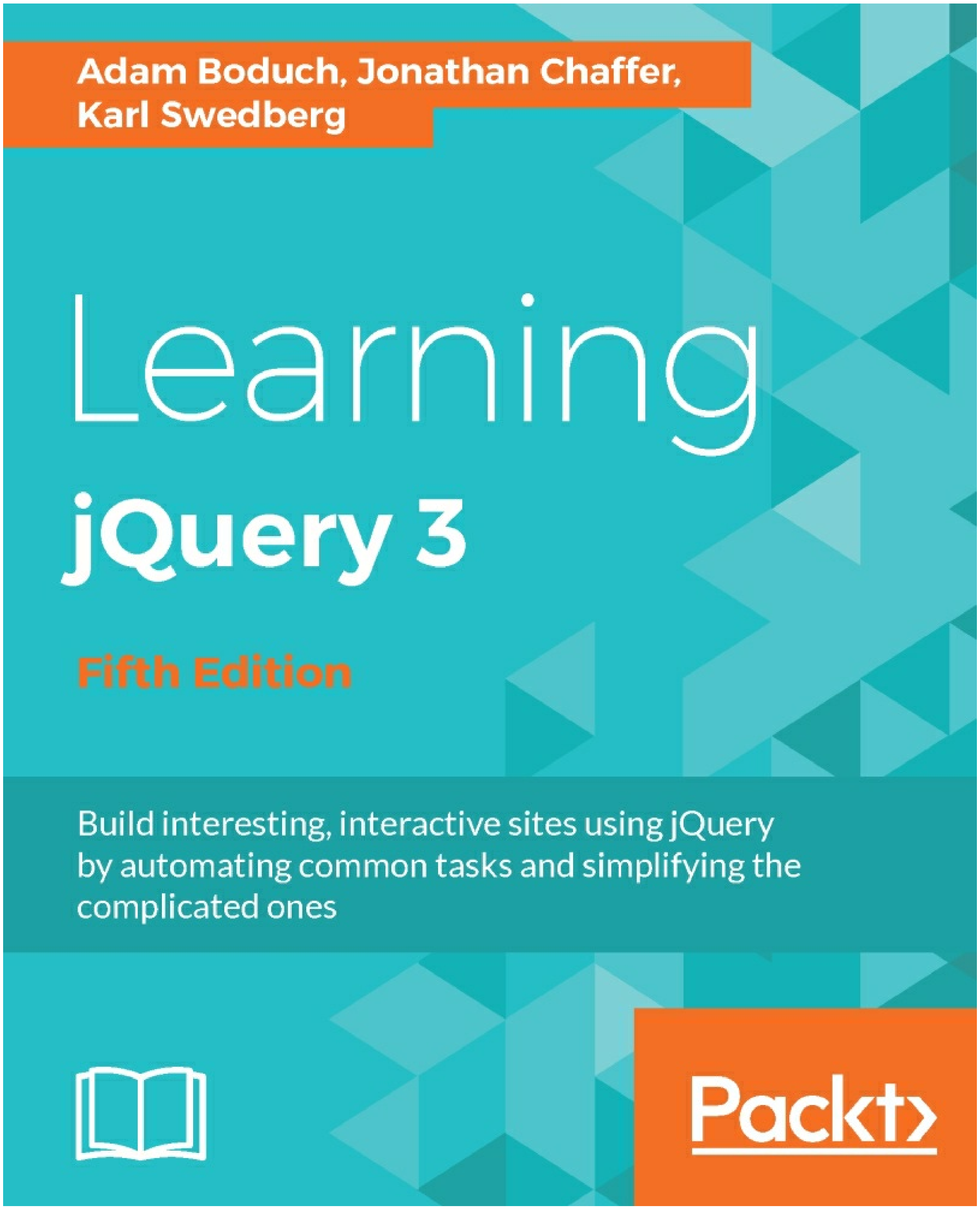

Adam Boduch, Jonathan Chaffer,
Karl Swedberg
Learning
jQuery 3
Fifth Edition
Build interesting, interactive sites using jQuery by automating common tasks and simplifying the complicated ones
Packt

Contents

Chapter 1. Getting Started

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.

Note

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">
        "—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;
}
```

Note

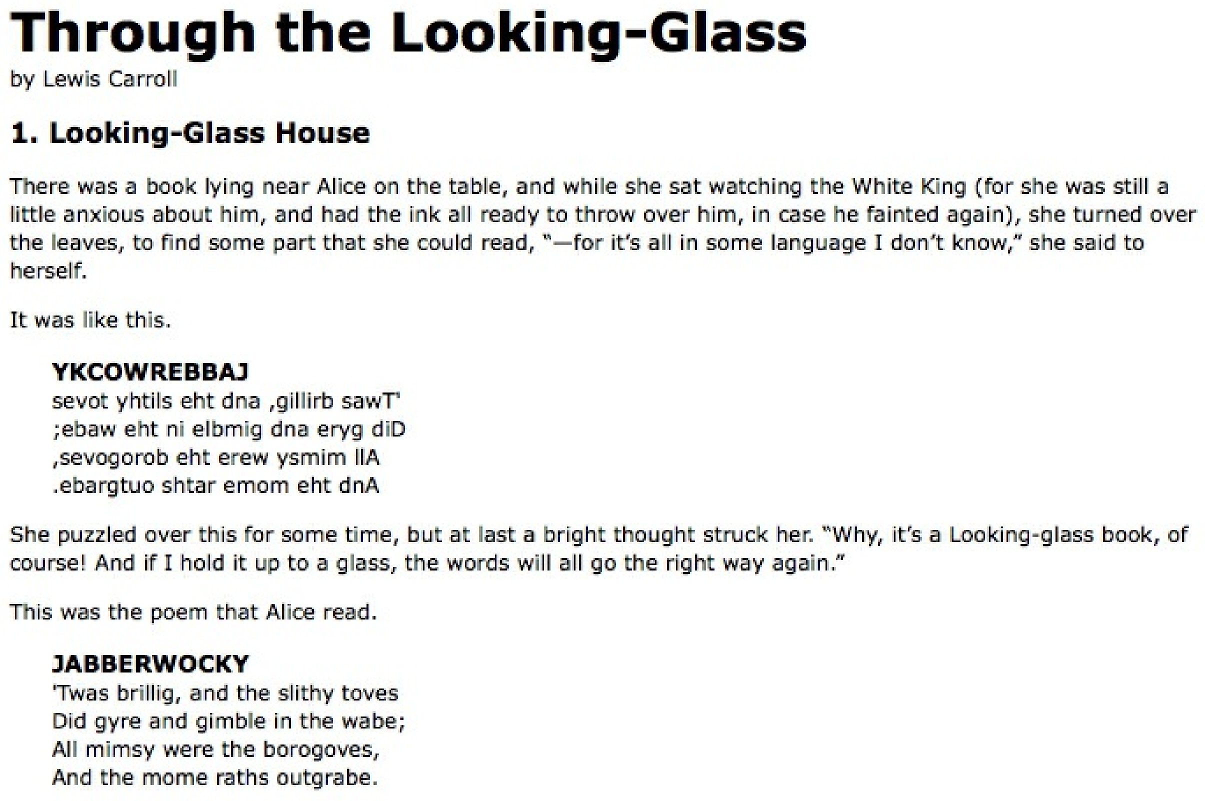
`Getting the example code`You can access the example code from the following GitHub repository: https://github.com/PacktPublishing/Learning-jQuery-3.

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.

Note

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:



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.

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:

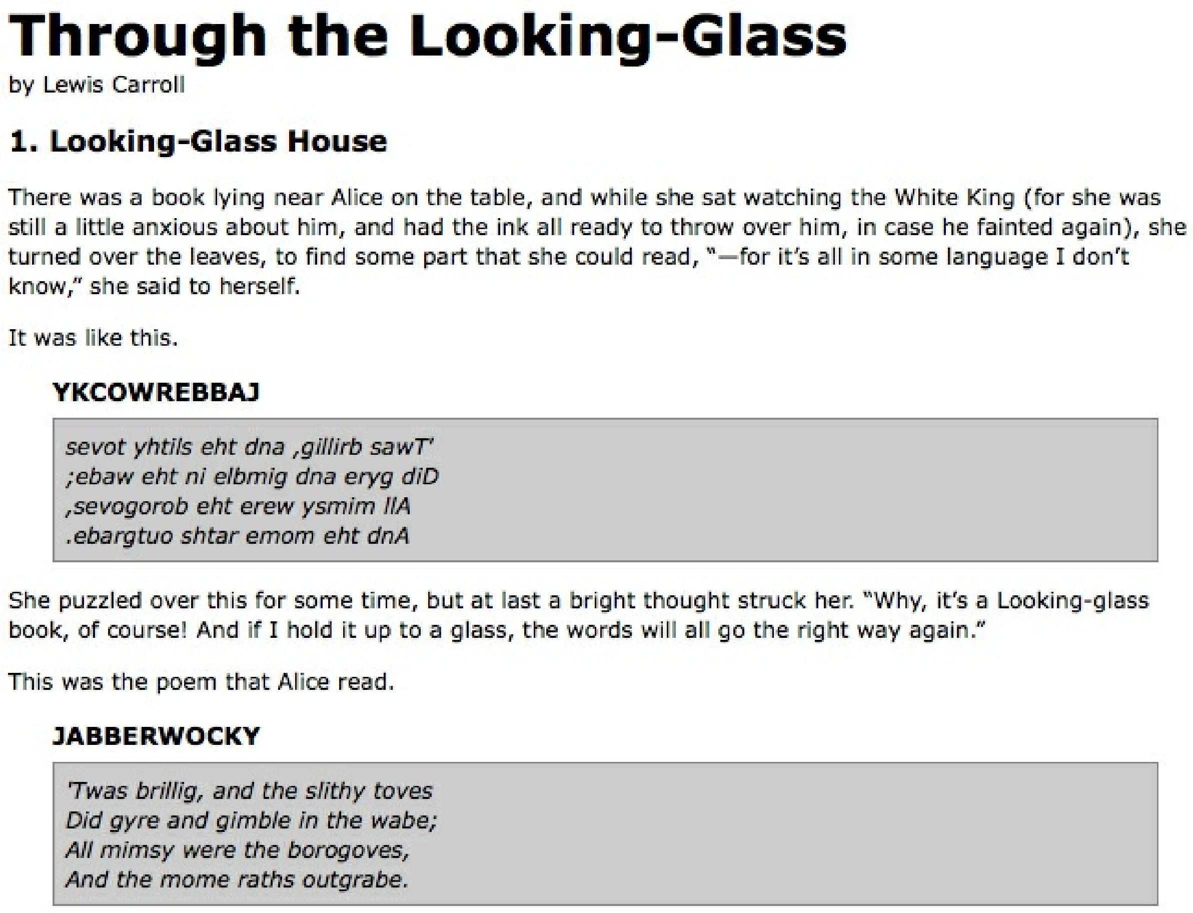
```
$(() =>
  $('div.poem-stanza').addClass('highlight')
);
```

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, 'highlig
      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.

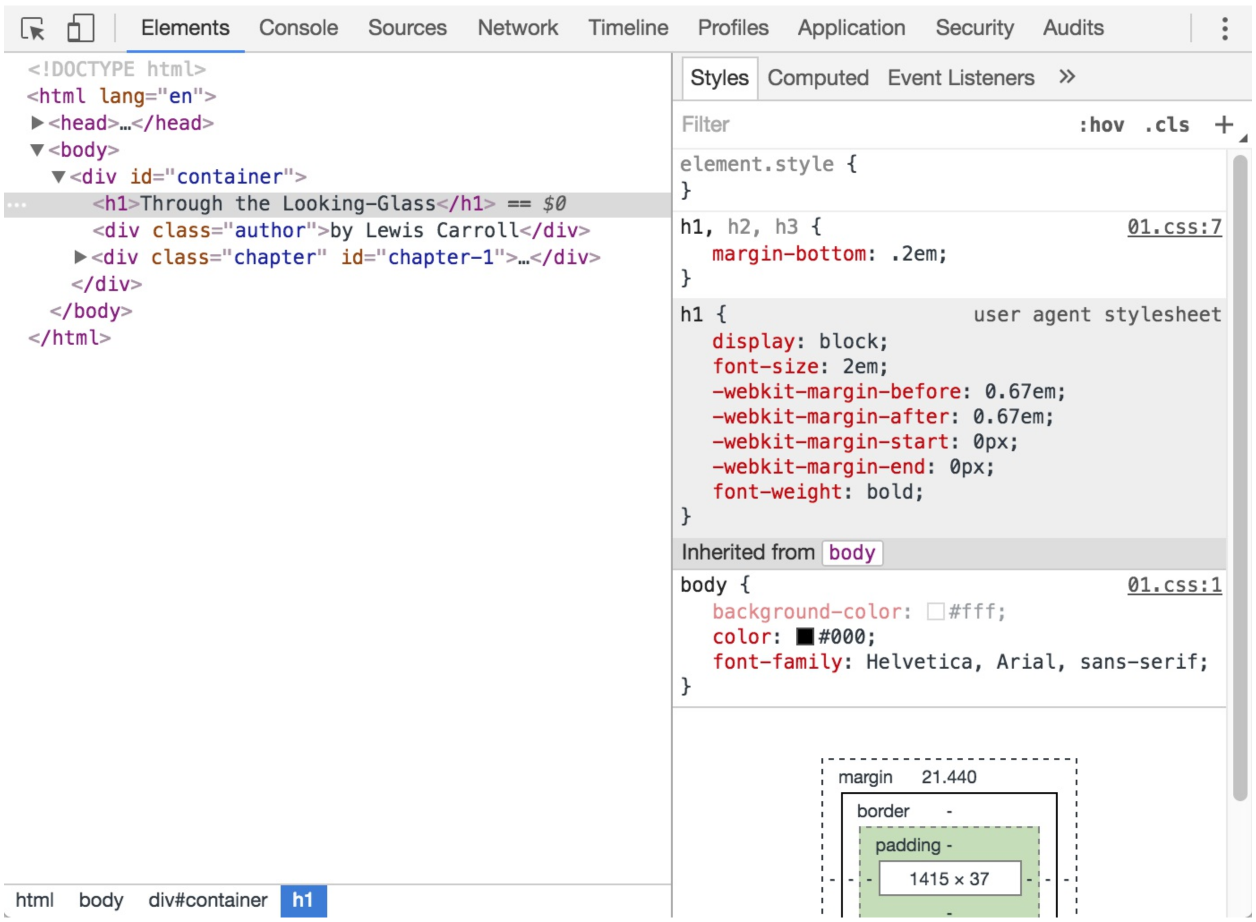
Chrome Developer Tools

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.

Note

`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:

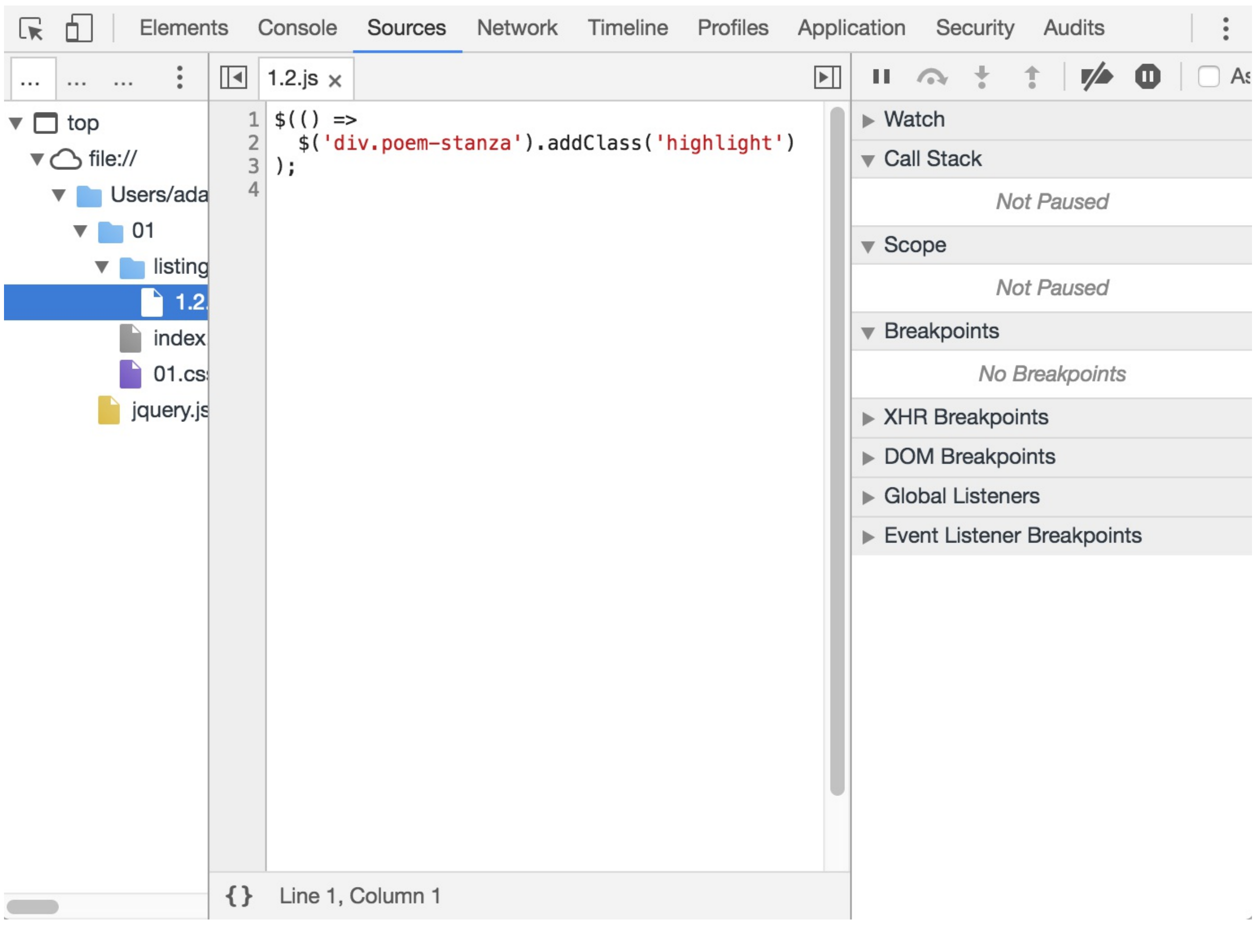


