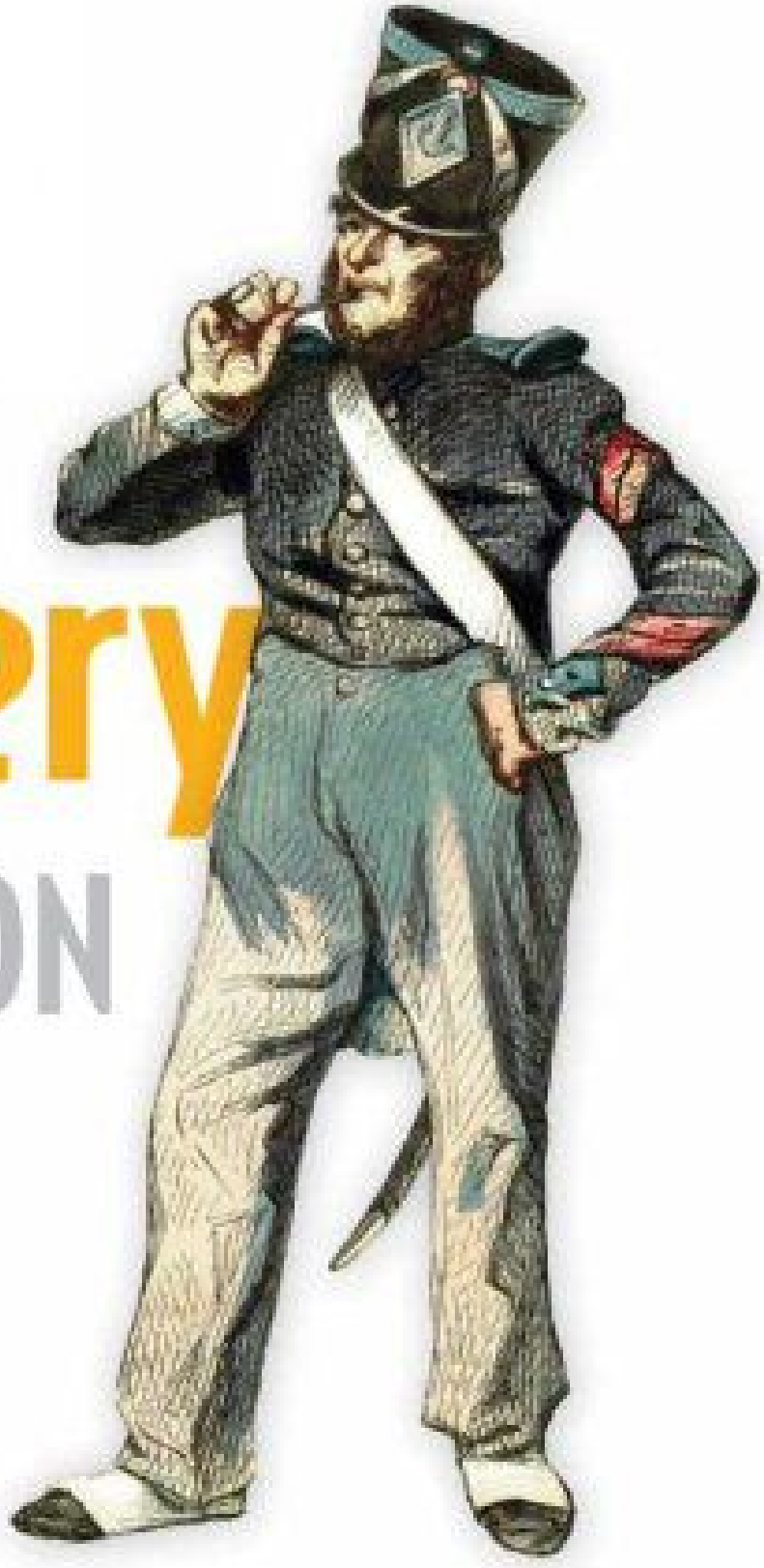
Bear Bibeault
Yehuda Katz

Covers jQuery 1.4 and jQuery UI 1.8



jQuery IN ACTION

SECOND EDITION

MANNING

Table of Contents

1 Introducing jQuery

This chapter covers

- Why you should use jQuery
- What *Unobtrusive JavaScript* means
- The fundamental elements and concepts of jQuery
- Using jQuery in conjunction with other JavaScript libraries

Sneered at as a "not-very-serious" language by many web developers for much of its lifetime, JavaScript has regained its prestige in the past few years as a result of the renewed interest in highly interactive, next-generation web applications (which you might also have heard referred to as *rich internet applications* or DOM-*scripted applications*) and Ajax technologies. The language has been forced to grow up quickly as client-side developers have tossed aside cut-and-paste JavaScript for the convenience of full-featured JavaScript libraries that solve difficult cross-browser problems once and for all, and provide new and improved patterns for web development.

