PART 1

The jQuery API

1 Introduction to jQuery

JavaScript frameworks have arisen as necessary and useful companions for client-side web development. Just a few years ago, JavaScript frameworks were needed to pave over the many inconsistencies present with cross-platform web development. Before Microsoft got its act together and gave us IE with vastly improved standards support, there was more often than not the IE way and the standard way. Frameworks like jQuery helped immensely to fill in the holes between standard and nonstandard. Today jQuery is a phenomenally popular, leading JavaScript framework and application development platform. It is leaner; it is faster loading; and it comes loaded with features that make the life of a JavaScript application developer much easier. No longer is JavaScript an afterthought, grafted onto stateless HTML. It is used more and more to be the foundation and the primary driving force of not only web development but also application development, from desktop to tablets and smartphones.

Thanks to renewed vigor in the browser and platform wars of the big tech giants, JavaScript has also become much leaner and faster. Today, the leading browser makers are delivering JavaScript capabilities that take the good ole reliable, interpreted language of JavaScript and instantly transform it into cached machine byte code that can be executed blazingly fast. Because of the collective advances and one-upmanship of Apple, Google, Mozilla, and Microsoft, today we have JavaScript that has never performed better.

When this book was first written in 2009, jQuery was emerging as the de facto standard JavaScript framework and application platform. Today jQuery sits atop the heap as a global leader facilitating cutting-edge web and application development from mom-and-pop shops to Fortune 500 companies. It is baked into iOS and Android apps and mobile websites both with and without the popular jQuery Mobile framework add-on, and it runs the websites of some of the world's biggest companies, such as Amazon, Apple, *The New York Times*, Google, BBC, Twitter, and IBM.

For years JavaScript frameworks have paved over the craters and inconsistencies of cross-browser web development to create a seamless, enjoyable client-side programming experience. Today, with Internet Explorer 11 and its underlying Trident engine, Microsoft finally has a world-class standards-compliant web browser that's caught up with competing offerings from Apple's Safari and world-leading, underlying, open-source WebKit, Google's Chrome browser and newly forked from WebKit Blink engine, and Mozilla's Firefox powered by the Gecko engine. Web developers have never had better platforms on which to build modern, fully standards-compliant applications.

One of jQuery's biggest innovations was its fantastic DOM querying tool using familiar CSS selector syntax. This component, now called Sizzle, is now a separate open-source component included within the larger open-source jQuery framework. It contains jQuery's added on CSS pseudo-class selectors and the full DOM querying CSS selector engine that works in browsers as old as IE6 as well as new browsers. It uses the native JavaScript document, `querySelectorAll()` function call, which makes DOM queries using CSS selectors fast, when it is available. Sizzle is one of the biggest driving forces that makes jQuery web development super easy and has thus attracted a large number of developers to the jQuery world.

Another feature that makes jQuery web development very easy and attractive is its support for chained method calls. Where the API supports it, you can call one method after another by chaining method calls on the backs of one another. This is what a chained method call looks like using jQuery:

```
$('<div/>')
    .addClass('selected')
    .attr({
        id : 'body',
        title : 'Welcome to jQuery'
    })
    .text("Hello, World!");
```

In the preceding example, a `<div>` element is created with jQuery. jQuery is contained within the dollar sign variable, `$`, which is a JavaScript variable just one character long. This variable contains the entire jQuery framework and is the starting point for everything that you can do with jQuery. The statement `$('<div/>')` creates the `<div>` element, and then you see multiple method calls following that statement. `.addClass('selected')` adds the class attribute to the `<div>` element. Then there is a call to `.attr()`, which adds two additional attributes to the `<div>` element, an `id` attribute and a `title` attribute, and then the call to `.text()` fills the `<div>` element with plain text content. With this little snippet of code, you have four separate method calls all strung together to form a single expression spanning multiple lines. This brief sample of what jQuery can do results in the creation of a `<div>` element that can be inserted into the DOM that looks like this:

```
        <div class="selected" id="body" title="Welcome to jQuery">Hello, World</
div>
```

jQuery packs a powerful punch; it helps you develop better JavaScript applications by facilitating powerful DOM interaction and manipulation with less code than you would use with a pure JavaScript approach. This is what is meant by jQuery's motto, "Write less, do more." Compare the snippet of jQuery that I presented with the following, which creates the same `<div>` element with pure JavaScript:

```
var div = document.createElement('div');

div.className = 'selected';
div.id = 'body';
```

```
    div.title = 'Welcome to jQuery';

    var text = document.createTextNode ("Hello, World!");

    div.appendChild(div);
```

As you can see, jQuery is much less verbose. It wraps around traditional, native JavaScript APIs to help you as a developer get more done with JavaScript using less code, allowing application development to go more quickly.

In this chapter I present the following information:

- What jQuery can do for you
- Who develops jQuery?
- Where and how to get jQuery
- How to install and use jQuery for the first time
- XHTML and CSS programming conventions
- JavaScript programming conventions

WHAT JQUERY CAN DO FOR YOU

As presented in the last section, jQuery makes many tasks easier. Its simplistic, chainable, and comprehensive API has the capability to completely change the way you write JavaScript. With the goals of doing more with less code, jQuery really shines in the following areas:

- jQuery makes iterating and traversing the DOM much easier via its various built-in methods.
- jQuery makes selecting items from the DOM easier via its sophisticated, built-in, and ubiquitous capability to use selectors, just like you would use in CSS.
- jQuery makes it easy to add your own custom methods via its simple-to-understand plug-in architecture.
- jQuery helps reduce redundancy in navigation and UI functionality, like tabs, CSS, and markup-based pop-up dialogs, animations, and transitions, and lots of other things.

Is jQuery the only JavaScript framework? No, certainly not. You can pick from several JavaScript frameworks: Yahoo UI, Prototype, SproutCore, Dojo, and so on. I like jQuery because I enjoy its simplicity and lack of verbosity. However, among the other frameworks, you'll find that there is a lot of similarity, and each provides its own advantages in terms of unifying Event APIs, providing sophisticated selector and traversal implementations, and providing simple interfaces for redundant JavaScript-driven UI tasks. Across the entire web, including websites that don't use any JavaScript frameworks, jQuery can be found on as many as half of all websites. So, jQuery definitely has the benefit of a ubiquitous, de facto standard. Based on its popularity, you're extremely likely to run into other developers who have experience with and know how to use jQuery.

