

Bear Bibeault  
Yehuda Katz  
Aurelio De Rosa

# jQuery

## IN ACTION

THIRD EDITION



 MANNING



**MEAP Edition  
Manning Early Access Program  
jQuery in Action, Third Edition  
Version 8**

Copyright 2015 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

# welcome

---

Thanks for purchasing the MEAP for *jQuery in Action, third edition*. We are taking the best from the previous editions, updating content to reflect changes in jQuery, and including a shiny new Part 3 that covers advanced topics. We're working really hard to provide the best resource for those who want to use jQuery in the most effective way possible. To that end we are adding in more examples that will help readers to easily learn the concepts discussed.

This book is for web developers at all levels. Beginners will find in it a complete guide to becoming a pro. They will learn how to use the main modules of the library, and the best practices involved. Mid-level and advanced developers will learn about advanced topics like Unit Testing and Deferred, and less-known techniques and tricks.

We're starting the MEAP with the first three chapters of the book. Chapter 1 explains basic concepts like the distinction between markup, style, and business logic, and the concept of unobtrusive JavaScript. Then, you'll learn about the several different versions of jQuery available, and how to include it in your pages. Finally, you'll discover more about the jQuery object and the concept of a wrapped set.

Chapter 2 is completely dedicated to jQuery selectors. Reading it, you'll become familiar with the plethora of selectors available, and you'll learn how to create custom filters (also referred as a custom selector or pseudo selector). The chapter also discusses context, the second parameter of the jQuery() method, describing the implications of its use and how it can improve your code performance.

Chapter 3 discusses how to create new DOM elements using the high flexibility of the jQuery() function (and its \$() alias). This capability is especially useful for injecting external data into a web page using the JSON and the XML format. Then, we'll cover methods that, starting from a wrapped set, accept a selector as a parameter to create a new set of elements. The last part of this chapter is dedicated to methods that aren't strictly related to selectors, but allow us to iterate over the elements in a wrapped set or perform a test on them.

As you're reading, we hope you'll take advantage of the Author Online forum. We'll be reading your comments and responding. We appreciate any feedback, as it is very helpful in the development process.

Thanks again!

— Aurelio De Rosa, Bear Bibeault, and Yahuda Katz

# *brief contents*

---

## **PART 1: STARTING WITH JQUERY**

### *1 Introducing jQuery*

## **PART 2: CORE JQUERY**

### *2 Selecting elements*

### *3 Operating on a jQuery collection*

### *4 Working with properties, attributes, and data*

### *5 Bringing pages to life with jQuery*

### *6 Events are where it happens!*

### *7 Demo: DVD discs locator*

### *8 Energizing pages with animations and effects*

### *9 Beyond the DOM with jQuery utility functions*

### *10 Talk to the server with Ajax*

### *11 Demo: An Ajax-powered contact form*

## **PART 3: ADVANCED TOPICS**

### *12 When jQuery is not enough...plugins to the rescue!*

### *13 Avoiding the callback hell with Deferred*

### *14 Unit Testing with QUnit*

### *15 How jQuery fits into large projects*

## **APPENDIXES:**

### *A JavaScript that you need to know but might not!*

# *Part 1*

## *Starting with jQuery*

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

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

## 1

# Introducing jQuery

## ***This chapter covers***

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

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

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

*jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of*

*versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.*

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

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

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

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

## 1.1 Write less, do more

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

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

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

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

```

        break;
    }
}

```

Contrast that with how it can be done using jQuery:

```

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

```

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

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

What is your answer? ☒ Yes ☐ No ☐ Maybe ☐ I dunno

Click me!

The radio element with value **yes** is checked.

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

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

When you perform a selection, you're relying on two things: a method and a selector. Today the latest versions of all major browsers support native methods for element selection like `document.querySelector()` and `document.querySelectorAll()`. They allow you to use more complex selectors instead of the usual selection by ID or class. In addition, the new CSS3 selectors are widely supported among modern browsers (see note). So, if you aimed to support only modern browsers, and the capabilities of jQuery lay only in selecting elements, you would have enough power to avoid the overhead introduced by the library in your website. The fact that a lot of people still rely on older browsers, which you may have to support, can be a real pain because you have to deal with all the inconsistencies. This is one of



the main reasons to employ jQuery. It allows you to reliably use its selectors without the worry of code not working in browsers that don't support them natively.