```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
   ▼<div id="container">
      <h1>Through the Looking-Glass</h1> == $0
      <div class="author">by Lewis Carroll</div>
     ▶<div class="chapter" id="chapter-1">…</div>
    </div>
  </body>
</html>
```

html body div#container h1

```
element.style {
}
h1, h2, h3 {                                01.css:7
   margin-bottom: .2em;
}
h1 {                          user agent stylesheet
   display: block;
   font-size: 2em;
   -webkit-margin-before: 0.67em;
   -webkit-margin-after: 0.67em;
   -webkit-margin-start: 0px;
   -webkit-margin-end: 0px;
   font-weight: bold;
}
Inherited from body
body {                                      01.css:1
   background-color: #fff;
   color: #000;
   font-family: Helvetica, Arial, sans-serif;
}
```

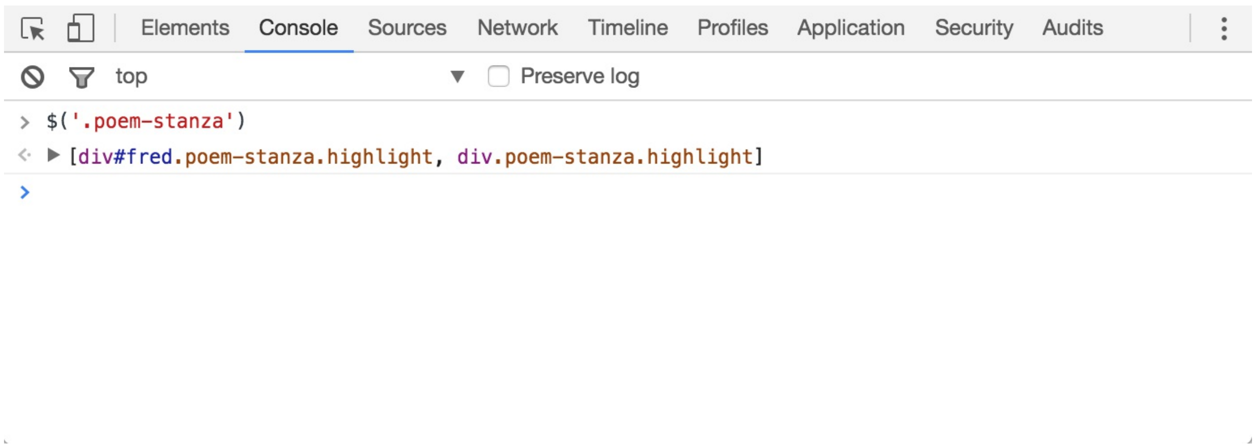
The **Sources**tab allows us to view the contents of all loaded scripts on the page. By right-clicking on a line number, we can set a breakpoint, set a conditional breakpoint, or have the script continue to that line after another breakpoint is reached. Breakpoints are effective ways to pause the execution 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:



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

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 of 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:



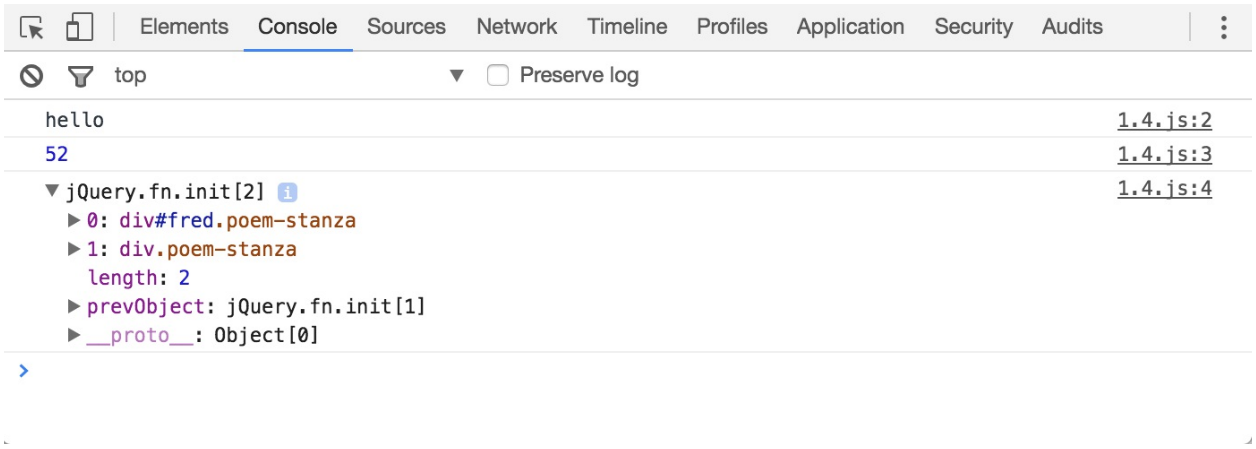
```
> $('.poem-stanza')
<· ▶ [div#fred.poem-stanza.highlight, div.poem-stanza.highlight]
>
```

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:



```
  hello                                                    1.4.js:2
  52                                                       1.4.js:3
  ▼jQuery.fn.init[2]                                       1.4.js:4
    ▶0: div#fred.poem-stanza
    ▶1: div.poem-stanza
     length: 2
    ▶prevObject: jQuery.fn.init[1]
    ▶__proto__: Object[0]
>
```

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.

Chapter 2. Selecting Elements

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 ,

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.

Note

`Making jQuery play well with other...`

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.

Note

`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:

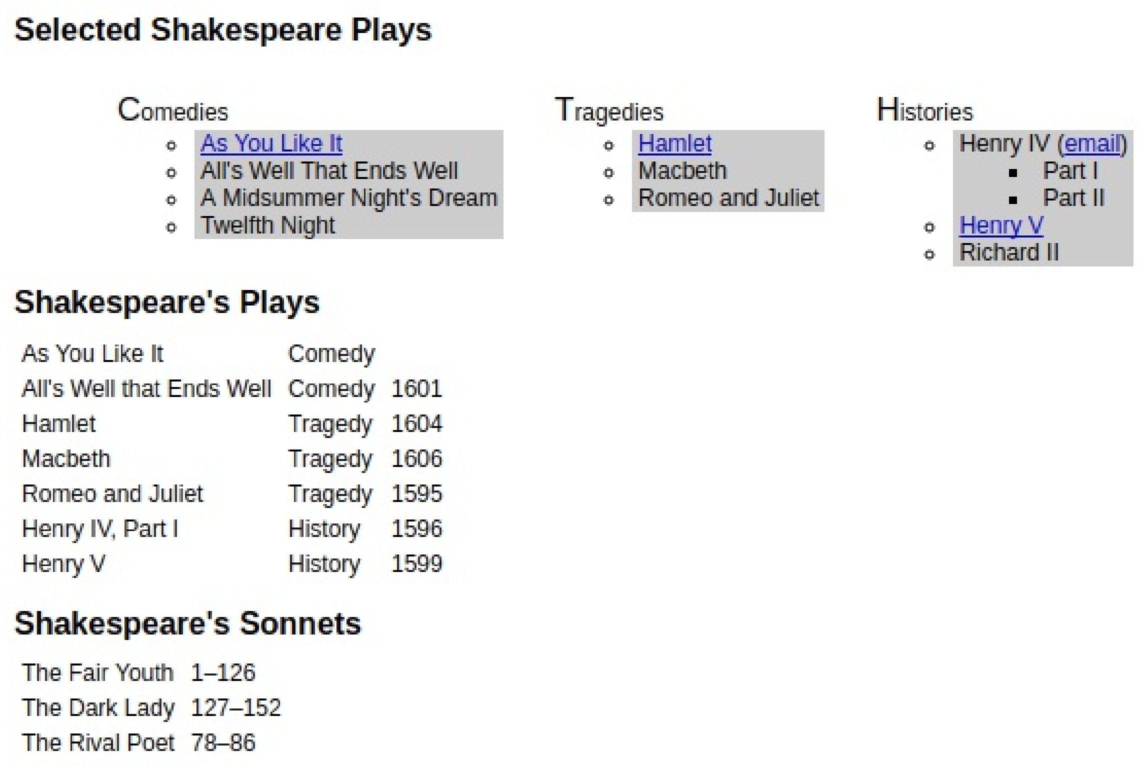
Selector specificity

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:



Selected Shakespeare Plays

Comedies
- As You Like It
- All's Well That Ends Well
- A Midsummer Night's Dream
- Twelfth Night

Tragedies
- Hamlet
- Macbeth
- Romeo and Juliet

Histories
- Henry IV (email)
  - Part I
  - Part II
- Henry V
- Richard II

Shakespeare's Plays

As You Like It	Comedy	
All's Well that Ends Well	Comedy	1601
Hamlet	Tragedy	1604
Macbeth	Tragedy	1606
Romeo and Juliet	Tragedy	1595
Henry IV, Part I	History	1596
Henry V	History	1599

Shakespeare's Sonnets

The Fair Youth	1–126
The Dark Lady	127–152
The Rival Poet	78–86

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...
```

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...

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.

Note

`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...

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:

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", "Histo
```

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...

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...

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.

Chapter 3. Handling Events

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....

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.

Note

`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...

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...

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...

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

Simulating user interaction

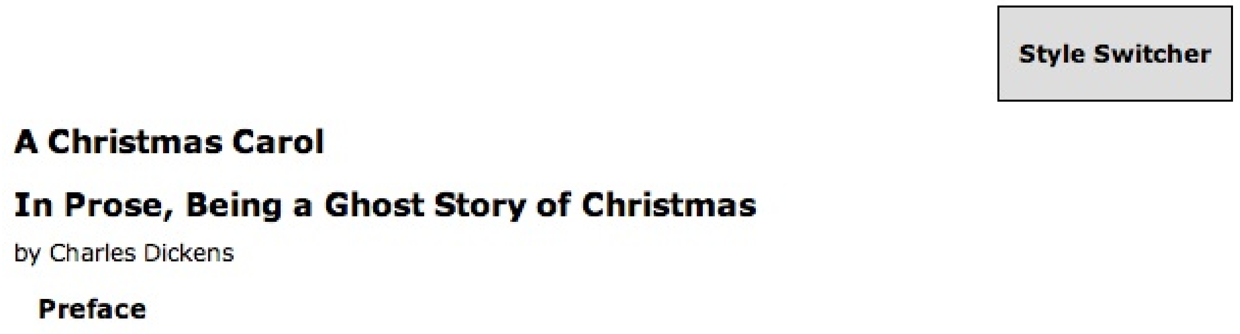
At times, it is convenient to execute code that we have bound to an event, even if the event isn't triggered directly by user input. For example, suppose we wanted our style switcher to begin in its collapsed state. We could accomplish this by hiding buttons from within the stylesheet, or by adding our `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:

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 the 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.

Chapter 4. Styling and Animating

If actions speak louder than words, then in the JavaScript world, effects make actions speak louder still. With jQuery, we can easily add impact to our actions through a set of simple visual effects and even craft our own sophisticated animations.

The effects offered by jQuery supply simple visual flourishes that grant a sense of movement and modernity to any page. However, apart from being mere decoration, they can also provide important usability enhancements that help orient the user when something happens on a page (especially common in Ajax applications).

In this chapter, we will cover:

- Changing the styling of elements on the fly
- Hiding and showing elements with various built-in effects
- Creating custom animations of elements
- Sequencing effects to happen one after another

Modifying CSS with inline properties

Before we jump into jQuery effects, a quick look at CSS is in order. In previous chapters, we have been modifying a document's appearance by defining styles for classes in a separate stylesheet and then adding or removing those classes with jQuery. Typically, this is the preferred process for injecting CSS into HTML because it respects the stylesheet's role in dealing with the presentation of a page. However, there may be times when we need to apply styles that haven't been or can't easily be defined in a stylesheet. Fortunately, jQuery offers the `.css()` method for such occasions.

This method acts as both a **getter** and a **setter**. To get the value of a single style property, we simply pass the name of the property as a string and get a string in return. To get the value of multiple style properties, we can pass the property names as an array of strings to get an object of property-value pairs in return. Multiword property names, such as `backgroundColor` can be interpreted by jQuery when in hyphenated CSS notation (`background-color`) or camel-cased DOM notation (`backgroundColor`):