Another aspect of jQuery programming I enjoy is that jQuery doesn't seek to impose its own opinions about programming onto you, its user. Some frameworks, ExtJS in particular, seek to completely circumvent traditional JavaScript, HTML, and CSS web development with complicated Model–View–Controller (MVC) implementations that seek to auto-generate the HTML and CSS portions for you, which is perfectly fine, if that's how you like to program. jQuery does not impose any kind of programming paradigm on you, the user. Combined with tools that are designed to work well with it, such as Mustache.js and Backbone.js, a more reasonable programming paradigm can be achieved alongside the popular MVC-based programming pattern, in which you still have control over the HTML and CSS that you create and use.

In past years, what initially drew developers to jQuery-based web development was its incredibly simple way of erasing the lines between browsers. It presented a unified API for event handling, whereas before JavaScript frameworks came along, you had a clumsy fragmented approach in which Microsoft had one way for IE and the other browsers had the standard way. jQuery made cross-browser web development easy and seamless. Today, developers continue to flock to jQuery, no longer because of fragmentation because those issues have been slowly resolved over the last four years, but simply because it is leaner and easier to understand and use than native JavaScript programming. Finally, Microsoft has implemented the standard event-handling model in Internet Explorer that everyone else has been using for more than a decade. The latest version of jQuery, version 2.0, sheds the legacy baggage that facilitated that cross-browser web development, allowing jQuery to become leaner and faster.

That's not to say that there are no more cross-browser issues. There are still areas of JavaScript development in which there are multiple approaches. Thankfully, these areas are becoming fewer and fewer. Cross-browser issues exist more today in cutting-edge CSS where browsers make brand-new, experimental features available using vendor-prefixed CSS properties. One of the most frustrating examples is that of gradients in CSS, where to implement the feature correctly in modern and legacy browsers, you have up to seven different ways of writing the same gradient:

- WebKit's extremely syntactically verbose first stab at CSS gradients: `-webkit-gradient`
- WebKit's implementation of the revised standard: `-webkit-linear-gradient` and `-webkit-radial-gradient`
- The current W3C CSS3 standard: `linear-gradient` and `radial-gradient`
- Microsoft's vendor-prefixed standards-compliant implementation of the W3C standard: `-ms-linear-gradient` and `-ms-radial-gradient`
- Microsoft's proprietary implementation of gradients found in the old `filter` and `-ms-filter` properties
- Mozilla's implementation: `-moz-linear-gradient` and `-moz-radial-gradient`
- Opera's implementation previous to their adoption of Google's WebKit fork (now Blink) engine: `-o-linear-gradient` and `–o-radial-gradient`

As you can see, the situation with using cutting-edge CSS that is still working its way through the standardization process is not pleasant for web developers. Unfortunately, rather than adopting a comprehensive approach, most web developers stop with the `-webkit-` variants, and they don't bother implementing the variants supported by other browsers. This, in part, is what persuaded Opera to discontinue development of its own Presto engine in favor of Google's Blink fork of WebKit. This used to be the kind of thing developers needed a framework like jQuery to solve. This particular situation is solved, by the way, by using server-side dynamically generated CSS template solutions, or even a client-side jQuery plugin.

The beauty of jQuery is that it can solve problems like vendor-specific CSS gradients as well as the remaining cross-browser issues that exist in JavaScript through its comprehensive and easy-to-use plugin ecosystem. Several great third-party jQuery plugins are presented later in this book.

WHO DEVELOPS JQUERY?

I won't spend a lot of time talking about the history of JavaScript frameworks, why they exist, and so on. I prefer to get straight to the point. That said, a brief mention of the people involved with developing jQuery is in order.

jQuery's original creator is John Resig, whose website is located at `www.ejohn.org`. John resides in Brooklyn, New York, and is presently the Dean of Computer Science at Khan Academy. John still helps with defining the direction and goals of the jQuery project, but jQuery has largely been transitioned to a large team of people. You can learn more about these people and what roles they played in jQuery's development at `https://jquery.org/team/`.

OBTAINING JQUERY

jQuery is a free, Open Source JavaScript Framework. The current stable, production release version, as of this writing, is 1.10.2 and 2.0.3. The difference in these two versions of jQuery largely revolve around legacy browser support; the 2.0 release of jQuery dispenses with the huge amount of legacy baggage it needed to facilitate support with older versions of Internet Explorer.

I use version 1.10.2 throughout the course of this book, for maximum browser compatibility. Getting jQuery is extremely easy—all you have to do is go to `www.jquery.com` and click the Download jQuery link. You'll see two options for downloading either the 1.x version or the 2.x version:

- A compressed production version
- An uncompressed development version

The uncompressed development version is recommended for use while you are developing. This version can facilitate doing back traces with web developer tools in any of the major browsers. You can walk through the JavaScript chain of execution and see what code is executing in nice, human-readable code. The compressed production version is recommended for use on production websites where size is a huge consideration; the file is compressed to remove all the extra whitespace so that it downloads quickly.

INSTALLING JQUERY

Throughout this book, I will refer to the jQuery script as though it is installed at the following path: `www.example.com/jQuery/jQuery.js`.

Therefore, if I were using the domain `example.com`, jQuery would have this path from the document root, `/jQuery/jQuery.js`. You do not have to install jQuery at this exact path. You can move jQuery wherever you like, but don't forget to update the path.

HELLO, WORLD IN JQUERY

In the following example you learn how to install jQuery and execute a remedial "Hello, World" jQuery-based JavaScript application. To start, follow these steps:

1. Download the jQuery script from `www.jquery.com`. Alternatively, I have also provided the jQuery script in this book's source code download materials available for free from `www.wrox.com/go/webdevwithjquery`.