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

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

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

## 1.2 Unobtrusive JavaScript

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

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

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

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

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

### 1.2.1 Separating behavior from structure

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

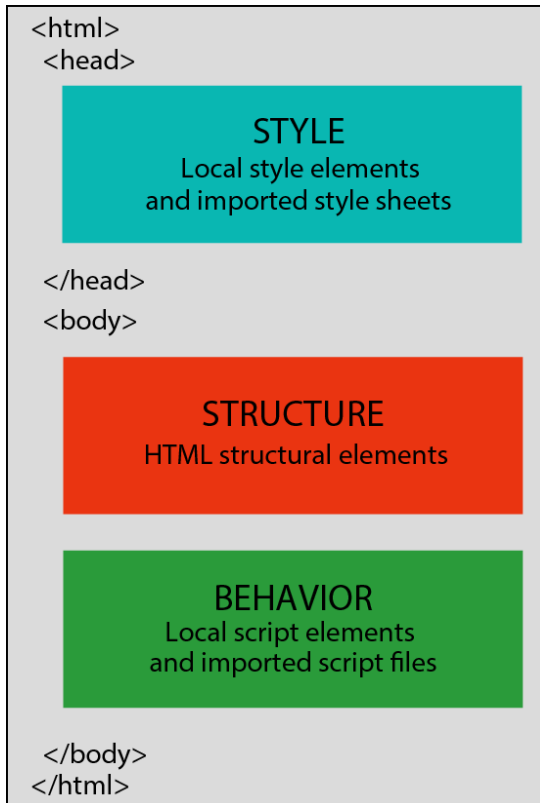


Figure 1.2 With structure, style, and behavior each neatly tucked away within a page, readability and maintainability are maximized.

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

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

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

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

### 1.2.2 Segregating the script

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

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

Placing the script at the bottom of the page, you don't need to use a handler attached to the `onload` event of the `window` object. At the time the browser parses this statement, the `button` element exists. So, you can safely augment it.

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

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

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

But none of that is bad; anything that persuades you to write your client-side code with the same level of care and respect usually allotted to server-side code is a good thing! But it is extra work—without jQuery, that is.

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

## 1.3 Installing jQuery

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

### 1.3.1 Choosing the right version

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

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

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

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

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

The first factor to base your decision on which version to use is what browsers your project must support. Table 1.1 describes the browsers supported by each major version of jQuery:

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

Browsers	jQuery 1	jQuery 2	jQuery Compat 3	jQuery 3
Internet Explorer	6+	9+	8+	9+
Chrome	Current and previous	Current and previous	Current and previous	Current and previous
Firefox	Current and previous	Current and previous	Current and previous	Current and previous
Safari	5.1+	5.1+	6.0+	6.0+
Opera	12.1x Current and previous	12.1x Current and previous	Current and previous	Current and previous
iOS	6.1+	6.1+	6.1+	6.1+
Android	2.3 4.0+	2.3 4.0+	2.3 4.0+	2.3 4.0+

As you can see from table 1.1, there's a certain degree of overlap in regard to the browser versions supported. But keep in mind that what's referred to as "Current and previous" (meaning the current and precedent version of a browser at the time a new version of jQuery is released) changes based on the release date of the new version of jQuery.

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

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

In summary, one of the factors is where you're going to use the library and what browsers you intend to support.

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

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

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

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


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

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

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

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

Uncompressed



```
ready:function(e){if(e===!0?!--x.readyWait:!x.isReady){if(!a.body)return
setTimeout(x.ready);x.isReady=!0,e!==!0&&--x.readyWait>0||(n.resolveWith(a,[x]),x.fn.trigge
r&&x(a).trigger("ready").off("ready"))}}
```

Compressed




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

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

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

### 1.3.2 Improving performances using a CDN

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

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

Let's say you want to include the compressed version of jQuery 1.11.1 using the jQuery CDN. You can do that by writing the following code:

```
<script src="//code.jquery.com/jquery-1.11.1.min.js"></script>
```