A relative latecomer to this world of JavaScript libraries, jQuery has taken the web development community by storm, quickly winning the support of major companies for use in mission-critical applications. Some of jQuery's prominent users include the likes of IBM, Netflix, Amazon, Dell, Best Buy, Twitter, Bank of America,

Chapter 1. Introducing jQuery. jQuery in Action, Second Edition, ISBN: 9781935182320

and scores of other prominent companies. Microsoft has even elected to distribute jQuery with its Visual Studio tool, and Nokia uses jQuery on all its phones that include their Web Runtime component.

Those are *not* shabby credentials!

Compared with other toolkits that focus heavily on clever JavaScript techniques, jQuery aims to change the way that web developers think about creating rich functionality in their pages. Rather than spending time juggling the complexities of advanced JavaScript, designers can leverage their existing knowledge of Cascading Style Sheets (CSS), Hypertext Markup Language (HTML), Extensible Hypertext Markup Language (XHTML), and good old straightforward JavaScript to manipulate page elements directly, making rapid development a reality.

In this book, we're going to take an in-depth look at what jQuery has to offer us as developers of highly interactive web applications. Let's start by finding out exactly what jQuery brings to the web development party.

You can obtain the latest version of jQuery from the jQuery site at http://jquery.com/. Installing jQuery is as easy as placing it within your web application and using the HTML `<script>` tag to include it in your pages, like this:

```
<script type="text/javascript"
        src="scripts/jquery-1.4.js"></script>
```

The specific version of jQuery that the code in this book was tested against is included as part of the downloadable code examples (available at http://www.manning.com/bibeault2).

1.1 Power in the economy of code

If you've spent any time at all trying to add dynamic functionality to your pages, you've found that you're constantly following a pattern of selecting an element (or group of elements) and operating upon those elements in some fashion. You could be hiding or revealing the elements, adding a CSS class to them, animating them, or inspecting their attributes.

Using raw JavaScript can result in dozens of lines of code for each of these tasks, and the creators of jQuery specifically created that library to make common tasks trivial. 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 might be implemented as follows:

```
var checkedValue;
var elements = document.getElementsByTagName('input');
for (var n = 0; n < elements.length; n++) {
  if (elements[n].type == 'radio' &&
      elements[n].name == 'someRadioGroup' &&
```

```
      elements[n].checked) {
    checkedValue = elements[n].value;
  }
}
```