2. Enter the following XHTML document, and save the document as **Example** 1-1.html. Adjust your path to jQuery appropriately; the path that I use reflects the path needed for the example to work when opened in a browser via the source code materials download made available for this book.

```
<!DOCTYPE HTML>
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="content-type"
            content="application/xhtml+xml; charset=utf-8" />
        <meta http-equiv="content-language" content="en-us" />
        <title>Hello, World</title>
        <script type='text/javascript' src='../jQuery.js'></
script>
        <script type='text/javascript' src='Example 1-1.js'>
             </script>
        <link type='text/css' href='Example 1-1.css'
            rel='stylesheet' />
    </head>
    <body>

    </body>
</html>
```

3. Enter the following JavaScript document, and save the document as **Example 1-1.js:**

```
$(document).ready(
    function()
    {
        $('body').append(
            $('<div/>')
                .addClass('selected')
                .attr({
                    id : 'body',
                    title : 'Welcome to jQuery'
                })
                .text(
                    "Hello, World!"
                )
            );
        }
    );
```

4. Enter the following CSS document, and save the document as **Example 1-1.css:**

```
body {
       margin: 0;
       padding: 20px;
       font: 14px Helvetica, Arial, sans-serif;
}
div.selected {
       background: blue;
       color: white;
       padding: 5px;
       display: inline-block;
}
```

The preceding code results in the screen shot that you see in Figure 1-1 if the installation were unsuccessful. If installation were not successful, the page appears blank.

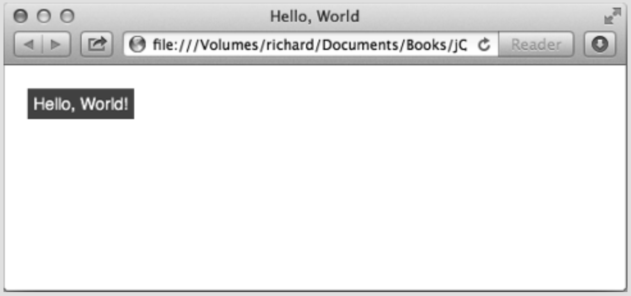

FIGURE 1-1

In the preceding example, you installed and tested your installation of the jQuery framework. The JavaScript that you included is executed when the document's `onready` event is fired, which is executed as soon as the DOM is fully loaded: all markup, JavaScript, and CSS, but not images. The callback function attached to the `onready` event then creates a `<div>` element with the *selected* class name and contains the text *Hello, World!*

You have now used jQuery for the first time.

PROGRAMMING CONVENTIONS

In web development, it's common for professional software engineers, web designers, and web developers—and anyone with a job title whose day-to-day activities encompass the maintenance of source code—to adopt standards and conventions with regard to how the source code is written. Standardization bodies like the W3C, which define the languages that you use to create websites, already decide on some standards for you. Some standards are not written but are rather de facto standards. De facto standards are standards that have become accepted throughout the industry, despite not appearing in any official document developed by a standards organization.

Throughout this book, I talk about standards, de facto and official, and how to develop and design web-based documents and even web-based applications that take those standards into account. For example, I talk extensively about how to separate behavior (JavaScript) from presentation (CSS) and structure (XHTML). JavaScript written in this way is commonly referred to as *nonintrusive* JavaScript—it's nonintrusive because it supplements the content of a web document, and, were it turned off, the document would still be functional. CSS is used to handle all the presentational aspects of the document. And the structure of the document lives in semantically written XHTML. XHTML that is *semantically written* is organized meaningfully with the right markup elements and contains little, if any, presentational components directly in the markup.

In addition to standards, I discuss how to develop web-based documents, taking into account different browser inconsistencies, discrepancies, and idiosyncrasies. There is some interactive functionality that nearly every browser handles differently; in those situations, other web professionals have already pioneered de facto standards that are used to bring all browsers into accord. The idea of a JavaScript foundational framework has become more popular and increasingly a dependency for HTML5 applications, like the ones you'll learn to develop using the jQuery framework.

Before I begin the discussion of how to use jQuery, the coming sections provide a generalized overview of programming conventions and good practice that you should follow.

Markup and CSS Conventions

It's important that your web documents be well organized, cleanly written, and appropriately named and stored. This requires discipline and even an obsessive attention to the tiniest of details.

The following is a list of rules to abide by when creating XHTML and CSS documents:

- When selecting id and class names, make sure that they are descriptive and are contained in a namespace. You never know when you might need to combine one project with another—namespaces help you to prevent conflicts.
- When defining CSS, avoid using generic type selectors. Make your CSS more specific. This can also help with preventing conflicts.
- Organize your files in a coherent manner. Group files from the same project in the same folder; separate multiple projects with multiple folders. Avoid creating huge file dumps that make it difficult to locate and associate files.

- Avoid inaccessible markup. Stay away from frames, where possible. Organize your markup using semantically appropriate elements. Place paragraphs in `<p>` elements. Place lists in `<ul>` or `<ol>` elements. Use `<h1>` through `<h6>` for headings, and so on.
- If you can, also consider the loading efficiency of your documents. For development, use small, modularized files organized by the component; combine and compress those modularized files for a live production site.

Id and Class Naming Conventions

Most web developers don't think too much about the topics of namespacing and naming conventions. Naming conventions are just as important in your markup id and class names as namespacing is important in programming languages.

First, what is namespacing, and why do you need to do it? *Namespacing* is the concept of making your programs, source code, and so on tailored to a particular naming convention, in an effort to make your programs more portable and more capable of living in diverse, foreign programming environments. In other words, if you want to directly insert a web application into your document, you want to be sure that the class and id names, style sheets and script, and all the bits that make your web application what it is do not conflict with any applications that are already present in the document. Your applications should be fully self-contained and self-sufficient and not collide or conflict with any elements already present in a document.

What are some common id names that people use in style sheets? Think first about what the typical components of a web application are. There's a body. There may be one or more columns. There may be a header and a footer, and there are lots of components that can potentially be identified as generic, redundant pieces that all web applications may have. Then, it stands to reason that plenty of websites are probably using id and class names like *body*, *header*, *footer*, *column*, *left*, *right*, and so on. If you name an element with the id or class name *body*, you have a good chance of conflicting with an overwhelming majority of websites in existence today. To avoid this type of conflict, it's considered good practice to prefix id and class names within a web application to avoid conflicts and namespace collisions. If you write an application called *tagger*, you might namespace that application by prefixing all your id and class names with the word *tagger*. For example, you might have *taggerBody*, *taggerHeader*, *taggerFooter*, and so on. It may be possible, however, that someone has already written an application called *tagger*. To be safe, you might do a web search on the name you've chosen for your application to make sure that no one's already using that name. Typically, simply prefixing your id and class names with your application's name is enough.

