determine how to handle the event based on the element that generated it.

We did have to add a call to `e.stopPropagation()`. The reason is so that the second click handler, the one that handles toggling the button visibility, doesn't have to worry about checking where the event came from. It's often easier to prevent propagation than it is to introduce edge case handling into event handler code.

With a few minor trade offs, we now have a single button click handler function that works with 3 buttons, or with 300 buttons. It's the little things like this that make jQuery code scale well.

Note

We'll fully examine this use of `.on()`, in Chapter 10, *Advanced Events*.

Removing an event handler

There are times when we will be done with an event handler we previously registered. Perhaps the state of the page has changed such that the action no longer makes sense. It is possible to handle this situation with conditional statements inside our event handlers, but it is more elegant to unbind the handler entirely.

Suppose that we want our collapsible style switcher to remain expanded whenever the page is not using the normal style. While the Narrow Column or Large Print button is selected, clicking on the background of the style switcher should do nothing. We can accomplish this by calling the `.off()` method to remove the collapsing handler when one of the non-default style switcher buttons is clicked:

```
$(() => {
  $('#switcher')
    .click((e) => {
      if (!$(e.target).is('button')) {
        $(e.currentTarget)
          .children('button')
          .toggleClass('hidden');
      }
    });
  $('#switcher-narrow, #switcher-large')
    .click(() => {
      $('#switcher').off('click');
    });
});
```

Listing 3.18

Now when a button such as Narrow Column is clicked, the `click` handler on the style switcher `<div>` is removed, and clicking on the background of the box no longer collapses it. However, the buttons don't work anymore! They are affected by the `click` event of the style switcher `<div>` as well, because we rewrote the button-handling code to use event delegation. This means that when we call `$('#switcher').off('click')`, both behaviors are removed.

Giving namespaces to event handlers

We need to make our `.off()` call more specific so that it does not remove both of the click handlers we have registered. One way of doing this is to use **event namespacing**. We can introduce additional information when an event is bound that allows us to identify that particular handler later. To use namespaces, we need to return to the non-shorthand method of binding event handlers, the `.on()` method itself.

The first parameter we pass to `.on()` is the name of the event we want to watch for. We can use a special syntax here, though, that allows us to subcategorize the event

```
$(() => {
  $('#switcher')
    .on('click.collapse', (e) => {
      if (!$(e.target).is('button')) {
        $(e.currentTarget)
          .children('button')
          .toggleClass('hidden');
      }
    });
  $('#switcher-narrow, #switcher-large')
   .click(() => {
     $('#switcher').off('click.collapse');
   });
});
```

Listing 3.19

The `.collapse` suffix is invisible to the event handling system; `click` events are handled by this function, just as if we wrote `.on('click')`. However, the addition of the namespace means that we can unbind just this handler without affecting the separate `click` handler we wrote for the buttons.

Tip

There are other ways of making our `.off()` call more specific, as we will see in a moment. However, event namespacing is a useful tool in our arsenal. It is especially handy in the creation of plugins, as we'll see in later chapters.

Rebinding events

Now clicking on the Narrow Column or Large Print button causes the style switcher collapsing functionality to be disabled. However, we want the behavior to return when the Default button is pressed. To do this, we will need to **rebind** the handler whenever Default is clicked.

First, we should give our handler function a name so that we can use it more than once without repeating ourselves:

```
$(() => {
  const toggleSwitcher = (e) => {
    if (!$(e.target).is('button')) {
      $(e.currentTarget)
        .children('button')
        .toggleClass('hidden');
    }
  };

  $('#switcher')
    .on('click.collapse', toggleSwitcher);
  $('#switcher-narrow, #switcher-large')
    .click((e) => {
      $('#switcher').off('click.collapse');
    });
});
```

Listing 3.20

Recall that we are passing `.on()` a **function reference** as its second argument. It is important to remember when referring to a function that we must omit parentheses after the function name; parentheses would cause the function to be *called* rather than *referenced*.

Now that the `toggleSwitcher()` function can be referenced, we can bind it again later, without repeating the function definition:

```
$(() => {
  const toggleSwitcher = (e) => {
    if (!$(e.target).is('button')) {
      $(e.currentTarget)
        .children('button')
        .toggleClass('hidden');
    }
  };

  $('#switcher').on('click.collapse', toggleSwitcher);
  $('#switcher-narrow, #switcher-large')
    .click(() => {
      $('#switcher').off('click.collapse');
    });
  $('#switcher-default')
    .click(() => {
      $('#switcher').on('click.collapse', toggleSwitcher);
    });
});
```

Listing 3.21

Now the toggle behavior is bound when the document is loaded, unbound when Narrow Column or Large Print is clicked, and rebound when Default is clicked after that.