```
// Get a single...
```

Hiding and showing elements

The basic `.hide()` and `.show()` methods, without any parameters, can be thought of as smart shorthand methods for `.css('display', 'string')`, where `'string'` is the appropriate display value. The effect, as might be expected, is that the matched set of elements will be immediately hidden or shown with no animation.

The `.hide()` method sets the inline style attribute of the matched set of elements to `display: none`. The smart part here is that it remembers the value of the display property--typically `block`, `inline`, or `inline-block`--before it was changed to `none`. Conversely, the `.show()` method restores the display properties of the matched set of elements to whatever they initially were before `display: none` was applied.

Note

`The display property`For more information about the `display` property and how its values are visually represented in a web page, visit the Mozilla Developer Center at https://developer.mozilla.org/en-US/docs/CSS/display and view examples at https://developer.mozilla.org/samples/cssref/display.html.

This feature of `.show()` and `.hide()` is especially helpful when hiding elements that have had their...

Effects and duration

When we include a duration (sometimes also referred to as a speed) with `.show()` or `.hide()`, it becomes animated--occurring over a specified period of time. The `.hide(duration)` method, for example, decreases an element's height, width, and opacity simultaneously until all three reach zero, at which point the CSS rule `display: none` is applied. The `.show(duration)` method will increase the element's height from top to bottom, width from the left-hand side to the right-hand side, and opacity from 0 to 1 until its contents are completely visible.

Speeding in

With any jQuery effect, we can use one of the two preset speeds, `'slow'` or `'fast'`. Using `.show('slow')` makes the show effect complete in 600 milliseconds (0.6 seconds), `.show('fast')` in 200 milliseconds. If any other string is supplied, jQuery's default animation duration of 400 milliseconds will be used. For even greater precision, we can specify a number of milliseconds, for example, `.show(850)`.

Let's include a speed in our example when showing the second paragraph of **`Abraham Lincoln's Gettysburg Address`**:

```
$(() => {
  $('p')
    .eq(1)
    .hide();

 ...
```

Creating custom animations

In addition to the prebuilt effect methods, jQuery provides a powerful `.animate()` method that allows us to create our own custom animations with fine-grained control. The `.animate()` method comes in two forms. The first takes up to four arguments:

- An object of style properties and values, which is similar to the `.css()` argument discussed earlier in this chapter
- An optional duration, which can be one of the preset strings or a number of milliseconds
- An optional easing type, which is an option that we will not use now, but which we will discuss in Chapter 11, *Advanced Effects*
- An optional callback function, which will be discussed later in this chapter

All together, the four arguments look like this:

```
.animate(
  { property1: 'value1', property2: 'value2'},
  duration,
  easing,
  () => {
    console.log('The animation is finished.');
  }
);
```

The second form takes two arguments: an object of properties and an object of options:

```
.animate({properties}, {options})
```

In this form, the second argument wraps up the second through fourth arguments of the first form into another object and adds some more advanced options...

Simultaneous versus queued effects

The `.animate()` method, as we've just discovered, is very useful for creating **simultaneous** effects affecting a particular set of elements. There may be times, however, when we want to **queue** our effects to have them occur one after the other.

Working with a single set of elements

When applying multiple effects to the same set of elements, queuing is easily achieved by chaining those effects. To demonstrate this queuing, we'll revisit *Listing 4.17* by moving the **`Text Size`** box to the right-hand side, increasing its height and border width. This time, however, we perform the three effects sequentially simply by placing each in its own `.animate()` method and chaining the three together:

```
$(() => {
  $('div.label')
    .click((e) => {
      const $switcher = $(e.target).parent();
      const paraWidth = $('div.speech p').outerWidth();
      const switcherWidth = $switcher.outerWidth();

      $switcher
        .css('position', 'relative')
        .animate({ borderWidth: '5px' }, 'slow')
        .animate({ left: paraWidth - switcherWidth }, 'slow')
        .animate({ height: '+=20px' }, 'slow');
    });
});...
```

Summary

Using the effect methods that we have explored in this chapter, we should now be able to modify inline style attributes from JavaScript, apply prepackaged jQuery effects to elements, and create our own custom animations. In particular, you learned how to incrementally increase and decrease text size using either the `.css()` or `.animate()` methods, gradually hide and show page elements by modifying several attributes, and how to animate elements (simultaneously or sequentially) in a number of ways.

In the first four chapters of this book, all of our examples have involved manipulating elements that have been hardcoded into the page's HTML. In Chapter 5, *Manipulating the DOM*, we will explore ways to manipulate the DOM directly, including using jQuery to create new elements and insert them into the DOM wherever we choose.

Further reading

The topic of animation will be explored in more detail in Chapter 11, *Advanced Effects*. A complete list of effect and styling methods is available in Appendix B of this book, or in the official jQuery documentation at http://api.jquery.com/.

Exercises

The challenge exercise may require the use of the official jQuery documentation at http://api.jquery.com/:

1. Alter the stylesheet to hide the contents of the page initially. When the page is loaded, fade in the contents slowly.
2. Give each paragraph a yellow background only when the mouse is over it.
3. Make a click on the title `(<h2>)` and simultaneously fade it to 25 percent opacity and grow its left-hand margin to `20px`. Then, when this animation is complete, fade the speech text to 50 percent opacity.
4. Here's a challenge for you. React to presses of the arrow keys by smoothly moving the switcher box 20 pixels in the corresponding direction. The key codes for the arrow keys are: `37` (left), `38` (up), `39` (right), and `40` (down).

Chapter 5. Manipulating the DOM

The Web experience is a partnership between web servers and web browsers. Traditionally, it has been the domain of the server to produce an HTML document that is ready for consumption by the browser. The techniques we have seen in this book have shifted this arrangement slightly, using CSS techniques to alter the appearance of the HTML document on the fly. To really flex our JavaScript muscles, though, you'll need to learn to alter the document itself.

In this chapter, we will cover:

- Modifying the document using the interface provided by the **Document Object Model** (**DOM**)
- Creating elements and text on a page
- Moving or deleting elements
- Transforming a document by adding, removing, or modifying attributes and properties

Manipulating attributes and properties

Throughout the first four chapters of this book, we have been using the `.addClass()` and `.removeClass()` methods to demonstrate how we can change the appearance of elements on a page. Although we discussed these methods informally in terms of manipulating the `class` attribute, jQuery actually modifies a DOM property called `className`. The `.addClass()` method creates or adds to the property, while `.removeClass()` deletes or shortens it. Add to these the `.toggleClass()` method, which alternates between adding and removing class names, and we have an efficient and robust way of handling classes. These methods are particularly helpful in that they avoid adding a class if it already exists on an element (so we don't end up with `<div class="first first">`, for example), and correctly handle cases where multiple classes are applied to a single element, such as `<div class="first second">`.

Non-class attributes

We may need to access or change several other attributes or properties from time to time. For manipulating attributes such as `id`, `rel`, and `href`, jQuery provides the `.attr()` and `.removeAttr()` methods....

DOM tree manipulation

The `.attr()` and `.prop()` methods are very powerful tools, and with them we can make targeted changes to the document. We still haven't seen ways to change the overall structure of the document though. To actually manipulate the DOM tree, you'll need to learn a bit more about the function that lies at the very heart of the `jQuery` library.

The $() function revisited

From the start of this book, we've been using the `$()` function to access elements in a document. As we've seen, this function acts as a factory, producing new jQuery objects that point to the elements described by CSS selectors.

This isn't all that the `$()` function can do. It can also change the contents of a page. Simply by passing a snippet of HTML code to the function, we can create an entirely new DOM structure.

Note

`Accessibility reminder`We should keep in mind, once again, the inherent danger in making certain functionality, visual appeal, or textual information available only to those with web browsers capable of (and enabled for) using JavaScript. Important information should be accessible to all, not just people who happen to be using the right...

Copying elements

So far in this chapter, we have inserted newly created elements, moved elements from one location in the document to another, and wrapped new elements around existing ones. Sometimes, though, we may want to copy elements. For example, a navigation menu that appears in the page's header could be copied and placed in the footer as well. Whenever elements can be copied to enhance a page visually, we can let jQuery do the heavy lifting.

For copying elements, jQuery's `.clone()` method is just what we need; it takes any set of matched elements and creates a copy of them for later use. As in the case of the `$()` function's element creation process we explored earlier in this chapter, the copied elements will not appear in the document until we apply one of the insertion methods.

For example, the following line creates a copy of the first paragraph inside `<div class="chapter">`:

```
$('div.chapter p:eq(0)').clone();
```

This alone is not enough to change the content of the page. We can make the cloned paragraph appear before `<div class="chapter">` with an insertion method:

```
$('div.chapter p:eq(0)')
  .clone()
 ...
```

Content getter and setter methods

It would be nice to be able to modify the pull quote a bit by dropping some words and replacing them with ellipses to keep the content brief. To demonstrate this, we have wrapped a few words of the example text in a `<span class="drop">` tag.

The easiest way to accomplish this replacement is to directly specify the new HTML entity that is to replace the old one. The `.html()` method is perfect for this:

```
$(() => {
  $('span.pull-quote')
    .each((i, span) => {
      $(span)
        .clone()
        .addClass('pulled')
        .find('span.drop')
          .html('&hellip;')
          .end()
        .prependTo(
          $(span)
            .parent()
            .css('position', 'relative')
        );
    });
});
```

Listing 5.20

The new lines in *Listing 5.20* rely on the DOM traversal techniques we learned in Chapter 2, *Selecting Elements*. We use `.find()` to search inside the pull quote for any `<span class="drop">` elements, operate on them, and then return to the pull quote itself by calling `.end()`. In between these methods, we invoke `.html()` to change the content into an ellipsis (using the appropriate HTML...

DOM manipulation methods in a nutshell

The extensive DOM manipulation methods that jQuery provides vary according to their task and their target location. We haven't covered them all here, but most are analogous to the ones we've seen, and more will be discussed in Chapter 12, *Advanced DOM Manipulation*. The following outline can serve as a reminder of which method we can use to accomplish which task:

- To *create* new elements from HTML, use the `$()` function
- To *insert* new elements *inside* every matched element, use the following functions:
    - `.append()`
    - `.appendTo()`
    - `.prepend()`
    - `.prependTo()`
- To *insert* new elements *adjacent to* every matched element, use the following functions:
    - `.after()`
    - `.insertAfter()`
    - `.before()`
    - `.insertBefore()`
- To *insert* new elements *around* every matched element, use the following functions:
    - `.wrap()`
    - `.wrapAll()`
    - `.wrapInner()`
- To *replace* every matched element with new elements or text, use the following functions:
    - `.html()`
    - `.text()`
    - `.replaceAll()`
    - `.replaceWith()`
- To *remove* elements inside every matched element, use the following

function:
    - `.empty()`
- To *remove* every matched element and descendants from the document without actually deleting them, use the...

Summary

In this chapter, we have created, copied, reassembled, and embellished content using jQuery's DOM modification methods. We've applied these methods to a single web page, transforming a handful of generic paragraphs to a footnoted, pull-quoted, linked, and stylized literary excerpt. This chapter has shown us just how easy it is to add, remove, and rearrange the contents of a page with jQuery. In addition, you have learned how to make any changes we want to the CSS and DOM properties of page elements.

Next up, we'll take a round-trip journey to the server via jQuery's Ajax methods.

Further reading

The topic of DOM manipulation will be explored in more detail in Chapter 12, *Advanced DOM Manipulation*. A complete list of DOM manipulation methods is available in Appendix B, *Quick Reference*, of this book, or in the official jQuery documentation at http://api.jquery.com/.

Exercises

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

1. Alter the code that introduces the **`back to top`** links, so that the links only appear after the fourth paragraph.
2. When a **`back to top`** link is clicked on, add a new paragraph after the link containing the message **`You were here`**. Ensure that the link still works.
3. When the author's name is clicked, turn it bold (by adding an element, rather than manipulating classes or CSS attributes).
4. **`Challenge`**: On a subsequent click of the bolded author's name, remove the `<b>` element that was added (thereby toggling between bold and normal text).
5. **`Challenge`**: Add a class of `inhabitants` to each of the chapter's paragraphs without calling `.addClass()`. Make sure to preserve any existing classes.

Chapter 6. Sending Data with Ajax

The term **Asynchronous JavaScript and XML** (**Ajax**) was coined by *Jesse James Garrett* in 2005. Since then, it has come to represent many different things, as the term encompasses a group of related capabilities and techniques. At its most basic level, an Ajax solution includes the following technologies:

- **JavaScript**: This is used to capture interactions with the user or other browser-related events and to interpret the data from the server and present it on the page
- **XMLHttpRequest**: This allows requests to be made to the server without interrupting other browser tasks
- **Textual data:** The server provides data in a format such as XML, HTML, or JSON

Ajax transforms static **web pages** into interactive **web applications**. Unsurprisingly, browsers are not entirely consistent with their implementations of the `XMLHttpRequest` object, but jQuery will assist us.

In this chapter, we will cover:

- Loading data from the server without a page refresh
- Sending data from JavaScript in the browser back to the server
- Interpreting data in a variety of formats, including HTML, XML, and JSON
- Providing feedback to the user about the status...

Loading data on demand

Ajax is just a means of loading data from the server into the web browser without a page refresh. This data can take many forms, and we have many options for what to do with it when it arrives. We'll see this by performing the same basic task, using different approaches.

We are going to build a page that displays entries from a dictionary, grouped by the starting letter of the dictionary entry. The HTML defining the content area of the page will look like this:

```
<div id="dictionary">
</div>
```

Our page will have no content to begin with. We are going to use jQuery's various Ajax methods to populate this `<div>` tag with dictionary entries.

Note

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

We're going to need a way to trigger the loading process, so we'll add some links for our event handlers to latch onto:

```
<div class="letters">
  <div class="letter" id="letter-a">
    <h3><a href="entries-a.html">A</a></h3>
  </div>
  <div class="letter" id="letter-b">
    <h3><a href="entries-a.html">B</a></h3>
  </div>
 ...
```

Choosing a data format

We have looked at four formats for our external data, each of which is handled by jQuery's Ajax functions. We have also verified that all four can handle the task at hand, loading information onto an existing page when the user requests it and not before. How, then, do we decide which one to use in our applications?

HTML snippets require very little work to implement. The external data can be loaded and inserted into the page with one simple method that doesn't even require a callback function. No traversal of the data is necessary for the straightforward task of adding the new HTML into the existing page. On the other hand, the data is not necessarily structured in a way that makes it reusable for other applications. The external file is tightly coupled with its intended container.

JSON files are structured for simple reuse. They are compact and easy to read. The data structure must be traversed to pull out the information and present it on the page, but this can be done with standard JavaScript techniques. Since modern browsers parse the files natively with a single call to `JSON.parse()`, reading in a JSON...

Passing data to the server

Our examples to this point have focused on the task of retrieving static data files from the web server. However, the server can dynamically shape the data based on input from the browser. We're helped along by jQuery in this task as well; all of the methods we've covered so far can be modified so that data transfer becomes a two-way street.

Note

`Interacting with server-side code`Since demonstrating these techniques requires interaction with the web server, we'll need to use server-side code for the first time here. The examples given will use Node.js, which is very widely used as well as freely available. We will not cover any Node.js or Express specifics here, but there are plentiful resource on the web if you Google either of these technologies.

Performing a GET request

To illustrate the communication between client (using JavaScript) and server (also using JavaScript), we'll write a script that only sends one dictionary entry to the browser on each request. The entry chosen will depend on a parameter sent from the browser. Our script will pull its data from an internal data structure like this:

```
const...
```

Keeping an eye on the request

So far, it has been sufficient for us to make a call to an Ajax method and patiently await the response. At times, though, it is handy to know a bit more about the HTTP request as it progresses. If such a need arises, jQuery offers a suite of functions that can be used to register callbacks when various Ajax-related events occur.

The `.ajaxStart()` and `.ajaxStop()` methods are two examples of these observer functions. When an Ajax call begins with no other transfer in progress, the `.ajaxStart()` callback is fired. Conversely, when the last active request ends, the callback attached with `.ajaxStop()` will be executed. All of the observers are global, in that they are called when any Ajax communication occurs, regardless of what code initiates it. And all of them, can only be bound to `$(document)`.

We can use these methods to provide some feedback to the user in the case of a slow network connection. The HTML for the page can have a suitable loading message appended:

```
<div id="loading">
  Loading...
</div>
```

This message is just a piece of arbitrary HTML; it could include an animated GIF image as a loading...

Error handling

So far, we have only dealt with successful responses to Ajax requests, loading the page with new content when everything goes as planned. Responsible developers, however, should account for the possibility of network or data errors and report them appropriately. Developing Ajax applications in a local environment can lull developers into a sense of complacency since, aside from a possible mistyped URL, Ajax errors don't just happen locally. The Ajax convenience methods such as `$.get()` and `.load()` do not provide an error callback argument themselves, so we need to look elsewhere for a solution to this problem.

Aside from using the `global .ajaxError()` method, we can react to errors by capitalizing on jQuery's deferred object system. We will discuss deferred objects more fully in Chapter 11, *Advanced Effects*, but, for now, we'll simply note that we can chain `.done()`, `.always()`, and `.fail()` methods to any Ajax function except `.load()`, and use these methods to attach the relevant callbacks. For example, if we take the code from *Listing 6.16* and change the URL to one that doesn't exist, we can test the `.fail()` method:

```
$(()...
```

Ajax and events

Suppose we wanted to allow each dictionary term name to control the display of the definition that follows; clicking on the term name would show or hide the associated definition. With the techniques we have seen so far, this should be pretty straightforward:

```
$(() => {
  $('h3.term')
    .click((e) => {
      $(e.target)
        .siblings('.definition')
        .slideToggle();
    });
});
```

Listing 6.16

When a term is clicked on, this code finds siblings of the element that have a class of `definition`, and slides them up or down as appropriate.

All seems in order, but a click does nothing with this code. Unfortunately, the terms have not yet been added to the document when we attach the `click` handlers. Even if we managed to attach `click` handlers to these items, once we clicked on a different letter the handlers would no longer be attached.

This is a common problem with areas of a page populated by Ajax. A popular solution is to rebind handlers each time the page area is refreshed. This can be cumbersome, however, as the event-binding code needs to be called each time anything causes the DOM structure of the page to...

Deferreds and promises

jQuery deferred objects were introduced at a time when there was no consistent way to handle asynchronous behavior in JavaScript code. Promises help us orchestrate asynchronous stuff, such as multiple HTTP requests, file reads, animations, and so on. Promises aren't exclusive to JavaScript, nor are they a new idea. The best way to think about a promise is as a contract that promises to resolve a value *eventually*.

Now that promises are officially part of JavaScript, jQuery now fully supports promises. That is, jQuery deferred objects behave just like any other promise. This is important, as we'll see in this section, because it means that we can use jQuery deferreds to compose complex asynchronous behavior with other code that return native promises.

Performing Ajax calls on page load

Right now, our dictionary doesn't show any definitions on the initial page load. Instead, it just shows some empty space. Let's change that by showing the "A" entries when the document is ready. How do we do this?

One approach is to simply add the `load('a.html')` call into our document ready handler (`$(() => {})`) along with...

Summary

You have learned that the Ajax methods provided by jQuery can help us to load data in several different formats from the server without a page refresh. We can execute scripts from the server on demand and send data back to the server.

You've also learned how to deal with common challenges of asynchronous loading techniques, such as keeping handlers bound after a load has occurred and loading data from a third-party server.

This concludes our tour of the basic components of the `jQuery` library. Next, we'll look at how these features can be expanded upon easily using jQuery plugins.

Further reading

The topic of Ajax will be explored in more detail in Chapter 13, *Advanced Ajax*. A complete list of Ajax methods is available in Appendix B, *Quick Reference*, of this book or in the official jQuery documentation at http://api.jquery.com/.

Exercises

The challenge exercise may require the use of the official jQuery documentation athttp://api.jquery.com/:

1. When the page loads, pull the body content of `exercises-content.html` into the content area of the page.
2. Rather than displaying the whole document at once, create tooltips for the letters in the left-hand column by loading just the appropriate letter's content from `exercises-content.html` when the user's mouse is over the letter.
3. Add error handling for this page load, displaying the error message in the content area. Test this error handling code by changing the script to request `does-not-exist.html` rather than `exercises-content.html`.
4. Here's a challenge. When the page loads, send a JSONP request to GitHub and retrieve a list of repositories for a user. Insert the name and URL of each repository into the content area of the page. The URL to retrieve the jQuery project's repositories is https://api.github.com/users/jquery/repos.

Chapter 7. Using Plugins

Throughout the first six chapters of this book, we examined jQuery's core components. Doing this has illustrated many of the ways in which the jQuery library can be used to accomplish a wide variety of tasks. Yet as powerful as the library is at its core, its elegant **plugin architecture** has allowed developers to extend jQuery, making it even more feature rich.

The jQuery community created hundreds of plugins--from small selector helpers to full-scale user-interface widgets. You will now learn how to tap into this vast resource.

In this chapter, we will cover:

- Downloading and setting up plugins
- Calling jQuery methods provided by plugins
- Finding elements using custom selectors defined by jQuery plugins
- Adding sophisticated user interface behaviors using jQuery UI
- Implementing mobile-friendly features using jQuery Mobile

Using a plugin

Using a jQuery plugin is very straightforward. We need to simply obtain the plugin code, reference the plugin from our HTML, and invoke the new capabilities from our own scripts.

We can easily demonstrate these tasks using the jQuery **Cycle** plugin. This plugin, by Mike Alsup, allows us to quickly transform a static set of page elements into an interactive slideshow. Like many popular plugins, it can handle complex, advanced needs well, but can also hide this complexity when our requirements are more straightforward.

Downloading and referencing the Cycle plugin

To install any jQuery plugins, we'll use the `npm` package manager. This is the de facto tool for declaring package dependencies for modern JavaScript projects. For example, we can use a `package.json` file to declare that we need jQuery, and a specific set of jQuery plugins.

Note

For help on installing `npm`, see https://docs.npmjs.com/getting-started/what-is-npm. For help on initializing a `package.json` file, see https://docs.npmjs.com/getting-started/using-a-package.json.

Once you have a `package.json` file in the root of your project directory, you're ready to start...

Other types of plugins

Plugins need not be limited to providing additional jQuery methods. They can extend the library in many ways and even alter the functionality of existing features.

Plugins can change the way other parts of the jQuery library operate. Some offer new animation easing styles, for instance, or trigger additional jQuery events in response to user actions. The Cycle plugin offers such an enhancement by adding a new custom selector.

Custom selectors

Plugins that add custom selector expressions increase the capabilities of jQuery's built-in selector engine so that we can find elements on the page in new ways. Cycle adds a custom selector of this kind, which gives us an opportunity to explore this capability.

Cycle's slideshows can be paused and resumed by calling `.cycle('pause')` and `.cycle('resume')`, respectively. We can easily add buttons that control the slideshow, as shown in the following code:

```
$(() => {
  const $books = $('#books').cycle({
    timeout: 2000,
    speed: 200,
    pause: true
  });
  const $controls = $('<div/>')
    .attr('id', 'books-controls')
    .insertAfter($books);

  $('<button/>')
   ...
```

The jQuery UI plugin library

While most plugins, such as Cycle and Cookie, focus on a single task, jQuery UI tackles a wide variety of challenges. In fact, while the jQuery UI code may often be packaged as a single file, it is actually a comprehensive suite of related plugins.

The jQuery UI team has created a number of core interaction components and full-fledged widgets to help make the web experience more like that of a desktop application. Interaction components include methods for dragging, dropping, sorting, selecting, and resizing items. The current stable of widgets includes buttons, accordions, datepickers, dialogs, and so on. Additionally, jQuery UI provides an extensive set of advanced effects to supplement the core jQuery animations.

The full UI library is too extensive to be adequately covered within this chapter; indeed, there are entire books devoted to the subject. Fortunately, a major focus of the project is consistency among its features, so exploring a couple of pieces in detail will serve to get us started in using the rest of them as needed.

Downloads, documentation, and demos of all the jQuery UI modules are...

The jQuery Mobile plugin library

We have seen how jQuery UI can assist us in assembling the user interface features needed for even a complex web application. The challenges it overcomes are varied and complex. A different set of hurdles exists, however, when preparing our pages for elegant presentation and interaction on mobile devices. To create a website or application for modern smart phones and tablets, we can turn to the jQuery Mobile project.

Like jQuery UI, jQuery Mobile consists of a suite of related components that can be used *ala carte* but which work together seamlessly. The framework provides an Ajax-driven navigation system, mobile-optimized interactive elements, and advanced touch event handlers. Again, as with jQuery UI, exploring all the features of jQuery Mobile is a daunting task, so instead we will provide some simple examples and refer to the official documentation for more details.

Note

Downloads, documentation, and demos for jQuery Mobile are available athttp://jquerymobile.com/. Our jQuery Mobile example will use Ajax technology, so web server software will be required in order to try these examples. More...

Summary

In this chapter we have examined ways in which we can incorporate third-party plugins into our web pages. We've looked closely at the Cycle plugin, jQuery UI, and jQuery Mobile, and in the process have learned the patterns that we will encounter time and again in other plugins. In the next chapter, we'll take advantage of jQuery's plugin architecture to develop a few different types of plugins of our own.

Exercises

1. Increase the cycle transition duration to half a second, and change the animation such that each slide fades out before the next one fades in. Refer to the Cycle documentation to find the appropriate option to enable this.
2. Set the `cyclePaused` cookie to persist for 30 days.
3. Constrain the title box to resize only in ten pixel increments.
4. Make the slider animate smoothly from one position to the next as the slideshow advances.
5. Instead of letting the slideshow loop forever, cause it to stop after the last slide is shown. Disable the buttons and slider when this happens.
6. Create a new jQuery UI theme that has a light blue widget background and dark blue text and apply the theme to our sample document.
7. Modify the HTML in `mobile.html` so that the list view is divided up by the first letters of the book titles. See the jQuery Mobile documentation for `data-role="list-divider"` for details.

Chapter 8. Developing Plugins

The available third-party plugins provide a bevy of options for enhancing our coding experience, but sometimes we need to reach a bit farther. When we write code that could be reused by others or even just ourselves, we may want to package it up as a new plugin. Fortunately, the process of developing a plugin is not much more involved than writing the code that uses it.

In this chapter, we will cover:

- Adding new global functions within the `jQuery` namespace
- Adding jQuery object methods that allow us to act on DOM elements
- Creating widget plugins using the jQuery UI widget factory
- Distributing plugins

Using the dollar ($) alias in plugins

When we write jQuery plugins, we must assume that the jQuery library is loaded. We cannot assume, however, that the dollar (`$`) alias is available. Recall from Chapter 3, *Handling Events*, that the `$.noConflict()` method can relinquish control of this shortcut. To account for this, our plugins should always call jQuery methods using the full jQuery name or internally define `$` themselves.

Especially in larger plugins, many developers find that the lack of the dollar (`$`) shortcut makes code more difficult to read. To combat this, the shortcut can be locally defined for the scope of the plugin by defining a function and immediately invoking it. This syntax for defining and invoking a function at once, often referred to as an **Immediately Invoked Function Expression** (**IIFE**), looks like this:

```
(($) => {
  // Code goes here
})(jQuery);
```

The wrapping function takes a single parameter to which we pass the global `jQuery` object. The parameter is named `$`, so within the function we can use the dollar (`$`) alias with no conflicts.

Adding new global functions

Some of the built-in capabilities of jQuery are provided via what we have been calling global functions. As we've seen, these are actually methods of the jQuery object, but practically speaking, they are functions within a `jQuery` namespace.

A prime example of this technique is the `$.ajax()` function. Everything that `$.ajax()` does could be accomplished with a regular global function called `ajax()`, but this approach would leave us open for function name conflicts. By placing the function within the `jQuery` namespace, we only have to worry about conflicts with other jQuery methods. This `jQuery` namespace also signals to those who might use the plugin that the jQuery library is required.

Many of the global functions provided by the core jQuery library are utility methods; that is, they provide shortcuts for tasks that are frequently needed, but not difficult to do by hand. The array-handling functions `$.each()`, `$.map()`, and `$.grep()` are good examples of these. To illustrate the creation of such utility methods, we'll add two simple functions to their number.

To add a function to the `jQuery` namespace, we can just...

Adding jQuery object methods

Most of jQuery's built-in functionality is provided through its object instance methods, and this is where plugins shine as well. Whenever we would write a function that acts on part of the DOM, it is probably appropriate instead to create an **instance method**.

We have seen that adding global functions requires extending the `jQuery` object with new methods. Adding instance methods is similar, but we instead extend the `jQuery.fn` object:

```
jQuery.fn.myMethod = function() {
  alert('Nothing happens.');
};
```

Note

The `jQuery.fn` object is an alias to `jQuery.prototype`, provided for conciseness.

We can then call this new method from our code after using any selector expression:

```
$('div').myMethod();
```

Our alert is displayed (once for each `<div>` in the document) when we invoke the method. We might as well have written a global function, though, as we haven't used the matched DOM nodes in any way. A reasonable method implementation acts on its context.

Object method context

Within any plugin method, the keyword `this` is set to the current jQuery object. Therefore, we can call any built-in jQuery method on `this` or extract its...

Providing flexible method parameters

In Chapter 7, *Using Plugins*, we saw some plugins that can be fine-tuned to do exactly what we want through the use of parameters. We saw that a cleverly constructed plugin helps us by providing sensible defaults that can be independently overridden. When we make our own plugins, we should follow this example by keeping the user in mind.

To explore the various ways in which we can let a plugin's user customize its behavior, we need an example that has several settings that can be tweaked and modified. As our example, we'll replicate a feature of CSS by using a more brute-force JavaScript approach--an approach that is more suitable for demonstration than for production code. Our plugin will simulate a shadow on an element by creating a number of copies that are partially transparent overlaid in different positions on the page:

```
(function($) {
  $.fn.shadow = function() {
    return this.each((i, element) => {
      const $originalElement = $(element);

      for (let i = 0; i < 5; i++) {
        $originalElement
          .clone()
          .css({
            position: 'absolute',
            left:...
```

Creating plugins with the jQuery UI widget factory

As we saw in Chapter 7, *Using Plugins*, jQuery UI has an assortment of widgets--plugins that present a particular kind of UI element, such as a button or slider. These widgets present a consistent API to JavaScript programmers. This consistency makes learning to use one easy. When a plugin that we're writing will create a new user interface element, extending the jQuery UI library with a widget plugin is often the right choice.

A widget is an intricate piece of functionality, but fortunately we are not left to our own devices in creating one. The jQuery UI core contains a `factory` method called `$.widget()`, which does a lot of the work for us. Using this factory will help ensure that our code meets the API standards shared by all jQuery UI widgets.

Plugins we create using the widget factory have many nice features. We get all of these perks (and more) with very little effort on our part:

- The plugin becomes **stateful**, meaning that we can examine, alter, or even completely reverse the effects of the plugin after it has been applied
- User-supplied options are merged with customizable...

Plugin design recommendations

Now that we have examined common ways to extend jQuery and jQuery UI by creating plugins, we can review and supplement what we've learned with a list of recommendations:

- Protect the dollar (`$`) alias from potential interference from other libraries by using `jQuery` instead or passing `$` into an IIFE, so that it can be used as a local variable.
- Whether extending the jQuery object with `$.myPlugin` or the jQuery prototype with `$.fn.myPlugin`, add no more than one property to the `$` namespace. Additional public methods and properties should be added to the plugin's namespace (for example, `$.myPlugin.publicMethod` or `$.fn.myPlugin.pluginProperty`).
- Provide an object containing default options for the plugin: `$.fn.myPlugin.defaults = {size: 'large'}`.
- Allow the plugin user to optionally override any of the default settings for all subsequent calls to the method (`$.fn.myPlugin.defaults.size = 'medium';`) or for a single call (`$('div').myPlugin({size: 'small'});`).
- In most cases when extending the jQuery prototype `($.fn.myPlugin`), return `this` to allow the plugin user to chain additional jQuery methods to it (for example,

Summary

In this chapter, we have seen how the functionality that is provided by the jQuery core need not limit the library's capabilities. In addition to the readily available plugins we explored in Chapter 7, *Using Plugins*, we now know how to extend the menu of features ourselves.

The plugins we've created contain various features, including global functions that use the jQuery library, new methods of the jQuery object for acting on DOM elements, and sophisticated jQuery UI widgets. With these tools at our disposal, we can shape jQuery--and our own JavaScript code--into whatever form we desire.

Exercises

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

1. Create new plugin methods called `.slideFadeIn()` and `.slideFadeOut()`, combining the opacity animations of `.fadeIn()` and `.fadeOut()` with the height animations of `.slideDown()` and `.slideUp()`.
2. Extend the customizability of the `.shadow()` method so that the z-index of the cloned copies can be specified by the plugin user.
3. Add a new submethod called `isOpen` to the tooltip widget. This submethod should return `true` if the tooltip is currently displayed and `false` otherwise.
4. Add code that listens for the `tooltipopen` event that our widget fires and logs a message to the console.
5. **Challenge**: Provide an alternative `content` option for the tooltip widget that fetches the content of the page that an anchor's `href` points to via Ajax, and displays that content as the tooltip text.
6. **Challenge**: Provide a new `effect` option for the tooltip widget that, if specified, applies the named jQuery UI effect (such as `explode`) to the showing and hiding of the tooltip.

Chapter 9. Advanced Selectors and Traversing

In January 2009, jQuery's creator John Resig introduced a new open source JavaScript project called **Sizzle**. A standalone **CSS selector engine**, Sizzle was written to allow any JavaScript library to adopt it with little or no modification to its codebase. In fact, jQuery has been using Sizzle as its own selector engine ever since version 1.3.

Sizzle is the component within jQuery that is responsible for parsing the CSS selector expressions we put into the `$()` function. It determines which native DOM methods to use as it builds a collection of elements that we can then act on with other jQuery methods. The combination of Sizzle and jQuery's set of traversal methods makes jQuery an extremely powerful tool for finding elements on the page.

In Chapter 2, *Selecting Elements*, we looked at each of the basic types of selector and traversal method so that we have a roadmap of what's available to us in the jQuery library. In this more advanced chapter, we will cover:

- Using selectors to find and filter data in various ways
- Writing plugins that add new selectors and DOM traversal methods
- Optimizing our...

Selecting and traversing revisited

To kick off this more in-depth look into selectors and traversing, we'll build a script that will provide yet more selecting and traversing examples to inspect. For our sample, we'll build an HTML document containing a list of news items. We'll place those items in a table so that we can experiment with selecting rows and columns in several ways:

```
<div id="topics">
  Topics:
  <a href="topics/all.html" class="selected">All</a>
  <a href="topics/community.html">Community</a>
  <a href="topics/conferences.html">Conferences</a>
  <!-- continued... -->
</div>
<table id="news">
  <thead>
    <tr>
      <th>Date</th>
      <th>Headline</th>
      <th>Author</th>
      <th>Topic</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th colspan="4">2011</th>
    </tr>
    <tr>
      <td>Apr 15</td>
      <td>jQuery 1.6 Beta 1 Released</td>
      <td>John Resig</td>
      <td>Releases</td>
    </tr>
    <tr>
      <td>Feb 24</td>
      <td>jQuery Conference 2011: San Francisco Bay Area</td>
      <td>Ralph Whitbeck</td>
      <td>Conferences</td>
    </tr>
    <!-- continued......
```

Customizing and optimizing selectors

Many techniques that we've seen give us a tool chest that can be used to find any page element we want to work with. The story doesn't end here though; there is much to learn about performing our element-finding tasks efficiently. This efficiency can take the form of both code that is easier to write and read, and code that executes more quickly inside the web browser.

Writing a custom selector plugin

One way to improve legibility is to encapsulate code snippets in reusable components. We do this all the time by creating functions. In Chapter 8, *Developing Plugins*, we expanded this idea by crafting jQuery plugins that added methods to jQuery objects. This isn't the only way plugins can help us reuse code, though. Plugins can also provide additional **selector expressions**, such as the `:paused` selector that Cycle gave us in Chapter 7, *Using Plugins*.

The easiest type of selector expression to add is a **pseudo-class**. This is an expression that starts with a colon, such as `:checked` or `:nth-child()`. To illustrate the process of creating a selector expression, we'll build a pseudo-class called `:group()`....

DOM traversal under the hood

In Chapter 2, *Selecting Elements*, and again at the beginning of this chapter, we looked at ways of traveling from one set of DOM elements to another by calling DOM traversal methods. Our (far from exhaustive) survey of such methods included simple ways to reach neighboring cells, such as `.next()` and `.parent()`, and more complex ways of combining selector expressions, such as `.find()` and `.filter()`. By now, we should have a fairly strong grasp to these approaches of getting from one DOM element to another step by step.

Each time we take one of these steps, though, jQuery takes note of our travels, laying down a trail of breadcrumbs we can follow back home if needed. A couple of the methods we briefly touched on in that chapter, `.end()` and `.addBack()`, take advantage of this record keeping. To be able to get the most out of these methods, and in general to write efficient jQuery code, we need to understand a bit more about how the DOM traversal methods do their jobs.

jQuery traversal properties

As we know, we typically construct a jQuery object instance by passing a selector expression to the `$()` function....

Summary

In this chapter, we delved more deeply into jQuery's extensive capabilities for finding elements in a document. We've looked at some of the details of how the Sizzle selector engine works, and the implications this has on designing effective and efficient code. In addition, we have explored the ways in which we can extend and enhance jQuery's selectors and DOM traversal methods.

Further reading

A complete list of selectors and traversal methods is available in Appendix B, *Quick Reference* in this book, or in the official jQuery documentation at http://api.jquery.com/.

Exercises

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

1. Modify the table row striping routine so that it gives no class to the first row, a class of `alt` to the second row, and a class of `alt-2` to the third row. Repeat this pattern for every set of three rows in a section.
2. Create a new selector plugin called `:containsExactly()` that selects elements with text content that exactly matches what is put inside the parentheses.
3. Use this new `:containsExactly()`selector to rewrite the filtering code from *Listing 9.3*.
4. Create a new DOM traversal plugin method called `.grandparent()` that moves from an element or elements to their grandparent elements in the DOM.
5. **Challenge**: Using http://jsperf.com/, paste in the content of `index.html` and compare the performance of finding the closest ancestor table element of `<td id="release">` using the following:

- The `.closest()` method
- The `.parents()` method, limiting the result to the first table found

6. **Challenge**: Using http://jsperf.com/, paste in the content of `index.html` and compare the performance of finding the final `<td>` element in each row using the...

Chapter 10. Advanced Events

To build interactive web applications, we need to observe the user's activities and respond to them. We have seen that jQuery's event system can simplify this task, and we have already used this event system many times.

In Chapter 3, *Handling Events*, we touched upon a number of features that jQuery provides for reacting to events. In this more advanced chapter, we will cover:

- Event delegation and the challenges it presents
- Performance pitfalls associated with certain events and how to address them
- Custom events that we define ourselves
- The special event system that jQuery uses internally for sophisticated interactions

Revisiting events

For our sample document, we will create a simple photo gallery. The gallery will display a set of photos with an option to display additional photos upon the click of a link. We'll also use jQuery's event system to display textual information about each photo when the cursor is over it. The HTML that defines the gallery is as follows:

```
<div id="container">
  <h1>Photo Gallery</h1>

  <div id="gallery">
    <div class="photo">
      <img src="photos/skyemonroe.jpg">
      <div class="details">
        <div class="description">The Cuillin Mountains,
          Isle of Skye, Scotland.</div>
        <div class="date">12/24/2000</div>
        <div class="photographer">Alasdair Dougall</div>
      </div>
    </div>
    <div class="photo">
      <img src="photos/dscn1328.jpg">
      <div class="details">
        <div class="description">Mt. Ruapehu in summer</div>
        <div class="date">01/13/2005</div>
        <div class="photographer">Andrew McMillan</div>
      </div>
    </div>
    <div class="photo">
      <img src="photos/024.JPG">
      <div class="details">
        <div...
```

Event delegation

Recall that to implement event delegation by hand, we check the `target` property of the `event` object to see if it matches the element that we want to trigger the behavior. The event target represents the innermost, or most deeply nested, element that is receiving the event. With our sample HTML this time, however, we're presented with a new challenge. The `<div class="photo">` elements are unlikely to be the event target, since they contain other elements, such as the image itself and the image details.

What we need is the `.closest()` method, which works its way up the DOM from parent to parent until it finds an element that matches a given selector expression. If no elements are found, it acts like any other DOM traversal method, returning a new empty jQuery object. We can use `.closest()` to find `<div class="photo">` from any element it contains, as follows:

```
$(() => {
  $('#gallery')
    .on('mouseover mouseout', (e) => {
      const $target = $(e.target)
        .closest('div.photo');
      const $related = $(e.relatedTarget)
        .closest('div.photo');
      const $details = $target
        .find('.details');

   ...
```

Defining custom events

The events that get triggered naturally by the DOM implementations of browsers are crucial to any interactive web application. However, we aren't limited to this set of events in our jQuery code. We can also add our own custom events. We saw this briefly in Chapter 8, *Developing Plugins*, when we saw how jQuery UI widgets trigger events, but here we will investigate how we can create and use custom events outside of plugin development.

Custom events must be triggered manually by our code. In a sense, they are like regular functions that we define, in that we can cause a block of code to be executed when we invoke it from another place in the script. The `.on()` call for a custom event behaves like a function definition, while the `.trigger()` call acts like a function invocation.

However, event handlers are decoupled from the code that triggers them. This means that we can trigger events at any time, without knowing in advance what will happen when we do. A regular function call causes a single piece of code to be executed. A custom event, however, could have no handlers, one handler, or many handlers bound to...

Throttling events

A major issue with the infinite scrolling feature as we've implemented it in *Listing 10.10* is its performance impact. While our code is brief, the `checkScrollPosition()` function does need to do some work to measure the dimensions of the page and window. This effort can accumulate rapidly, because in some browsers the `scroll` event is triggered repeatedly during the scrolling of the window. The result of this combination could be choppy or sluggish performance.

Several native events have the potential for frequent triggering. Common culprits include `scroll`, `resize`, and `mousemove`. To account for this, we will implement **event throttling**. This technique involves limiting our expensive calculations so that they only occur after some of the event occurrences, rather than each one. We can update our code from *Listing 10.13* to implement this technique as follows:

```
$(() => {
  var timer = 0;

  $(window)
    .scroll(() => {
      if (!timer) {
        timer = setTimeout(() => {
          checkScrollPosition();
          timer = 0;
        }, 250);
      }
    })
    .trigger('scroll');
});
```

Listing 10.14

Rather than setting

Extending events

Some events, such as `mouseenter` and `ready`, are designated as **special events** by the jQuery internals. These events use the elaborate event extension framework offered by jQuery. Such events get the opportunity to take action at various times in the life cycle of an event handler. They may react to handlers being bound or unbound, and they can even have preventable default behaviors like clicked links or submitted forms do. The event extension API lets us create sophisticated new events that act much like native DOM events.

The throttling behavior we implemented for scrolling in *Listing 10.13* is useful, and we may want to generalize it for use in other projects. We can accomplish this by creating a new event that encapsulates the throttling technique within the special event hooks.

To implement special behavior for an event, we add a property to the `$.event.special` object. This added property, which is itself an object, has our event name as its key. It can contain callbacks called at many different specific times in an event's life cycle, including the following:

- `add`: This is called every time a handler for this...

Summary

The jQuery event system can be very powerful if we choose to leverage it fully. In this chapter, we have seen several aspects of the system, including event delegation methods, custom events, and the event extension API. We have also found ways of sidestepping pitfalls associated with delegation and with events that are triggered frequently.

Further reading

A complete list of event methods is available in Appendix B, *Quick Reference,* of this book, or in the official *jQuery documentation* at http://api.jquery.com/.

Exercises

The following challenge exercise may require the use of the official jQuery documentation at http://api.jquery.com/.

1. When the user clicks on a photo, add or remove the `selected` class on the photo `<div>`. Make sure this behavior works even for photos added later using the **`Next Page`** link.
2. Add a new custom event called `pageLoaded` that fires when a new set of images has been added to the page.
3. Using the `nextPage` and `pageLoaded` handlers, show a **`Loading`** message at the bottom of the page only while a new page is being loaded.
4. Bind a `mousemove` handler to photos that logs the current mouse position (using `console.log()`).
5. Revise this handler to perform the logging no more than five times a second.
6. **`Challenge:`** Create a new special event named `tripleclick` that fires when the mouse button is clicked on three times within 500 milliseconds. To test the event, bind a `tripleclick` handler to the `<h1>` element which hides and reveals the contents of `<div id="gallery">`.

Chapter 11. Advanced Effects

Since learning about jQuery's animation capabilities, we have found many uses for them. We can hide and reveal objects on the page with ease, we can gracefully resize elements, and we can smoothly reposition elements. This effects library is versatile, and contains even more techniques and specialized abilities than we have seen so far.

In Chapter 4, *Styling and Animating*, you learned about jQuery's basic animation capabilities. In this more advanced chapter, we will cover:

- Ways to gather information about the state of animations
- Methods for interrupting active animations
- Global effect options that can affect all animations on the page at once
- Deferred objects, which allow us to act once animations have completed
- Easing, which alters the rate at which animations occur

Animation revisited

To refresh our memory about jQuery's effect methods, we'll set up a baseline from which to build in this chapter, starting with a simple hover animation. Using a document with photo thumbnails on it, we'll make each photo *grow* slightly when the user's mouse is over it, and shrink back to its original size when the mouse leaves. The HTML tags we'll use also contain some textual information that's hidden for now, which we'll use later in the chapter:

```
<div class="team">
  <div class="member">
    <img class="avatar" src="photos/rey.jpg" alt="" />
    <div class="name">Rey Bango</div>
    <div class="location">Florida</div>
    <p class="bio">Rey Bango is a consultant living in South Fl
    specializing in web application development...</p>
  </div>
  <div class="member">
    <img class="avatar" src="photos/scott.jpg" alt="" />
    <div class="name">Scott González</div>
    <div class="location">North Carolina</div>
    <div class="position">jQuery UI Development Lead</div>
    <p class="bio">Scott is a web developer living in Raleigh,
  </div>
  <!-- Code continues...
```

Observing and interrupting animations

Our basic animation already reveals a problem. As long as there is enough time for the animation to complete after each `mouseenter` or `mouseleave` event, the animations proceed as intended. When the mouse cursor moves rapidly and the events are triggered quickly, however, we see that the images also grow and shrink repeatedly, well after the last event is triggered. This occurs because, as discussed in Chapter 4, *Styling and Animating*, animations on a given element are added to a queue and called in order. The first animation is called immediately, completes in the allotted time, and then is removed from the queue, at which point the next animation becomes first in line, is called, completes, is shifted, and so on until the queue is empty.

There are many cases in which this animation queue, known within jQuery as `fx`, causes desirable behavior. In the case of hover actions such as ours, though, it needs to be circumvented.

Determining the animation state

One way to avoid the undesirable queuing of animations is to use jQuery's custom `:animated` selector. Inside the `mouseenter`/`mouseleave` event...

Using global effect properties

The effects module in jQuery includes a handy `$.fx` object that we can access when we want to change the characteristics of our animations across the board. Although some of this object's properties are undocumented and intended to use solely within the library itself, others are provided as tools for globally altering the way our animations run. In the following examples, we'll take a look at a few of the documented properties.

Disabling all effects

We have already discussed a way to halt animations that are currently running, but what if we need to disable all animations entirely? We may, for example, wish to provide animations by default, but disable those animations for low-resource devices where animations could look choppy, or for users who find animations distracting. To do so, we can simply set the `$.fx.off` property to `true`. For our demonstration, we will display a previously hidden button to allow the user to toggle animations on and off:

```
$(() => {
  $('#fx-toggle')
    .show()
    .on('click', () => {
      $.fx.off = !$.fx.off;
    });
});
```

Listing 11.4

The hidden button is displayed between...

Multi-property easing

The `showDetails()` function almost accomplishes the unfolding effect we set out to achieve, but because the `top` and `left` properties are animating at the same rate, it looks more like a sliding effect. We can subtly alter the effect by changing the easing equation to `easeInQuart` for the `top` property only, causing the element to follow a curved path rather than a straight one. Remember, however, that using any easing other than `swing` or `linear` requires a plugin, such as the effects core of jQuery UI (http://jqueryui.com/).

```
.each((i, element) => {
  $(element)
    .animate({
      left: 0,
      top: 25 * i
    },{
      duration: 'slow',
      specialEasing: {
        top: 'easeInQuart'
      }
    });
 });
```

Listing 11.8

The `specialEasing` option allows us to set a different acceleration curve for each property that is being animated. Any properties that aren't included in the option will use the `easing` option's equation if it is provided, or the default `swing` equation if not.

We now have an attractive animation presenting most of the details associated with a team member. We aren't yet displaying a member's...

Using deferred objects

At times, we come across situations in which we want to act when a process completes, but we don't necessarily know how long the process will take, or even if it will be successful. To handle these cases, jQuery offers us **deferred objects** (promises). A deferred object encapsulates an operation that takes some time to complete.

A new deferred object can be created at any time by calling the `$.Deferred()` constructor. Once we have such an object, we can perform long-running operations and then call the `.resolve()` or `.reject()` methods on the object to indicate whether the operation was successful or unsuccessful. It is somewhat unusual to do this manually, however. Typically, rather than creating our own deferred objects by hand, jQuery or its plugins will create the object and take care of resolving or rejecting it. We just need to learn how to use the object that is created.

Note

Rather than detailing how the `$.Deferred()` constructor operates, we will focus here on how jQuery effects take advantage of deferred objects. In Chapter 13, *Advanced Ajax*, we will further explore deferred objects in the context of Ajax...

Taking fine-grained control of animations

Even though we've looked at a number of advanced features, jQuery's effects module has much more to explore. A rewrite of this module for jQuery 1.8 introduced a number of ways for advanced developers to fine-tune various effects and even change the underlying engine that drives the animations. For example, in addition to offering options such as `duration` and `easing`, the `.animate()` method provides a couple of callback options that let us inspect and modify an animation each step of the way:

```
$('#mydiv').animate({
  height: '200px',
  width: '400px'
}, {
  step(now, tween) {
   // monitor height and width
   // adjust tween properties
  },
  progress(animation, progress, remainingMs) {}
});
```

The `step()` function, which is called roughly once every 13 milliseconds for each animated property during the animation, allows us to adjust properties of the `tween` object such as the end value, the type of easing, or the actual property being animated based on the current value of a property via the passed `now` argument. A complex demonstration might, for example, use the `step()` function to...

Summary

In this chapter, we have further investigated several techniques that can assist us in crafting beautiful animations that are helpful to our users. We can now individually control the acceleration and deceleration of each property we are animating, and halt these animations individually or globally if needed. We learned about the properties that jQuery's effects library defines internally, and how to change some of them to suit our needs. We made our first foray into the jQuery deferred object system, which we will explore further in Chapter 13, *Advanced Ajax*, and we got a taste of the many opportunities to fine-tune jQuery's animation system.

Further reading

A complete list of effect and animation methods is available in *Appendix B* of this book, or in the official jQuery documentation at http://api.jquery.com/.

Exercises

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

1. Define a new animation speed constant called `zippy` and apply this to the biography display effect.
2. Change the easing of the horizontal movement of member details so that they bounce into place.
3. Add a second deferred callback function to the promise that adds a `highlight` class to the current member's location `<div>`.
4. **`Challenge`**: Add a delay of two seconds before animating the biography. Use the jQuery `.delay()` method.
5. **`Challenge:`** When the active photo is clicked, collapse the bio details. Stop any running animation before doing so.

Chapter 12. Advanced DOM Manipulation

Throughout this book, we have used jQuery's powerful DOM manipulation methods to alter the content of the document. We have now seen several ways in which we can insert new content, move existing content around, or remove content altogether. We also know how to alter the attributes and properties of elements to suit our needs.

In Chapter 5, *Manipulating the DOM*, we were introduced to these important techniques. In this more advanced chapter, we will cover:

- Sorting page elements using `.append()`
- Attaching custom data to elements
- Reading HTML5 data attributes
- Creating elements from JSON data
- Extending the DOM manipulation system using CSS hooks

Sorting table rows

The majority of the topics we're investigating in this chapter can be demonstrated through sorting the rows of a table. This common task is a useful way to assist users in quickly finding the information they need. There are, naturally, a number of ways to do this.

Sorting tables on the server

A common solution for data sorting is to perform it on the server. Data in tables often comes from a database, which means that the code that pulls it out of the database can request it in a given sort order (using, for example, the SQL language's `ORDER BY` clause). If we have server-side code at our disposal, it is straightforward to begin with a reasonable default sort order.

Sorting is most useful, though, when the user can determine the sort order. A common user interface for this is to make the table headers (`<th>`) of sortable columns into links. These links can go to the current page, but with a query string appended indicating the column to sort by, as shown in the following code snippet:

```
<table id="my-data">
  <thead>
    <tr>
      <th class="name">
        <a href="index.php?sort=name">Name</a>
      </th>
   ...
```

Moving and inserting elements revisited

Over the course of the coming examples, we will build a flexible sorting mechanism that works on each of the columns. To do this, we will use the jQuery DOM manipulation methods to insert some new elements and move other existing elements to new positions within the DOM. We will start with the most straightforward piece of the puzzle--linking the table headers.

Adding links around existing text

We'd like to turn the table headers into links that sort the data by their respective columns. We can use jQuery's `.wrapInner()` method to add them; we recall from Chapter 5, *Manipulating the DOM*, that `.wrapInner()` places a new element (in this case an `<a>` element) *inside* the matched element, but *around* child elements:

```
$(() => {
  const $headers = $('#t-1')
    .find('thead th')
    .slice(1);

  $headers
    .wrapInner($('<a/>').attr('href', '#'))
    .addClass('sort');
});
```

Listing 12.1

We skipped the first `<th>` element of each table (using `.slice()`) because it contains no text other than white space, as there is no need to either label or sort the cover photos. We then added a class of `sort` to the...

Storing data alongside DOM elements

Our code works, but it is quite slow. The culprit is the comparator function, which is doing a lot of work. This comparator will be called many times during the course of a sort, which means that it needs to be fast.

Note

`Array sorting performance`The actual sort algorithm used by JavaScript is not defined by the standard. It may be a simple sort such as a **bubble sort** (worst case of $\Theta(n^2)$ in computational complexity terms), or a more sophisticated approach such as a **quick sort** (which is $\Theta(n \log n)$ on average). It is safe to say, though, that doubling the number of items in an array will more than double the number of times the comparator function is called.

The remedy for our slow comparator is to **pre-compute** the keys for the comparison. We can do most of the expensive work in an initial loop and store the result with jQuery's `.data()` method, which sets or retrieves arbitrary information associated with page elements. Then we can simply examine the keys within the comparator function, and our sort is markedly faster:

```
$('#t-1')
  .find('thead th')
  .slice(1)
  .wrapInner($('<a/>').attr('href',...
```

Using HTML5 custom data attributes

So far, we've been relying on the content within the table cells to determine the sort order. While we've managed to sort the rows correctly by manipulating that content, we can make our code more efficient by outputting more HTML from the server in the form of **HTML5 data attributes**. The second table in our example page includes these attributes:

```
<table id="t-2" class="sortable">
  <thead>
    <tr>
      <th></th>
      <th data-sort='{"key":"title"}'>Title</th>
      <th data-sort='{"key":"authors"}'>Author(s)</th>
      <th data-sort='{"key":"publishedYM"}'>Publish Date</th>
      <th data-sort='{"key":"price"}'>Price</th>
    </tr>
  </thead>
  <tbody>
    <tr data-book='{"img":"2862_OS.jpg",
      "title":"DRUPAL 7","authors":"MERCER DAVID",
      "published":"September 2010","price":44.99,
      "publishedYM":"2010-09"}'>
      <td><img src="images/2862_OS.jpg" alt="Drupal 7"></td>
      <td>Drupal 7</td>
      <td>David Mercer</td>
      <td>September 2010</td>
      <td>$44.99</td>
    </tr>
    <!-- code continues -->
  </tbody>
</table>
```

Notice that...

Sorting and building rows with JSON

So far in this chapter, we have been moving in the direction of outputting more and more information from the server into HTML so that our client-side scripts can remain as lean and efficient as possible. Now let's consider a different scenario, one in which a whole new set of information is displayed when JavaScript is available. Increasingly, web applications rely on JavaScript to deliver content as well as manipulate it once it arrives. In our third table sorting example, we'll do the same.

We'll start by writing three functions:

- `buildAuthors()`: This builds a string list of author names
- `buildRow()`: This builds the HTML for a single table row
- `buildRows()`: This builds the HTML for the entire table by mapping the rows built by `buildRow()`

```
const buildAuthors = row =>
  row
    .authors
    .map(a => `${a.first_name} ${a.last_name}`)
    .join(', ');

const buildRow = row =>
  `
    <tr>
      <td><img src="images/${row.img}"></td>
      <td>${row.title}</td>
      <td>${buildAuthors(row)}</td>
      <td>${row.published}</td>
      <td>$${row.price}</td>
    </tr>
  `;

const buildRows = rows =>
 ...
```

Revisiting attribute manipulation

By now, we are used to getting and setting values that are associated with DOM elements. We have done this with simple methods such as `.attr()`, `.prop()`, and `.css()`, convenient shorthands such as `.addClass()`, `.css()`, and `.val()`, and complex bundles of behavior such as `.animate()`. Even the simple methods, though, do quite a bit of work for us behind the scenes. We can get even more utility out of them if we better understand what they do.

Using shorthand element creation syntax

We often create new elements in our jQuery code by providing an HTML string to the `$()` function or to DOM insertion functions. For example, we create a large HTML fragment in *Listing 12.9* in order to produce many DOM elements. This technique is fast and concise. There are circumstances when it is not ideal. We might, for instance, want to escape special characters from text before it is used, or apply style rules that are browser-dependent. In these cases, we can create the element and then chain on additional jQuery methods to alter it, as we have done many times already. In addition to this standard technique, the `$()`...

Summary

In this chapter, we have solved a common problem--sorting a data table--in three different ways, comparing the benefits of each approach. In doing so, we practiced the DOM modification techniques that we have learned earlier and explored the `.data()` method for getting and setting data associated with any DOM element or attached using HTML5 data attributes. We also pulled back the curtain on several DOM modification routines, learning how to extend them for our own purposes.

Further reading

A complete list of DOM manipulation methods is available in *Appendix C* of this book, or in the official jQuery documentation at http://api.jquery.com/.

Exercises

The challenge exercise may require the use of the official jQuery documentation at http://api.jquery.com/.

1. Modify the key computation for the first table so that titles and authors are sorted by length rather than alphabetically.
2. Use the HTML5 data in the second table to compute the sum of all of the book prices and insert this sum into the heading for that column.
3. Change the comparator used for the third table so that titles containing the word **`jQuery`** come first when sorted by title.
4. **`Challenge`**: Implement the `get` callback for the `glowColor` CSS hook.

Chapter 13. Advanced Ajax

Many web applications require frequent network communication. Using jQuery, our web pages can exchange information with the server without requiring new pages to be loaded in the browser.

In Chapter 6, *Sending Data with Ajax*, you learned simple ways to interact with the server asynchronously. In this more advanced chapter, we will cover:

- Error-handling techniques for dealing with network interruptions
- The interactions between Ajax and the jQuery deferred object system
- Caching and throttling techniques for reducing network traffic
- Ways to extend the inner workings of the Ajax system using transports, prefilters, and data type converters

Implementing progressive enhancement with Ajax

Throughout this book, we encountered the concept of *progressive enhancement*. To reiterate, this philosophy ensures a positive user experience for all users by mandating that a working product be put in place first before additional embellishments are added for users with modern browsers.

As an example, we'll build a form that searches GitHub repositories:

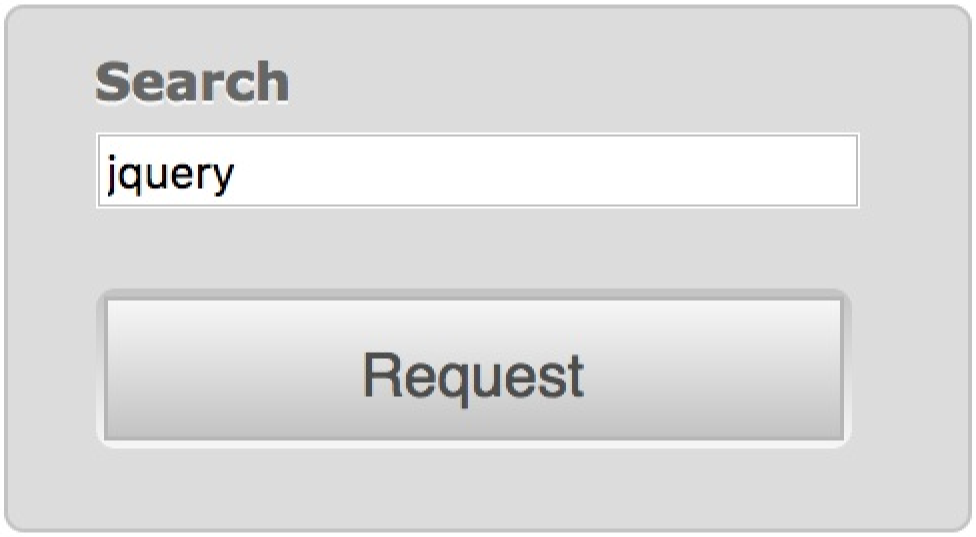
```
<form id="ajax-form" action="https://github.com/search" method=
  <fieldset>
    <div class="text">
      <label for="title">Search</label>
      <input type="text" id="title" name="q">
    </div>