In addition, it also helps to prefix id and class names with type selectors in style sheets. *Type* selectors help you narrow down what to look for when modifying or maintaining a document. For example, the id selector `#thisId` is ambiguous. You don't know what kind of element `thisId` is, and thus would likely have to scan the entire document to find it. But `div#thisId` is more specific. By including the `div` in the selector, you instantly know you're looking for a `<div>` element. Including the type in the selector also helps you in another way: When dealing with class names, you can have the same class name applied to different types of elements. Although I may not condone that as good practice, at least in the style sheet, you can control which element gets which style. `span.someClass` and `div.someClass` are selectors that differentiate style based on the type of element, whereas `.someClass` is more ambiguous and applies to any element.

Id and class names should also be descriptive of their purpose in a semantically meaningful way. Keep in mind that an id name can potentially be used in a URL as an HTML anchor. Which is better: `www.example.com/index.html#left` or `www.example.com/index.html#exampleRelatedDocuments`? The latter id anchor is namespaced `example` for *example.com*, and `RelatedDocuments` is the name of the element; thus, the latter URL includes more information about what purpose the element serves and greatly increases the maintainability of the document in a very intuitive way. In addition, the latter has more benefit in terms of search engine optimization (SEO). The former is too ambiguous and won't provide much in the way of SEO. Think of each of your id and class names as though it is part of the URL of your document. Give each id and class name that you create semantic names that convey meaning and purpose.

Generic Type Selectors

Generic type selectors are style-sheet rules that look something like this:

```
a {
   color: #29629E;
}
```

In the preceding style-sheet rule, you see what's probably a pretty common scenario, changing the color of every link in a document via a generic type selector that refers to all `<a>` elements. Generic type selectors should be avoided for the same reason that it is good to namespace id and class names within a document, avoiding conflicts when multiple scripts or style sheets are combined in the same document. Instead, it's best practice to apply id or class names to these elements, or at the very least place them in a container that has an id or class name, and only use descendant selectors when referencing those elements via a style sheet.

```
div#exampleBanner a {
   color: #29629E;
}
```

The preceding example avoids the pitfalls introduced by using a blanket, generic selector style-sheet rule by limiting the scope of the style-sheet rule's application. Now, only `<a>` elements that are descendants of a `<div>` with the id name *exampleBanner* receive the declaration `color: #29629E;`.

Storing and Organizing Files

How files are organized and stored is important to the maintainability of a document. You should maintain your documents in an easy-to-understand, easy-to-learn directory hierarchy. Different people have different approaches to storing and organizing files, obviously. What matters is that there is an organization scheme, rather than none at all. Some choose to store documents by type and then separate them by application, whereas others prefer to separate by application first and then sort by type.

Avoid Making Documents Inaccessible

Accessibility is also an important factor to consider in the design of a web document. You should do your best to make your JavaScript nonintrusive, but also avoid taking away a document's accessibility by either script or markup.

- Avoid using frames.
- Limit the number of images to those that actually contribute to the content of a document. With the CSS3, Data URIs and SVG standards, much more of what used to be required image content for the design of a site no longer has to be included in images and can be programmed with either CSS3 or SVG (for example, gradients or inner or drop shadows). When you have to use images, try to contain as much of the design as possible in CSS background images. Make available double-resolution images for retina or high-resolution devices. And keep images that directly contribute to the content in `<img />` elements. Be sure to include `alt` attributes that describe the image for each `<img />` element.
- Place content in semantically appropriate markup containers—use `<p>` for paragraphs, `<h1>` through `<h6>` for headings. Use the new HTML5/XHTML5 elements designed to make semantic content more semantic: `<heading>`, `<article>`, `<aside>`, `<summary>`, to name just a few.
- Make the design high contrast when possible. Imagine what the document would look like through the eyes of someone with poor vision. Can you easily read the content?
- Avoid wandering too far away from established user-interface conventions. Can you distinguish hyperlinks from normal content?
- Consider making the content keyboard-accessible. Can you navigate without a pointing device?
- Make the content more unobtrusive. Can you use the website without Flash and JavaScript functionality? JavaScript and Flash should enhance web content in a complementary way, not be a requirement.
- Avoid placing a large number of links at the beginning of every document. If you were listening to the content being read to you, rather than seeing it visually, would the experience be enjoyable?

Accessibility should be practiced to the point of becoming an automatic reflex. It should be cemented in your development practices in a fundamental way in the same way that namespacing and file organization are. Although other best practices can become second nature easily, it's also easy to get into the habit of ignoring accessibility, so a conscious effort must be made to periodically review accessibility and ingrain accessibility in the development process.

Efficiency in Markup and CSS

Markup and CSS in a complex website can easily become large and bloated and drag down overall loading and execution times more and more. This can become particularly troublesome as the overall popularity of a site increases. As the complexity of a website increases, it becomes necessary to look into ways of streamlining the content. It's best to limit the number of external files being loaded, but all CSS and JavaScript should be included in at least one external file. Were JavaScript and CSS included directly in a document, the initial loading time would improve, but you'd also lose the advantage of separately caching JavaScript and CSS on the client side.

For the best of the best in efficiency, combine the following concepts:

- Server-side gzip compression. You should test your website with this feature enabled and disabled because it has some trade-offs involved. See if gzip compression is right for you. In my experience gzip may make files load more quickly, but it can also delay when you see content because it prevents incremental rendering from occurring. It is usually more important that your users see content as quickly as possible.
- Aggressive client-side caching; this makes subsequent page loads much faster.
- Automatic compression of markup content by removing excess whitespace and comments from the markup source.
- Automatic compression and consolidation of multiple CSS and JavaScript files by removing all excess whitespace and comments from each file. Appropriately combining files further decreases load times by reducing HTTP latency.