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

Using a CDN isn't all wine and roses, though. No server or network has 100% uptime on the internet, and CDNs are no exception. If you rely on a CDN to load jQuery, in the rare situations where it's down or not accessible and the visitor's browser doesn't have a cached copy, your website's code will stop working. For critical applications this can be a real problem. To avoid it, there's a simple and smart solution you can adopt, employed by a lot of developers. Once again, you want to include the minified version of jQuery 1.11.1, but now you'll use this smart solution:

```
<script src="//code.jquery.com/jquery-1.11.1.min.js"></script>
<script>window.jQuery || document.write('<script src="javascript/jquery-
1.11.1.min.js"></script>');</script>
```

The idea behind this code is to request a copy of the library from a CDN and check if it has been loaded, testing whether the `jQuery` property of the `window` object is defined. If the test fails, you inject a code that will load a local hosted copy that, in this specific example, is

stored in a folder called `javascript`. In case the `jQuery` property is present, you can use jQuery's methods safely without the need to load the local hosted copy.

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

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

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

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

## 1.4 How jQuery is structured

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

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

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

On the other hand, jQuery's source code follows the asynchronous module definition (AMD) format. The AMD format is a proposal for defining modules where both the module and its dependencies can be asynchronously loaded. In practice, this means that although you use jQuery as unique, single block, its source is split into several files (modules) as shown in figure 1.4. The dependencies relative to these files are managed through the use of a dependencies manager, in this case, RequireJS.



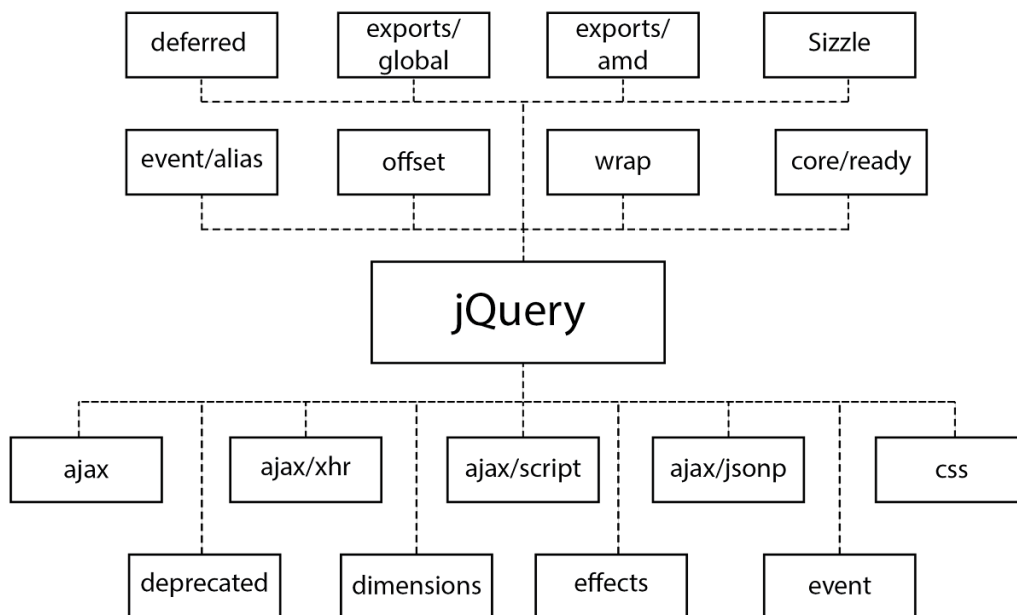


Figure 1.4 A schema representing jQuery's modules: `ajax`, `ajax/xhr`, `ajax/script`, `ajax/jsonp`, `css`, `deprecated`, `dimensions`, `effects`, `event`, `event/alias`, `offset`, `wrap`, `core/ready`, `deferred`, `exports/global`, `exports/amd`, and `Sizzle`

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

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

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

### 1.4.1 Save space creating your own custom build

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

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

To create a custom version you can use Grunt. Imagine that you need a minified version of jQuery 1.11.1 with all the functionalities except the deprecated functions and properties, and the effects. To perform this task, you need to install Node.js, Git, and Grunt on your local machine. After installing them, you have to clone jQuery's repository by running the following command using the command-line interface (CLI):

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

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

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

You're finished! Inside the folder named `dist` you'll find your custom jQuery build in both minified and not minified versions.