    <div class="actions">
      <button type="submit">Request</button>
    </div>
  </fieldset>
</form>
```

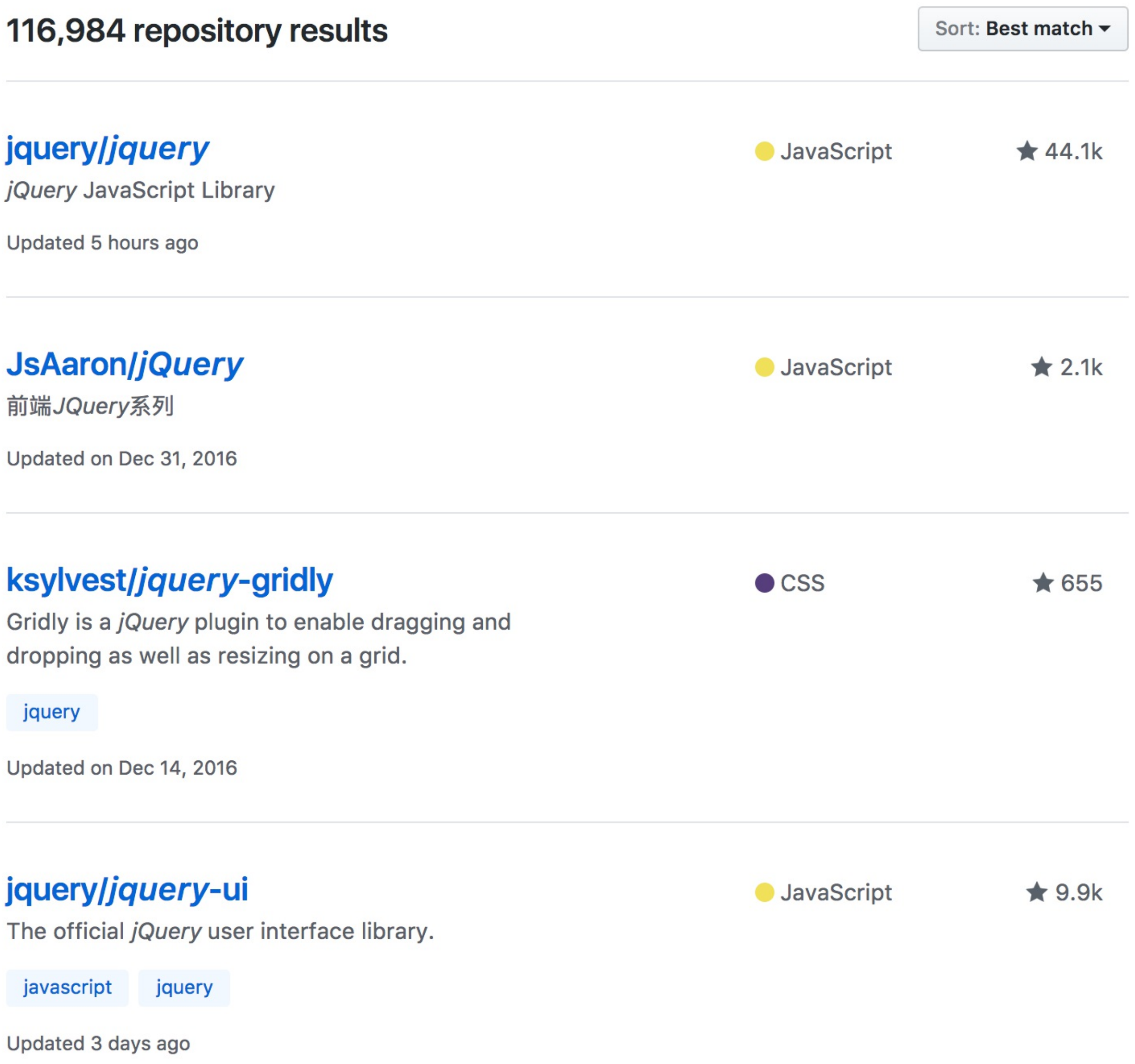
Note

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

The search form is a normal form element with a text input and a submit button labeled as **`Request`**:



When the **`Request`** button of this form is clicked on, the form submits as normal; the user's browser is directed to https://github.com/search and the results are displayed:



116,984 repository results

Sort: **Best match** ▾

jquery/*jquery*
jQuery JavaScript Library
● JavaScript ★ 44.1k
Updated 5 hours ago

JsAaron/*jQuery*
前端*JQuery*系列
● JavaScript ★ 2.1k
Updated on Dec 31, 2016

ksylvest/*jquery*-gridly
Gridly is a *jQuery* plugin to enable dragging and dropping as well as resizing on a grid.
● CSS ★ 655
jquery
Updated on Dec 14, 2016

jquery/*jquery*-ui
The official *jQuery* user interface library.
● JavaScript ★ 9.9k
javascript jquery
Updated 3 days ago

Handling Ajax errors

Introducing any kind of network interaction into an application brings along some degree of uncertainty. The user's connection could drop in the middle of an operation or a temporary server issue could interrupt communications. Because of these reliability concerns, we should always plan for the worst case and prepare for error scenarios.

The `$.ajax()` function can take a callback function named `error` to be called in these situations. In this callback, we should provide some kind of feedback to the user indicating that an error has occurred:

```
$(() => {
  $('#ajax-form')
    .on('submit', (e) => {
      e.preventDefault();

      $.ajax({
        url: 'https://api.github.com/search/repositories',
        dataType: 'jsonp',
        data: { q: $('#title').val() },
        error() {
          $('#response').html('Oops. Something went wrong...');
        }
      });
    });
});
```

Listing 13.5

The error callback can be triggered for a number of reasons. Among these are:

- The server returned an error status code, such as **`403 Forbidden`**, **`404 Not Found`**, or **`500 Internal Server Error`**.
- The server returned a redirection status code,...

Using the jqXHR object

When an Ajax request is made, jQuery determines the best mechanism available for retrieving the data. This transport could be the standard `XMLHttpRequest` object, the Microsoft ActiveX `XMLHTTP` object, or a `<script>` tag.

Because the transport used can vary from request to request, we need a common interface in order to interact with the communication. The `jqXHR` object provides this interface for us. It is a wrapper for the `XMLHttpRequest` object when that transport is used, and in other cases, it simulates `XMLHttpRequest` as best it can. Among the properties and methods it exposes are:

- `.responseText` or `.responseXML`, containing the returned data
- `.status` and `.statusText`, containing a status code and description
- `.setRequestHeader()` to manipulate the HTTP headers sent with the request
- `.abort()` to prematurely halt the transaction

This `jqXHR` object is returned from all of jQuery's Ajax methods, so we can store the result if we need access to any of these properties or methods.

Ajax promises

Perhaps a more important aspect of `jqXHR` than the `XMLHttpRequest` interface, however, is that it also acts as a promise. In Chapter 11,

Throttling Ajax requests

A common feature of searches is to display a dynamic list of results as the user is typing. We can emulate this "live search" feature for our jQuery API search by binding a handler to the `keyup` event:

```
$('#title')
  .on('keyup', (e) => {
    $(e.target.form).triggerHandler('submit');
  });
```

Listing 13.10

Here, we simply trigger the form's submit handler whenever the user types something in the **`Search`** field. This could have the effect of sending many requests across the network in rapid succession, depending on the speed at which the user types. This behavior could bog down JavaScript's performance; it could clog the network connection, and the server might not be able to handle that kind of demand.

We're already limiting the number of requests with the request caching that we've just put in place. We can further ease the burden on the server, however, by throttling the requests. In Chapter 10, *Advanced Events*, we introduced the concept of throttling when we created a special `throttledScroll` event to reduce the number of times the native scroll event is fired. In this case, we want to make a similar reduction in...

Extending Ajax capabilities

The jQuery Ajax framework is powerful, as we've seen, but even so there are times when we might want to change the way it behaves. Unsurprisingly, it offers multiple hooks that can be used by plugins to give the framework brand new capabilities.

Data type converters

In Chapter 6, *Sending Data with Ajax*, we saw that the `$.ajaxSetup()` function allows us to change the defaults used by `$.ajax()`, thus potentially affecting many Ajax operations with a single statement. This same function can also be used to add to the range of data types that `$.ajax()` can request and interpret.

As an example, we can add a converter that understands the YAML data format. YAML (http://www.yaml.org/) is a popular data representation with implementations in many programming languages. If our code needs to interact with an alternative format such as this, jQuery allows us to build compatibility for it into the native Ajax functions.

A simple YAML file containing GitHub repository search criteria:

```
Language:
 - JavaScript
 - HTML
 - CSS
Star Count:
 - 5000+
 - 10000+
 - 20000+
```

We can wrap jQuery around an existing YAML parser such as...

Summary

In this final chapter, we have taken an in-depth look at jQuery's Ajax framework. We can now craft a seamless user experience on a single page, fetching external resources when needed with proper attention to error handling, caching, and throttling. We explored details of the inner operations of the Ajax framework, including promises, transports, prefilters, and converters. You also learned how to extend these mechanisms to serve the needs of our scripts.

Further reading

A complete list of *Ajax methods* is available in Appendix B, *Quick Reference*, of this book or in the official jQuery documentation at http://api.jquery.com/.

Exercises

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

1. Alter the `buildItem()` function so that it includes the long description of each jQuery method it displays.
2. Here's a challenge for you. Add a form to the page that points to a Flickr public photo search (http://www.flickr.com/search/) and make sure it has `<input name="q">` and a submit button. Use progressive enhancement to retrieve the photos from Flickr's JSONP feed service at http://api.flickr.com/services/feeds/photos_public.gne instead and insert them into the content area of the page. When sending data to this service, use `tags` instead of `q` and set `format` to `json`. Also note that rather than `callback`, the service expects the JSONP callback name to be `jsoncallback`.
3. Here's another challenge for you. Add error handling for the Flickr request in case it results in `parsererror`. Test it by setting the JSONP callback name back to `callback`.

Chapter 14. Testing JavaScript with QUnit

Throughout this book we've written a lot of JavaScript code, and we've seen the many ways in which jQuery helps us write this code with relative ease. Yet whenever we've added a new feature, we've had to take the extra step of manually checking our web page to ensure that everything is working as expected. While this process may work for simple tasks, as projects grow in size and complexity, manual testing can become quite onerous. New requirements can introduce *regression bugs* that break parts of the script that previously worked well. It's far too easy to miss these bugs that don't specifically relate to the latest code changes because we naturally only test for what we've just done.

What we need instead is an automated system that runs our tests for us. The **QUnit** testing framework is just such a system. While there are many other testing frameworks, and they all have their own benefits, we recommend QUnit for most jQuery projects because it is written and maintained by the jQuery project. In fact, jQuery itself uses QUnit. In this appendix, we will cover:

- How to set up the QUnit...

Downloading QUnit

The QUnit framework can be downloaded from the official QUnit website at http://qunitjs.com/. There we can find links to the stable version (currently 2.3.0) as well as a development version (qunit-git). Both versions include a style sheet in addition to the JavaScript file for formatting the test output.

Setting up the document

Once we have the QUnit files in place, we can set up the test HTML document. In a typical project, this file would be named `index.html` and placed in the same test subfolder as `qunit.js` and `qunit.css`. For this demonstration, however, we'll put it in the parent directory.

The `<head>` element of the document contains a `<link>` tag for the CSS file and `<script>` tags for jQuery, QUnit, the JavaScript we'll be testing (`A.js`), and the tests themselves (`listings/A.*.js`). The `<body>` tag consists of two main elements for running and displaying the results of the tests.

To demonstrate QUnit, we'll use portions of Chapter 2, *Selecting Elements*, and Chapter 6, *Sending Data with Ajax*:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Appendix A Tests</title>
  <link rel="stylesheet" href="qunit.css" media="screen">
  <script src="jquery.js"></script>
  <script src="test/qunit.js"></script>
  <script src="A.js"></script>
  <script src="test/test.js"></script>
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture">
    <!-- Test Markup Goes Here -->
  </div>
</body>
</html>
```

Since

Organizing tests

QUnit provides two levels of test grouping named after their respective function calls: `QUnit.module()` and `QUnit.test()`. The **module** is like a general category under which the tests will be run; the test is actually a *set* of tests; the function takes a callback in which all of that test's specific **unit tests** are run. We'll group our tests by the chapter topic and place the code in our `test/test.js` file:

```
QUnit.module('Selecting');

QUnit.test('Child Selector', (assert) => {
  assert.expect(0);
});

QUnit.test('Attribute Selectors', (assert) => {
  assert.expect(0);
});

QUnit.module('Ajax');
```

Listing A.1

It's not necessary to set up the file with this test structure, but it's good to have some overall structure in mind. In addition to the `QUnit.module()` and `QUnit.test()` grouping, we have to tell the test how many assertions to expect. Since we're just getting organized, we need to tell the test that there aren't any assertions yet (`assert.expect(0)`) in order for the tests to run.

Notice that our modules and tests do not need to be placed inside a `$(() => {})` call because QUnit by default waits until the window has...