When the preceding items are combined, you make the loading times of a web document the best possible; however, there are some caveats to consider that may at first seem contradictory:

- Maintainable markup should be written in a neat and organized manner. It should be well spaced and indented and contain line breaks where appropriate.
- Good programming practice means modularized development, so break up your CSS and JavaScript by component and application. Make small, easy-to-digest chunks. This will speed up your ability to maintain and extend projects.
- Client-side caching can lead to headaches when updates are made to CSS or script files. Browsers will continue to use the old version of the CSS and script files after an update is made, when caching is working correctly.

The good news is that all the preceding caveats can be overcome. The bad news is that it's not particularly easy to overcome them.

The best way to implement efficiency in markup, JavaScript, and CSS documents is to make the efficiency automatic. That is to say, write server-side applications that handle efficiency tasks for you. A well-designed, professional content management system can work out those bits for you. It can allow you to make your JavaScript, markup, and CSS documents modularized and separate them based on the task each is designed to perform but automatically combine and compress those documents for you.

Unfortunately, not everyone can use a professional content management system to serve their content. For those individuals, there are some compromises to be made:

- JavaScript and CSS can be hand-compressed using a web-based utility like Dean Edwards's packer, `http://dean.edwards.name/packer`. Development can continue to be modularized, and the compression and consolidation portion of development simply becomes a manual task.

- You can limit the amount of whitespace you use in a document. Indent content with two spaces instead of four.

Overcoming the headaches with document caching, however, is a much easier task. You can force a browser to update a document by changing its path. For example, say you have the following script included in your markup:

```
<script src='/script/my.js' type='text/javascript'></script>
```

You change the path from `/script/my.js` to `/script/my.js?lastModified=09/16/07`. The latter references the same, `my.js`, but is technically a different path to the browser, and the browser, consequently, will force refreshing of its cached copy of the document. The *?lastModified=09/16/07* portion of the path is the *query string* portion of the path. The query string begins with a question mark and then contains one or more query string variables. Query string variables are used by a server-side programming language or client-side JavaScript to pass information from one document to another. In this example, there is no information being passed per se. You're including the time of the last modification, although I could have just as easily included the revision, or even a random string of characters. The inclusion of a query string in this example has only one purpose: to force the browser into refreshing the cached version of the document.

The same can be done with CSS:

```
 <link type='text/css' rel='stylesheet' href='/styles/
my.css?lastModified=09/16/07' />
```

In the preceding snippet of markup that includes an external CSS document, the query string is used to force a refresh of the browser's cached copy of the style sheet `my.css`.

The next section talks about some conventions specific to JavaScript.

JavaScript Conventions

In JavaScript, several things should be considered bad practice and avoided:

- Include all script in external documents—JavaScript code should be included only in external script files. Script should not be embedded in markup documents or be included inline, directly on markup elements.
- Write clean, consistent code—JavaScript code should be neatly formatted and organized in a consistent, predicable way.
- Namespace JavaScript code—JavaScript variables, functions, objects, and the like should be namespaced to minimize potential namespace conflicts and collisions with other JavaScript applications.
- Avoid browser detection—Browser detection should be avoided where possible. Instead, detect specific browser features.

The next sections present cursory, generalized overviews of each of the preceding concepts.

Include All Script in External Documents

Part of making JavaScript nonobtrusive means making JavaScript complementary and supplemental, rather than required and mandatory. This concept is explored in detail throughout this book; however, it should be noted why this is the best approach.

Consider the following code example:

```
<!DOCTYPE HTML>
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="content-type"
            content="application/xhtml+xml; charset=utf-8" />
        <meta http-equiv="content-language" content="en-us" />
        <title>Hello, World</title>
 <link type='text/css' href='Example 1-2.css' rel='stylesheet' />
    </head>
    <body>
        <p>
             <img src="pumpkin.jpg" alt="Pumpkin" />
             <a href="javascript:void(0);"
                onclick="window.open(
                    'pumpkin.jpg',
                    'picture',
                    'scrollbars=no,width=300,height=280,resizable=yes');">
                 Open Picture
             </a>
        </p>
    </body>
</html>
```

Combine the preceding markup with the following style sheet:

```
img {
    display: block;
    margin: 10px auto;
    width: 100px;
    border: 1px solid rgb(128, 128, 128);
}
body {
    font: 14px sans-serif;
}
p {
    width: 150px;
    text-align: center;
}
```

The preceding code gives you something like what you see in Figure 1-2.

In Figure 1-2, you see what is probably a common scenario: You have a thumbnail, and you can click to see a bigger version of the thumbnail. This is the kind of thing that JavaScript works well for—giving you the bigger version in a separate pop-up window that doesn't have any controls.

Now examine why what I did in Figure 1-2 was the wrong way to go about adding this functionality.

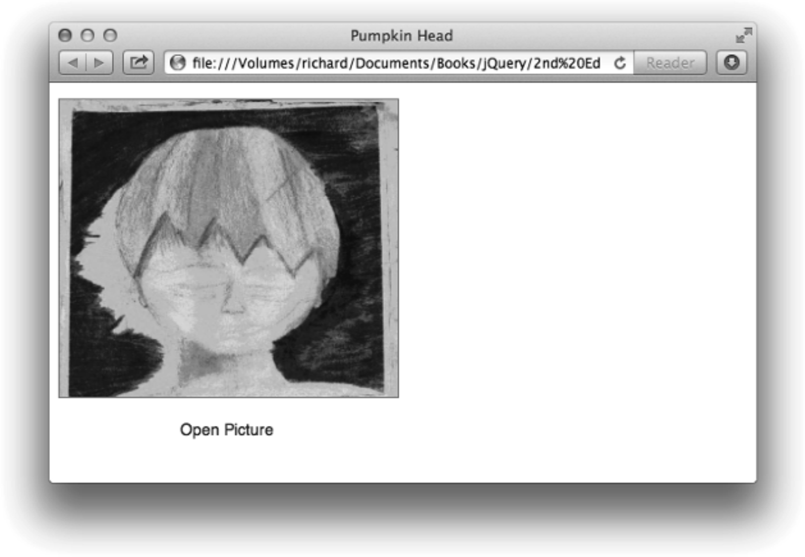

FIGURE 1-2

Here are the problems with this approach:

- If JavaScript is disabled, viewing the larger picture doesn't work.
  - JavaScript can be disabled out of personal preference.
  - JavaScript can be disabled because of company policy.
- Placing the JavaScript directly in the markup document adds unnecessary bloat and complexity to the markup document.