Since we have named the function, we no longer need to use namespacing. The `.off()` method can take a function as a second argument; in this case, it unbinds only that specific handler. However, we have run into another problem. Remember that when a handler is bound to an event in jQuery, previous handlers remain in effect. In this case, each time Default is clicked, another copy of the `toggleSwitcher` handler is bound to the style switcher. In other words, the function is called an extra time for each additional click until the user clicks Narrow or Large Print , which unbinds all of the `toggleSwitcher` handlers at once.

When an even number of `toggleSwitcher` handlers are bound, clicks on the style switcher (but not on a button) appear to have no effect. In fact, the `hidden` class is being toggled multiple times, ending up in the same state it was when it began. To remedy this problem, we can unbind the handler when a user clicks on *any* button, and rebind only after ensuring that the clicked button's ID is `switcher-default`:

```
$(() => {
  const toggleSwitcher = (e) => {
    if (!$(e.target).is('button')) {
      $(e.currentTarget)
        .children('button')
        .toggleClass('hidden');
    }
  };

  $('#switcher')
    .on('click', toggleSwitcher);
  $('#switcher button')
    .click((e) => {
      $('#switcher').off('click', toggleSwitcher);

      if (e.target.id == 'switcher-default') {
        $('#switcher').on('click', toggleSwitcher);
      }
    });
});
```

Listing 3.22

A shortcut is also available for the situation in which we want to unbind an event handler immediately after the first time it is triggered. This shortcut, called `.one()`, is used as follows:

```
$('#switcher').one('click', toggleSwitcher);
```

This would cause the toggle action to occur only once.

Simulating user interaction

At times, it is convenient to execute code that we have bound to an event, even if the event isn't triggered directly by user input. For example, suppose we wanted our style switcher to begin in its collapsed state. We could accomplish this by hiding buttons from within the stylesheet, or by adding our `hidden` class or calling the `.hide()` method from a `$(() => {})` handler. Another way would be to simulate a click on the style switcher so that the toggling mechanism we've already established is triggered.

The `.trigger()` method allows us to do just this:

```
$(() => {
  $('#switcher').trigger('click');
});
```

Listing 3.23

Now when the page loads, the switcher is collapsed just as if it had been clicked, as shown in the following screenshot:

If we were hiding content that we wanted people without JavaScript enabled to see, this would be a reasonable way to implement **graceful degradation**. Although, this is very uncommon these days.

The `.trigger()` method provides the same set of shortcut methods that `.on()` does. When these shortcuts are used with no arguments, the behavior is to trigger the action rather than bind it:

```
$(() => {
  $('#switcher').click();
});
```

Listing 3.24

Reacting to keyboard events

As another example, we can add keyboard shortcuts to our style switcher. When the user types the first letter of one of the display styles, we will have the page behave as if the corresponding button was clicked. To implement this feature, we will need to explore **keyboard events**, which behave a bit differently from **mouse events**.

There are two types of keyboard events: those that react to the keyboard directly (`keyup` and `keydown`) and those that react to text input (`keypress`). A single character entry event could correspond to several keys, for example, when the *Shift* key in combination with the *X* key creates the capital letter *X* . While the specifics of implementation differ from one browser to the next (unsurprisingly), a safe rule of thumb is: if you want to know what key the user pushed, you should observe the `keyup` or `keydown` event; if you want to know what character ended up on the screen as a result, you should observe the `keypress` event. For this feature, we just want to know when the user presses the *D*, *N*, or *L* key, so we will use `keyup`.

Next, we need to determine which element should watch for the event. This is a little less obvious than with mouse events, where we have a visible mouse cursor to tell us about the event's target. Instead, the target of a keyboard event is the element that currently has the **keyboard focus**. The element with focus can be changed in several ways, including using mouse clicks and pressing the *Tab* key. Not every element can get the focus, either; only items that have default keyboard-driven behaviors such as form fields, links, and elements with a `.tabIndex` property are candidates.

In this case, we don't really care what element has the focus; we want our switcher to work whenever the user presses one of the keys. Event bubbling will once again come in handy, as we can bind our `keyup` event to the `document` element and have assurance that eventually any key event will bubble up to us.