Adding and running tests

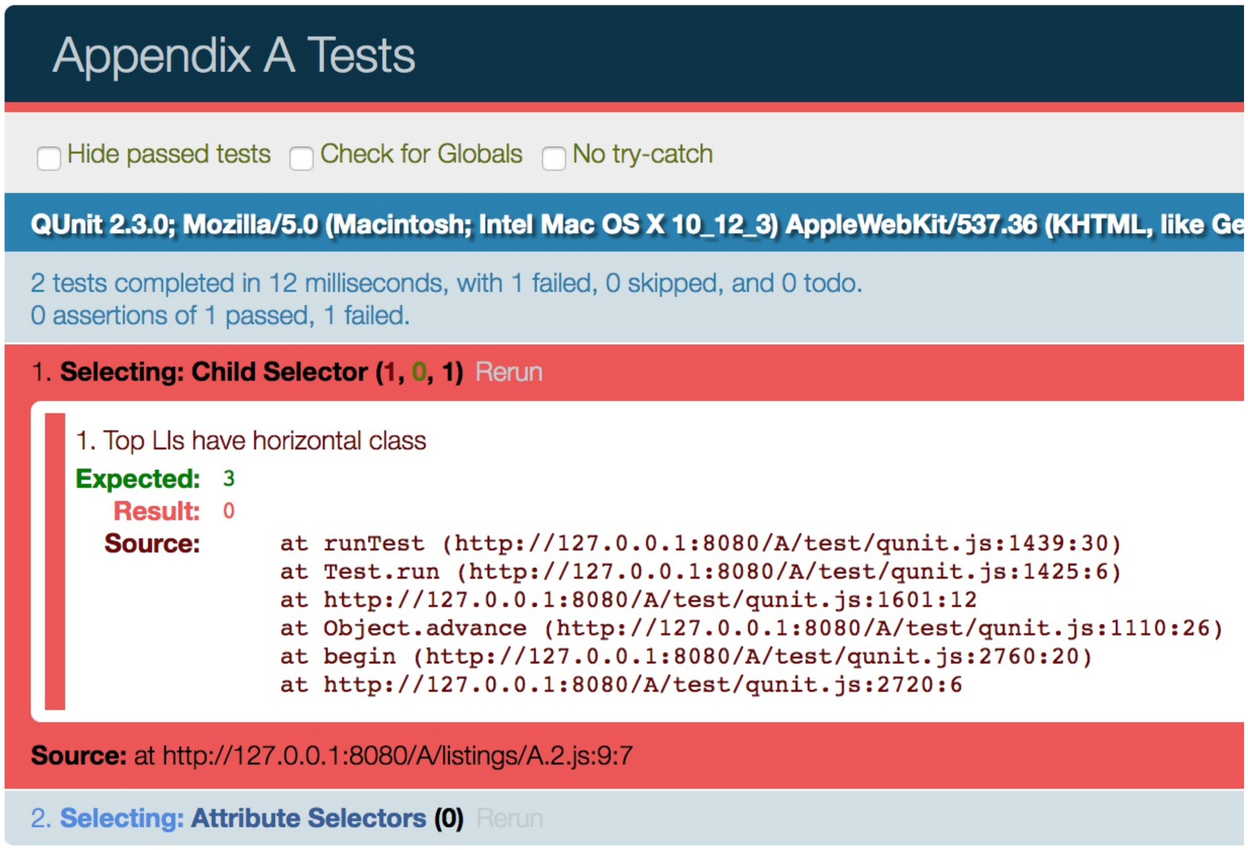
In **test-driven development**, we write tests before writing code. This way, we can observe when a test fails, add new code, and then see that the test passes, verifying that our change has the intended effect.

Let's start by testing the child selector that we used in Chapter 2, *Selecting Elements*, to add a `horizontal` class to all `<li>` elements that are children of `<ul id="selected-plays">`:

```
QUnit.test('Child Selector', (assert) => {
  assert.expect(1);
  const topLis = $('#selected-plays > li.horizontal');
  assert.equal(topLis.length, 3, 'Top LIs have horizontal class
});
```

Listing A.2

We're testing our ability to select elements on the page, so we use the assert `assert.equal()` test to compare the number of top-level `<li>` elements against the number `3`. If the two are equal, the test is successful and is added to the number of passed tests. If not, the test fails:



Appendix A Tests

Hide passed tests Check for Globals No try-catch

QUnit 2.3.0; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_3) AppleWebKit/537.36 (KHTML, like Ge

2 tests completed in 12 milliseconds, with 1 failed, 0 skipped, and 0 todo.
0 assertions of 1 passed, 1 failed.

1. **Selecting: Child Selector (1, 0, 1)** Rerun

1. Top LIs have horizontal class

Expected: 3

Result: 0

Source:

```
at runTest (http://127.0.0.1:8080/A/test/qunit.js:1439:30)
at Test.run (http://127.0.0.1:8080/A/test/qunit.js:1425:6)
at http://127.0.0.1:8080/A/test/qunit.js:1601:12
at Object.advance (http://127.0.0.1:8080/A/test/qunit.js:1110:26)
at begin (http://127.0.0.1:8080/A/test/qunit.js:2760:20)
at http://127.0.0.1:8080/A/test/qunit.js:2720:6
```

Source: at http://127.0.0.1:8080/A/listings/A.2.js:9:7

2. **Selecting: Attribute Selectors (0)** Rerun

Of course, the test fails because we have not yet written the code to add the `horizontal` class. It is simple to add that code, though. We do so in the main script file for the page, which we called `A.js`:

```
$(() => {
  $('#selected-plays >...
```

Other types of tests

QUnit comes with a number of other test functions as well. Some, such as `notEqual()` and `notDeepEqual()`, are simply the inverses of functions we've used, while others, such as `strictEqual()` and `throws()`, have more distinct uses. More information about these functions, as well as details and additional examples regarding QUnit in general, are available on the QUnit website (http://qunitjs.com/) as well as the QUnit API site (http://api.qunitjs.com/).

Practical considerations

The examples in this appendix have been necessarily simple. In practice, we can write tests that ensure the correct operation of quite complicated behaviors.

Ideally, we keep our tests as brief and simple as possible, even when the behaviors they are testing are intricate. By writing tests for a few specific scenarios, we can be reasonably certain that we are fully testing the behavior, even though we do not have a test for every possible set of inputs.

However, it is possible that an error is observed in our code even though we have written tests for it. When tests pass and yet an error occurs, the correct response is not to immediately fix the problem, but rather to first write a new test for the behavior that fails. This way, we can not only verify that the problem is solved when we correct the code, but also introduce an additional test that will help us avoid regressions in the future.

QUnit can be used for **functional testing** in addition to **unit testing**. While unit tests are designed to confirm the correct operation of code units (methods and functions), functional tests are written to ensure...

Summary

Writing tests with QUnit can be an effective aid in keeping our jQuery code clean and maintainable. We've seen just a few ways that we can implement tests in a project to ensure that our code is functioning the way we intend it to. By testing small, discrete units of code, we can mitigate some of the problems that occur when projects become more complex. At the same time, we can more efficiently test for regressions throughout a project, saving us valuable programming time.

Chapter 15. Quick Reference

This appendix is intended to be a quick reference for the jQuery API, including its selector expressions and methods. A more detailed discussion of each method and selector is available on the jQuery documentation site, http://api.jquery.com.

Selector expressions

The jQuery factory function `$()` is used to find elements on the page to work with. This function takes a string composed of CSS-like syntax, called a selector expression. Selector expressions are discussed in detail in Chapter 2, *Selecting Elements*.

Simple CSS

Selector	Matches
`*`	All elements.
`#id`	The element with the given ID.
`element`	All elements of the given type.
`.class`	All elements with the given class.
`a, b`	Elements that are matched by `a` or `b`.
`a b`	Elements `b` that are descendants of `a`.
`a > b`	Elements `b` that are children of `a`.
`a + b`	Elements `b` that immediately follow `a`.

`a ~ b`	Elements `b` that are siblings of `a` and follow `a`.

Position among siblings

Selector	**Matches**
`:nth-child(index)`	Elements that are the `index` child of their parent element (1-based).
`:nth-child(even)`	Elements that are an even child of their parent element (1-based).
`:nth-child(odd)`	Elements that are an odd child of their parent element (1-based).
`:nth-child(formula)`	Elements that are the nth child of their parent element (1-based). Formulas are of the form `an+b` for integers `a` and `b`.
`:nth-last-child()`	The same as `:nth-child()`, but counting from the last element to the first.

DOM traversal methods

After creating a jQuery object using `$()`, we can alter the set of matched elements we are working with by calling one of these DOM traversal methods. DOM traversal methods are discussed in detail in Chapter 2, *Selecting Elements*.

Filtering

Traversal method	**Returns a jQuery object containing...**
`.filter(selector)`	Selected elements that match the given selector.
`.filter(callback)`	Selected elements for which the callback function returns `true`.
`.eq(index)`	The selected element at the given 0-based index.
`.first()`	The first selected element.
`.last()`	The final selected element.
`.slice(start, [end])`	Selected elements in the given range of 0-based indices.
`.not(selector)`	Selected elements that do not match the given selector.

`.has(selector)`	Selected elements that have a descendant matching `selector`.

Descendants

Traversal method	Returns a jQuery object containing...
`.find(selector)`	Descendant elements that match the selector.
`.contents()`	Child nodes (including text nodes).
`.children([selector])`	Child nodes, optionally filtered by a selector.

Siblings

Traversal method	Returns a jQuery object containing...
`.next([selector])`	The sibling immediately...

Event methods

To react to user behavior, we need to register our handlers using these event methods. Note that many DOM events only apply to certain element types; these subtleties are not covered here. Event methods are discussed in detail in Chapter 3, *Handling Events*.

Binding

Event method	Description
`.ready(handler)`	Binds `handler` to be called when the DOM and CSS are fully loaded.
`.on(type, [selector], [data], handler)`	Binds `handler` to be called when the given type of event is sent to the element. If `selector` is provided, performs event delegation.
`.on(events, [selector], [data])`	Binds multiple handlers for events as specified in the `events` object parameter.
`.off(type, [selector], [handler])`	Removes bindings on the element.
`.one(type, [data], handler)`	Binds `handler` to be called when the given type of event is sent to the element. Removes the binding when the handler is called.

Shorthand binding

Event method	Description
`.blur(handler)`	Binds `handler` to be called when the element loses keyboard focus.
`.change(handler)`	Binds `handler` to be called when the element's value changes.
`.click(handler)`	Binds `handler` to be called when the element is clicked.

Effect methods

These effect methods may be used to perform animations on DOM elements. The effect methods are discussed in detail in Chapter 4, *Styling and Animating*.

Predefined effects

Effect method	Description
`.show()`	Displays the matched elements.
`.hide()`	Hides the matched elements.
`.show(speed, [callback])`	Displays the matched elements by animating `height`, `width`, and `opacity`.
`.hide(speed, [callback])`	Hides the matched elements by animating `height`, `width`, and `opacity`.
`.slideDown([speed], [callback])`	Displays the matched elements with a sliding motion.
`.slideUp([speed], [callback])`	Hides the matched elements with a sliding motion.
`.slideToggle([speed],`	Displays or hides the matched elements with a

`[callback])`	sliding motion.
`.fadeIn([speed], [callback])`	Displays the matched elements by fading them to opaque.
`.fadeOut([speed], [callback])`	Hides the matched elements by fading them to transparent.
`.fadeToggle([speed], [callback])`	Displays or hides the matched elements with a fading animation.
`.fadeTo(speed, opacity, [callback])`	Adjusts the opacity of the matched elements.

Custom animations

Effect method	Description
`.animate(properties, [speed], [easing],...`	

DOM manipulation methods

The DOM manipulation methods are discussed in detail in Chapter 5, *Manipulating the DOM*.

Attributes and properties

Manipulation method	Description
`.attr(key)`	Gets the attribute named `key`.
`.attr(key, value)`	Sets the attribute named `key` to `value`.
`.attr(key, fn)`	Sets the attribute named `key` to the result of `fn` (called separately on each matched element).
`.attr(obj)`	Sets attribute values given as key-value pairs.
`.removeAttr(key)`	Removes the attribute named `key`.
`.prop(key)`	Gets the property named `key`.
`.prop(key, value)`	Sets the property named `key` to `value`.
`.prop(key, fn)`	Sets the property named `key` to the result of `fn` (called separately on each matched element).

`.prop(obj)`	Sets property values given as key-value pairs.
`.removeProp(key)`	Removes the property named `key`.
`.addClass(class)`	Adds the given class to each matched element.
`.removeClass(class)`	Removes the given class from each matched element.
`.toggleClass(class)`	Removes the given class if present, and adds it if not, for each matched element.
`.hasClass(class)`	Returns `true` if any of the matched elements has the given class.
`.val()`	Gets the value attribute of the first matched element.

Ajax methods

We can retrieve information from the server without requiring a page refresh by calling one of these Ajax methods. Ajax methods are discussed in detail in Chapter 6, *Sending Data with Ajax*.

Issuing requests

Ajax method	Description
`$.ajax([url], options)`	Makes an Ajax request using the provided set of options. This is a low-level method that is often called via other convenience methods.
`.load(url, [data], [callback])`	Makes an Ajax request to `url` and places the response into the matched elements.
`$.get(url, [data], [callback], [returnType])`	Makes an Ajax request to `url` using the `GET` method.
`$.getJSON(url, [data], [callback])`	Makes an Ajax request to `url`, interpreting the response as a JSON data structure.
`$.getScript(url, [callback])`	Makes an Ajax request to `url`, executing the response as JavaScript.
`$.post(url,`	

`[data],`
`[callback],`
`[returnType])`
	Makes an Ajax request to `url` using the `POST` method.

Request monitoring

Ajax method	**Description**
`.ajaxComplete(handler)`	Binds `handler` to be called when any Ajax transaction completes.
`.ajaxError(handler)`	Binds `handler` to be called when any Ajax transaction completes with an error.
`.ajaxSend(handler)`	

Deferred objects

Deferred objects and their promises allow us to react to the completion of long-running tasks with a convenient syntax. They are discussed in detail in Chapter 11, *Advanced Effects*.

Object creation

Function	Description
`$.Deferred([setupFunction])`	Returns a new deferred object.
`$.when(deferreds)`	Returns a promise object to be resolved when the given deferred objects are resolved.

Methods of deferred objects

Method	Description
`.resolve([args])`	Sets the state of the object to resolved.
`.resolveWith(context, [args])`	Sets the state of the object to resolved while making the keyword `this` refer to `context` within callbacks.

`.reject([args])`	Sets the state of the object to rejected.
`.rejectWith(context, [args])`	Sets the state of the object to rejected while making the keyword `this` refer to `context` within callbacks.
`.notify([args])`	Executes any progress callbacks.
`.notifyWith(context, [args])`	Executes any progress callbacks while making the keyword `this` refer to `context`.
`.promise([target])`	Returns a promise object corresponding to this deferred object.

Methods of promise objects

Method	**Description**
`.done(callback)`	Executes `callback` when the object is...

Miscellaneous properties and functions

These utility methods do not fit neatly into the previous categories, but are often very useful when writing scripts using jQuery.

Properties of the jQuery object

Property	Description
`$.ready`	A promise instance that's resolved as soon as the DOM is ready.

Arrays and objects

Function	Description
`$.each(collection, callback)`	Iterates over `collection`, executing `callback` for each item.
`$.extend(target, addition, ...)`	Modifies the object `target` by adding properties from the other supplied objects.
`$.grep(array, callback, [invert])`	Filters `array` by using `callback` as a test.
`$.makeArray(object)`	Converts `object` into an array.

`$.map(array, callback)`	Constructs a new array consisting of the result of `callback` being called on each item.
`$.inArray(value, array)`	Determines whether `value` is in `array`.
`$.merge(array1, array2)`	Combines the contents of `array1` and `array2`.
`$.unique(array)`	Removes any duplicate DOM elements from `array`.

Object introspection

Function	Description
`$.isArray(object)`	Determines whether `object` is a true JavaScript array.
`$.isEmptyObject(object)`	Determines whether `object` is empty.
`$.isFunction(object)`	Determines...