The overwhelming point in all this is that inline JavaScript is a bad way to approach adding complementary, interactive functionality to a web document.

Here is a better approach to the application presented in Figure 1-2. First, take the inline JavaScript out of the markup and replace it with a reference to an externally loaded JavaScript. The following example names the externally loaded JavaScript `Example 1-3.js`:

```
<!DOCTYPE HTML>
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="content-type"
            content="application/xhtml+xml; charset=utf-8" />
        <meta http-equiv="content-language" content="en-us" />
        <title>Pumpkin Head</title>
        <script type='text/javascript' src='../jQuery.js'></script>
        <script type='text/javascript' src='Example 1-3.js'></script>
        <link type='text/css' href='Example 1-3.css' rel='stylesheet' />
    </head>
    <body>
        <p>
```

```
                <img src="pumpkin.jpg" alt="Pumpkin" />
                <a href="pumpkin.jpg" id="examplePumpkin" target="_blank">
                    Open Picture
                </a>
            </p>
        </body>
    </html>
```

Then in the externally loaded JavaScript you do something like the following:

```
    $(document).ready(
        function()
        {
            $('a#examplePumpkin').click(
                function(event)
                {
                    event.preventDefault();

                    window.open(
                        'pumpkin.jpg',
                        'Pumpkin',
                        'scrollbars=no,width=300,height=280,resizable=yes'
                    );
                }
            );

        }
    );
```

With the preceding bits of code, you get the results that you see in Figure 1-3.

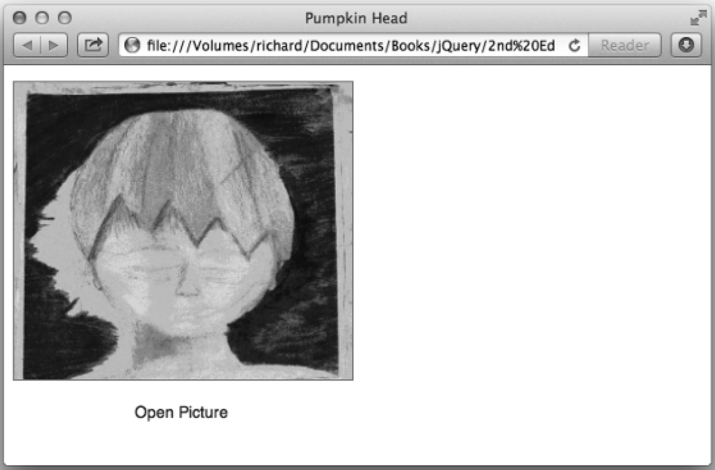

FIGURE 1-3

This is an example of nonobtrusive JavaScript. Nonobtrusive JavaScript provides extended, interactive functionality within a web document, but does not do so in a way that obstructs using the document in a plain-vanilla manner. That is to say, with JavaScript disabled, you can still use the website and get what you need from it.

In the preceding example, the JavaScript is moved to an external document called *Example 1-3.js*. Within *Example 1-3.js* jQuery is used to call upon an `<a>` element with the id name *examplePumpkin*, and this in turn opens the pop-up window. If JavaScript is disabled, the picture is still opened in another window, but if JavaScript is disabled, you just can't control the size of the window or whether it has controls.

So far, the user clicks an `<a>` element and gets a pop-up window. You want the window to pop up instead of initiating the default action that occurs when a user clicks a link, which instead of doing nothing, is now for the browser to navigate to the document defined in the `href` attribute of the `<a>` element, also in a new window. This default action is prevented with the call to `event.preventDefault()`.

In this simple example, you've seen how a simple example can become something a little more complex, but not much more complex. With a little further thought and attention to detail, a simple enhancement can continue to function if the user has disabled script in his browser.

Write Clean, Consistent Code

It's important to follow some predetermined criteria for producing clean, consistent, well-organized code. In the professional world, most programmers have a particular way they like to see their code formatted. Earlier in this section, I talked about how indenting and spacing markup and CSS documents can help you more easily catch errors in those documents and make those documents more maintainable. Well, the same can be applied to JavaScript. Here I talk about each of the programming conventions that I follow for writing JavaScript source code.

Indenting and Line Length

It's a good idea to indent your code so that it's easier to read and maintain. Take the following, for example:

```
window.onload=function(){var nodes=document.getElementsByTagName('a');
for(var i = 0,length=nodes.length;i<length;i++){nodes[i].onclick=
function(event){window.open(this.href,"picture",
"scrollbars=no,width=300,height=280,resizable=yes");
event? event.preventDefault():(window.event.returnValue=false);};}};
```

In the preceding block of code, you see the contents of *Example 1-3.js* presented above in this section, formatted without any indenting or spacing. Now, imagine that the preceding code is 10,000 lines of code spread out over many files, all formatted the same way. It's not a bad idea to reduce spacing for a live, production script; in fact, many professionals use compression routines specifically for this. But those same professionals don't maintain their scripts in the compressed format and often have a rigid programming standard to which every script they produce must conform.

A common, fairly universal programming standard is setting the size of an indentation to four spaces, although some use just two spaces or other values. This is in addition to setting a blanket rule that tabs cannot be used in place of individual spaces, even though, technically, a tab character results in less bytes added to a file when compared to four individual space characters. The "no tab" rule exists because of the wide variance in the interpretation of what a tab character is in text applications. Some text applications say that a tab character is equal to eight individual spaces. Some text applications say that a tab character is equal to four individual spaces, whereas others still let you explicitly define how big a tab character is. These variances have led to the tab character being unreliable for spacing purposes in code. Most professional integrated developer environments (IDEs) let you define the [Tab] key on a keyboard as individual spaces, in addition to letting you define how many spaces to insert.