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

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

## 1.5 *jQuery fundamentals*

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

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

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

### 1.5.1 *Properties, utilities, and methods*

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

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

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

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

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

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

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

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

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

### 1.5.2 The jQuery object

The first function you'll use in your path to learn jQuery is `jQuery()`. It accepts up to two arguments and, depending on their number and type, performs different tasks. Like many other (almost all) methods in the library, it allows for *chaining*. Chaining is a programming technique used to call several methods in a single statement. So, instead of writing

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

you can write

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

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

When CSS was introduced to web technologies in order to separate design from content, a way was needed to refer to groups of page elements from external style sheets. The method developed was to use selectors, which concisely represent elements based on their type, attributes, or position within the HTML document. Those familiar with XML might be familiar with XPath (more on this here: <http://www.w3.org/TR/xpath20/>) as a means to select elements within an XML document. CSS selectors represent an equally powerful concept but

are tuned for use within HTML pages, are a bit more concise, and are generally considered easier to understand.

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

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

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

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

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

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

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

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

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

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

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

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

extensively when you need to perform operations, like running a certain animation or applying a style, on several elements in the page.

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

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

### 1.5.3 The document ready handler

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

In the radio group example, the entire body has to be loaded before the behavior can be applied. Traditionally, the `onload` handler for the `window` instance is used for this purpose, executing statements after the entire page is fully loaded. The syntax is typically something like this:

```

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

```

This causes the defined code to execute *after* the document has fully loaded. Unfortunately, the browser not only delays executing the `onload` code until after the DOM tree is created but also waits until all external resources are fully loaded and the page is displayed in the browser window. This includes resources like images as well as QuickTime and Flash videos embedded in web pages. As a result, visitors can experience a serious delay between the time that they first see the page and the time that the `onload` script is executed.

Even worse, if an image or other resource takes significant time to load, visitors will have to wait for the image loading to complete before the rich behaviors become available. This could make the whole unobtrusive JavaScript proposition a non-starter for many real-life cases.

A much better approach would be to wait *only* until the document structure is fully parsed and the browser has converted the HTML into its resulting DOM tree before executing the script to apply the rich behaviors. Accomplishing this in a cross-browser manner that takes into account older browsers is somewhat difficult, but jQuery provides a simple means to trigger the execution of code once the DOM tree has loaded (without waiting for external resources).

The formal syntax to define such code is as follows:

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

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

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

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

By passing a function to `jQuery()` or its alias `$()`, you instruct the browser to wait until the DOM has fully loaded (but only the DOM) before executing the code. Even better, you can use this technique multiple times within the same HTML document, and the browser will execute all of the functions you specify in the order in which they're declared within the page.

In contrast, the window's `onload` technique allows for only a single function. This limitation can also result in hard-to-find bugs if any included third-party code uses the `onload` mechanism for its own purpose (not a best-practice approach).

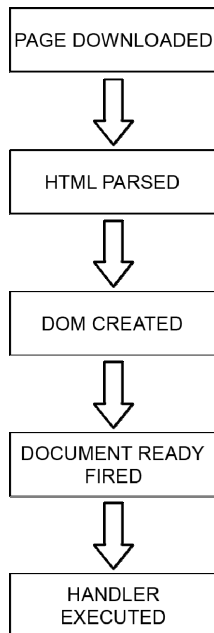


Figure 1.5 A representation of the steps performed by browsers before the document-ready handler is executed

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

Because `ready()` takes care to execute the code after the DOM is loaded, developers used to place the `script` elements in the `<head>` of the page. As we discussed in section 1.1.2, "Segregating the script," you can place them just before the closing body tag (`</body>`). Doing so, you can completely avoid the use of `$(document).ready()` because, at that point, all of the other elements are already in the DOM. Therefore, you can retrieve and use them safely. If you want to see an example of how `$(document).ready()` can be avoided, take a look at the source code of the file `chapter-1/radio.group.html`.

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

## 1.6 Summary

We've covered a great deal of material in this whirlwind introduction to jQuery. But to summarize, it's generally useful for any page that needs to perform anything but the most trivial of JavaScript operations. It's also strongly focused on enabling page authors to employ the concept of unobtrusive JavaScript within their pages. With this approach, behavior is separated from structure in the same way that CSS separates style from structure, achieving better page organization and increased code versatility.

Despite the fact that jQuery introduces only two new names in the JavaScript namespace—the self-named `jQuery` function and its `$` alias—the library provides a great deal of functionality by making that function highly versatile, adjusting the operation that it performs based on the parameters passed to it.

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

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