Finally, we will need to know which key was pressed when our `keyup` handler gets triggered. We can inspect the `event` object for this. The `.which` property of the event contains an identifier for the key that was pressed, and for alphabetic keys, this identifier is the ASCII value of the uppercase letter. With this information, we can now create an *object literal* of letters and their corresponding buttons to click. When the user presses a key, we'll see if its identifier is in the map, and if so, trigger the click:

```
$(() => {
  const triggers = {
    D: 'default',
    N: 'narrow',
    L: 'large'
  };

  $(document)
    .keyup((e) => {
      const key = String.fromCharCode(e.which);

      if (key in triggers) {
        $(`#switcher-${triggers[key]}`).click();
      }
    });
});
```

Listing 3.25

Presses of these three keys now simulate mouse clicks on the buttons--provided that the key event is not interrupted by features such as Firefox's search for text when I start typing .

As an alternative to using `.trigger()` to simulate this click, let's explore how to factor out code into a function so that more than one handler can call it--in this case, both `click` and `keyup` handlers. While not necessary in this case, this technique can be useful in eliminating code redundancy:

```
$(() => {
  // Enable hover effect on the style switcher
  const toggleHover = (e) => {
    $(e.target).toggleClass('hover');
  };

  $('#switcher').hover(toggleHover, toggleHover);

  // Allow the style switcher to expand and collapse.
  const toggleSwitcher = (e) => {
```

```
    if (!$(e.target).is('button')) {
      $(e.currentTarget)
        .children('button')
        .toggleClass('hidden');
    }
  };

  $('#switcher')
    .on('click', toggleSwitcher)
    // Simulate a click so we start in a collaped state.
    .click();

  // The setBodyClass() function changes the page style.
  // The style switcher state is also updated.
  const setBodyClass = (className) => {
    $('body')
      .removeClass()
      .addClass(className);

    $('#switcher button').removeClass('selected');
    $(`#switcher-${className}`).addClass('selected');
    $('#switcher').off('click', toggleSwitcher);

    if (className == 'default') {
      $('#switcher').on('click', toggleSwitcher);
    }
  };

  // Begin with the switcher-default button "selected"
  $('#switcher-default').addClass('selected');

  // Map key codes to their corresponding buttons to click
  const triggers = {
    D: 'default',
    N: 'narrow',
    L: 'large'
  };

  // Call setBodyClass() when a button is clicked.
  $('#switcher')
    .click((e) => {
      if ($(e.target).is('button')) {
        setBodyClass(e.target.id.split('-')[1]);
      }
    });

  // Call setBodyClass() when a key is pressed.
  $(document)
    .keyup((e) => {
      const key = String.fromCharCode(e.which);

      if (key in triggers) {
        setBodyClass(triggers[key]);
      }
    });
});
```

Listing 3.26

This final revision consolidates all the previous code examples of this chapter. We have moved the entire block of code into a single `$(() => {})` handler and made our code less redundant.

Summary

The abilities we've discussed in this chapter allow us to react to various user-driven and browser-initiated events. We have learned how to safely perform actions when page loads, how to handle mouse events such as clicking on links or hovering over buttons, and how to interpret keystrokes.

In addition, we have delved into some of the inner workings of the event system, and can use this knowledge to perform event delegation and to change the default behavior of an event. We can even simulate the effects of an event as if the user initiated it.

We can use these capabilities to build interactive pages. In the next chapter, we'll learn how to provide visual feedback to the user during these interactions.

Further reading

The topic of event handling will be explored in more detail in Chapter 10, *Advanced Events*. A complete list of jQuery's event methods is available in *Appendix C* of this book, or in the official jQuery documentation at http://api.jquery.com/.

Exercises

Challenge exercises may require the use of the official jQuery documentation at http://api.jquery.com/.

1. When Charles Dickens is clicked, apply the `selected` style to it.
2. When a chapter title (`<h3 class="chapter-title">`) is double-clicked, toggle the visibility of the chapter text.
3. When the user presses the right arrow key, cycle to the next `body` class. The key code for the right arrow key is `39`.
4. Challenge : Use the `console.log()` function to log the coordinates of the mouse as it moves across any paragraph. (Note: `console.log()` displays its results via the Firebug extension for Firefox, Safari's Web Inspector, or the Developer Tools in Chrome or Internet Explorer).
5. Challenge : Use `.mousedown()` and `.mouseup()` to track mouse events anywhere on the page. If the mouse button is released *above* where it was pressed, add the `hidden` class to all paragraphs. If it is released *below* where it was pressed, remove the `hidden` class from all paragraphs.