Some examples of IDEs are Coda, Adobe Dreamweaver, Eclipse, Zend Studio, and Microsoft Visual Studio: These are all development environments for either directly writing or generating source code. In addition, most IDEs try to guess what you mean when writing a source document by intelligently adjusting the number of spaces. For example, when you press [Return] to begin a new line in your source code document, the IDE can indent the new line with at least as much space as the preceding line. Most IDEs behave this way by default. Dreamweaver automatically inserts two spaces when you press the [Tab] key. Coda, Eclipse, and Zend Studio can all be configured to insert spaces instead of tab characters when you press the [Tab] key.

Throughout this book, I use four spaces for a [Tab] key, although limited space may sometimes require that I use two characters. Generally, the professional standard for client-side source code is two characters because four characters makes file sizes much larger. I've stuck with four because concerns about file size and bandwidth usage can be addressed by compressing your source code when it's used on a production website.

Control Structures

Control structures include programming statements that begin with the keywords `if`, `else`, `switch`, `case`, `for`, `while`, `try`, `catch`, and so on. Control structure programming statements are the building blocks of any programming language. Now see how control structure statements should be formatted with regard to popular programming standards and guidelines.

Although, ultimately, different people have different preferences for how to write source code, there are two prevailing methods for formatting control structures in use by the majority of the professional programming community.

The following convention, which is formally called *K&R Style*, is included in Sun's Coding Standards Guidelines for Java:

```
if (condition) {
    something = 1;
} else if (another) {
    something = 2;
} else {
    something = 3;
}
```

In the preceding code example, you see that the curly braces and the parentheses are used as markers for indention.

Compare the preceding to the next convention, which is known as *Allman Style*, which is the default in Microsoft Visual Studio:

```
if (condition)
{
    something = 1;
}
else if (another)
{
    something = 2;
}
else
{
    something = 3;
}
```

In Allman Style, all the curly braces line up in the source code, which makes it easier to detect when one is missing, in addition to preventing typos like missing curly braces from occurring in the first place because you have a visual aid for their placement. It also lends itself nicely to having more space between lines of code, making things easier to read.

When function calls, like `window.open` in the example, are long, sometimes the function call is broken up over multiple lines to make it easier to read. To the browser,

```
window.open(
    this.href,
    "picture",
    "scrollbars=no,width=300,height=280,resizable=yes"
);
```

and

```
window.open(this.href, "picture", "scrollbars=no,width=300,height=280,resizable=
yes");
```

are exactly the same. The former example just makes it easier for humans to parse the arguments present in the function call.

Sometimes these two conventions are mixed to form a third convention, which is known as the *One True Brace* convention. This convention is defined in the Coding Standards Guidelines for PHP's PEAR repository.

```
window.onload = function()
{
    var nodes = document.getElementsByTagName('a');

    for (var counter = 0, length = nodes.length; counter < length; counter++) {
        nodes[counter].onclick = function(event) {
            window.open(
                this.href,
                "picture",
                "scrollbars=no,width=300,height=280,resizable=yes"
            );
```

```
                event? event.preventDefault() : (window.event.returnValue = false);
            };
        }
    };
```

In the One True Brace convention, the function assigned to `window.onload` follows the Allman Style, while the code within it follows K&R Style.

When I write JavaScript code, I prefer a mixture of Allman Style and K&R Style. I use Allman Style for all function and class definitions as well as control structures, and I use K&R Style for array and object definitions (JSON), and function calls. In practice this looks something like this:

```
$(document).ready(
    function()
    {
        $('a#examplePumpkin').click(
            function(event)
            {
                event.preventDefault();

                window.open(
                    'pumpkin.jpg',
                    'Pumpkin',
                    'scrollbars=no,width=300,height=280,resizable=yes'
                );
            }
        );

    }
);
```

Which programming convention you use is, of course, a matter of personal taste. Often which convention to use can lead to endless battles among programming teams; sometimes people have different tastes. How you indent your code can be a touchy and personal topic. You should use whichever convention makes the most sense for you. Although the methods I've showcased are the most popular, there are a multitude of variations that exist out there in the real world. You can find more information about programming indention styles on Wikipedia at `http://en.wikipedia.org/wiki/Indent_style`.

Optional Curly Braces and Semicolons

In the previous conventions, you'll note that there is always a single space between the keyword that begins the control structure, like `if` and the opening parenthesis. The following is a `switch` control structure using the first convention:

```
switch (variable) {
    case 1:
         condition = 'this';
        break;

    case 2:
         condition = 'that';
```

```
            break;

        default:
             condition = 'those';
    }
```

Note in the preceding that no `break` statement appears in the default case. As the default, a break is implied, and it is not necessary to include the `break` statement. I tend to deviate from the norm with how I prefer `switch` control structures to be written.

```
    switch (variable)
    {
        case 1:
        {
            condition = 'this';
            break;
        };
        case 2:
        {
            condition = 'that';
            break;
        };
        default:
        {
            condition = 'those';
        };
    }
```

I like to add curly braces around each `case` in the `switch` statement; I do this because I believe it makes the `switch` statement easier to read and flow better to my eyes; however, ultimately, these are not necessary. Concerning optional curly braces, I always include them, even if they're technically optional. The same goes for semicolons. Terminating each line with a semicolon is technically optional in JavaScript, although there are some circumstances in which you won't be able to omit it. I include all optional semicolons and curly braces, as I think that this not only makes the code cleaner, more organized, and consistent, but also gives you a technical benefit. If you want to compress your code to remove all additional white space, comments, and so on, these optional bits suddenly are no longer optional, but needed to keep the program functional after it's been compressed. In the following example, you can see what I mean by optional components:

```
    if (condition)
        something = 1
    else if (another)
        something = 2
    else
        something = 3
```

In JavaScript, the preceding code is perfectly valid. The semicolon is implied where there is a line break. And as long as there is only a single statement being executed, technically you don't have to include curly braces. However, the above fails when it is compressed:

```
    if (condition) something = 1 else if (another) something = 2 else something = 3
```

The preceding fails with a syntax error when you try to execute it. It fails because the script interpreter has no idea where you intend one statement to end and the next to begin. The language could probably be extended to guess in some circumstances, but it's better to just be as explicit as possible. Some combination and compression tools such as *require.js* do their best to fill missing bits and are actually very good at it.

Something else that you might think is odd is the inclusion of a semicolon after some function definitions. You'll see this in JavaScript because a function can be a type of data, just like a number is a type of data or a string is a type of data. In JavaScript, it's possible to pass a function around as you would a number or a string. You can assign a function to a variable and execute the function later. You've already seen an example of this, and here it is again in the following code:

```
window.onload = function()
{
    var nodes = document.getElementsByTagName('a');

    for (var counter = 0, length = nodes.length; counter < length; counter++) {
        nodes[counter].onclick = function(event) {
            window.open(
                this.href,
                 "picture",
                 "scrollbars=no,width=300,height=280,resizable=yes"
            );
             event? event.preventDefault() : (window.event.returnValue = false);
        };
    }
};
```

In the preceding code, you can see that a function is assigned to the `onload` event of the `window` object. The function definition is terminated with a semicolon. Again, that semicolon is technically optional in this example, but I include it because I want the code to work if it gets compressed, and I think that it makes the code more consistent, organized, and easier to follow.

Naming Variables, Functions, Objects

Variable naming is also accounted for in the coding standards I follow throughout this book. I always use the `camelCase` convention when naming variables, functions, objects, or anything that I can potentially invent a name for. This is contrasted with underscore naming conventions, for example, `underscores_separate_words`.

Namespace JavaScript Code

It's important to think about the big picture when writing an application. Whether you're writing an application for your own use or writing an application that will be deployed in varying environments that you have no control over, you're likely to run into one problem at some point in your career: naming conflicts. I touched on this topic when I talked about namespacing class and id names in your CSS and markup. The same principles are also applicable to JavaScript. Your script applications need to run without invading the global namespace too much. I say "too much" because you need to invade it somewhat, but you need to do so in a controlled and intelligent way. As you may have done for your markup and CSS, namespacing your JavaScript may be as simple as sticking to object-oriented code, wrapping all your programs in just one or a handful of objects and then naming those objects in the global namespace in a noninvasive way. A common approach is to namespace those objects with a prefix of some kind that doesn't infringe on some other existing project. One example is how the jQuery JavaScript framework is namespaced. jQuery does a lot, but for all the code that's included in jQuery, there are precious few intrusions made on the global namespace, the "jQuery" object, and the dollar sign method the jQuery object is aliased to. All the functionality that jQuery provides is provided through those objects, and one of those two, the dollar sign variable can be turned off. (As it turns out this is a common thing for frameworks to do, to bind to a variable named `$`, so the ability of turning it off allows jQuery to be installed alongside other JavaScript frameworks.)

Without a well-thought-out approach to the namespacing problem, it's possible that your application may cause conflicts with others. It's best to just assume that everything you place in the global namespace will cause a conflict, and thus set out to make as minimal as possible an intrusion into the global namespace.

Avoid Browser Detection

Browser detection can be a real annoyance. You're surfing the web using your favorite browser, and you hit a website that locks you out—not because your web browser is technically incapable, but because it didn't match what the website's creators presupposed would be capable. So, I propose the following:

- Make no assumptions about the capabilities of a visitor's browser.
- Test for feature compatibility, rather than a browser name or browser version.
- Account for the official standards and the de facto standards. (Official standards should take precedence—de facto standards will either become or be replaced by the former.)
- The world is always changing—what's most popular today may not remain the most popular in the months and years to come.
- It may be time to turn to a framework for some compatibility bridging.

Anyone remember a company called Netscape? At one time, Netscape was the dominant, de facto standard. Now Netscape holds virtually nothing of the world market share, and Chrome, Firefox, Safari, and Internet Explorer are dominant. Another great example: At its most popular, IE held more than 90 percent of the market. Now IE holds 50 percent or less, and other browsers all hold the remaining 50 percent. On mobile, Safari and Chrome are the overwhelming dominant market leaders because they power the browsers on iOS and Android platforms. The browser market can and does fluctuate and change. In the real world, there are a lot of people who use less popular browsers. Some browsers hold 2 percent or less. Two percent may sound small at first glance, but keep in mind that can be 2 percent of a very large number, and thus itself be a very large number. According to `www.internetworldstats.com`, in 2013, as I write this, there are just more than 2.4 billion Internet users worldwide, which is 34.3 percent of the world's population. Therefore, the so-called less popular browsers aren't really doing too shabby in the grand scheme of things, and although 2 percent sounds small, it's actually a pretty large base of users. Throughout this book I present numerous examples to you of how to avoid using browser detection and use feature detection instead.

SUMMARY

jQuery takes what would otherwise be a more complex or verbose task in traditional JavaScript, and it makes it much easier, sometimes reducing many lines to one or a few. Throughout this book, you will learn about what jQuery has to offer and how to use its simple, easy-to-understand API to write spectacular, professional-appearing web applications.

In this chapter, I talked a little about what jQuery is, where it comes from, and who develops and maintains it, and I showed you how to install it and begin using it. In the next chapter, you get right down to business, learning about jQuery's powerful implementation of the Selectors API and its world-class Event API.

If you are interested in learning more about jQuery's origins, visit `www.jquery.com` and `www.ejohn.org`.

This chapter also covered some things that a good programmer will want to get into the habit of doing, such as adopting a formal programming convention and avoiding conflicts with others' code through using a namespace of some sort (whether that be via a feature provided by the language, or through prefixing the names that you use that make an impact on the global namespace). I've shown a few of the practices that I have adopted for myself; although I should emphasize that it doesn't matter what programming convention you adopt, but rather that you adopt one. The premise of a programming convention is that you have a set of rules that you can follow to format your code so that it is neat, organized, and easy to follow. My conventions might not be what you want, but there are many others to choose from.

You should avoid detecting the user's browser, especially when it may lead to one group or another being locked out from functionality.

Your code should take advantage of client-side caching and the increase in performance it provides.

In my opinion, it is better to write code in neatly organized modules and combine those into a larger script later using server-side programming.

Finally, it is also important that you adopt standards for the presentation and maintenance of client-side markup and CSS. Choose either XHTML5 or HTML5, both of which are accepted standards. I prefer XHTML, although it may be too strict for your taste.