Contrast that with how it can be done using jQuery:

```
var checkedValue = $('[name="someRadioGroup"]: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 out your own terse—but powerful—jQuery statements to make your pages come alive. Let's briefly examine how this powerful code snippet works.

We identify all elements that possess a `name` attribute with the value `someRadioGroup` (remember that radio groups are formed by naming all their elements using the same name), then filter that set to only those that are in `checked` state, and find the value of that element. (There will be only one such element, because the browser will only allow a single element of the radio group to be checked at a time.)

The real power in this jQuery statement comes from the *selector*, which is an expression used to identify target elements on a page. It allows us to easily locate and grab the elements that we 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/bibeault2 (don't forget the 2 at the end). Unpack the code and either navigate to the files individually, or use the nifty index page at index.html in the root of the unpacked folder hierarchy.

Load the HTML page that you find in file chapter1/radio.group.html into your browser. This page, shown in figure 1.1, uses the jQuery statement that we just examined to determine which radio button has been checked.

Even this simple example ought to convince you that jQuery is the hassle-free way to create the next generation of highly interactive web applications. But just wait until we show you how much power jQuery offers for taming your web pages as we progress through this book's chapters.

We'll soon study how to easily create the jQuery selectors that made this example so easy, but first let's examine how the inventors of jQuery think JavaScript can be most effectively used on our pages.

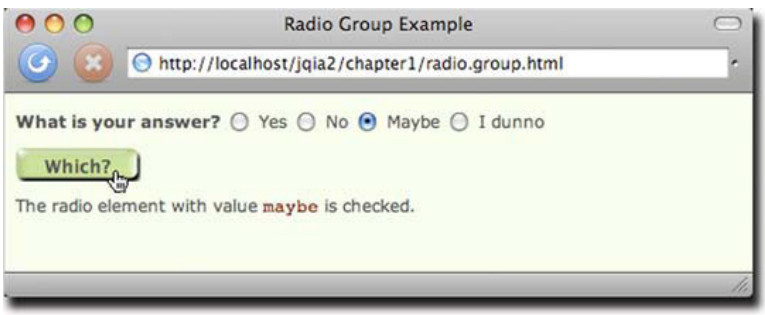


Figure 1.1 Determining which radio button is checked is easy to accomplish in one statement with jQuery!

1.2 Unobtrusive JavaScript

You may recall the bad old days before CSS, when we were forced to mix stylistic markup with the document structure markup in our 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 our web development toolkits allows us 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 our documents easier to manage, it also gives us the versatility to completely change the stylistic rendering of a page by simply swapping out different style sheets.

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

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

You can easily see that the style of this button element, including the font of its caption, isn't applied via the use of the `<font>` tag and other deprecated style-oriented markup, but is determined by whatever CSS rules (not shown) are in effect on the page. But although this declaration doesn't mix *style* markup with structure, it does mix *behavior* with structure by including 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, turns some Document Object Model (DOM) element with the `id` value of `xyz` red).

Let's examine how we 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.

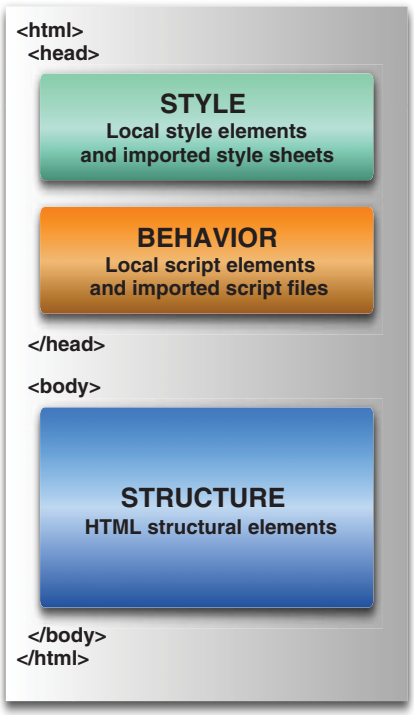


Figure 1.2 With structure, style, and behavior each neatly tucked away within a page, readability and maintainability are maximized.

This strategy, known as *Unobtrusive JavaScript*, was brought into the limelight by the inventors of jQuery and 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 embedded in the `<body>` of HTML pages, either as attributes of HTML elements (such as `onclick`) or in script blocks placed within 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 type="button" id="testButton">Click Me</button>
```

Much simpler! But now, you'll note, the button doesn't *do* anything. We 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, we'll segregate the script by moving it to a script block in the `<head>` section of the page, *outside* the scope of the document body (see note below), as follows:

```
<script type="text/javascript">
  window.onload = function() {
    document.getElementById('testButton').onclick = function() {
      document.getElementById('xyz').style.color = 'red';
    };
  };
</script>
```

We place the script in the `onload` handler for the page to assign an inline function to the `onclick` attribute of the button element.

We add this script in the `onload` handler (as opposed to within an inline script block) because we need to make sure that the button element exists *before* we attempt to augment it. (In section 1.3.3 we'll see how jQuery provides a better place for us to put such code.)

> **NOTE** For performance reasons, script blocks can also be placed at the bottom of the document body, though modern browsers make the performance difference rather moot. The important concept is to avoid embedding behavioral elements within the structural elements.

If any of the code in this example looks odd to you (such as the concept of function literals and inline functions), fear not! The appendix to this book provides a look at the important JavaScript concepts that you'll need to use jQuery effectively. We'll also examine, in the remainder of this chapter, how jQuery makes writing this code easier and quicker and makes the code more versatile all at the same time.

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 our goal than when we 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 us to write our 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.

As mentioned earlier, the jQuery team has specifically focused jQuery on the task of making it easy and delightful for us to code our pages using Unobtrusive JavaScript techniques, without paying a hefty price in terms of effort or code bulk. We'll find that making effective use of jQuery will enable us to accomplish much more on our pages while writing *less* code.

Without further ado, let's start looking at how jQuery makes it so easy for us to add rich functionality to our pages without the expected pain.

1.3 jQuery fundamentals

At its core, jQuery focuses on retrieving elements from HTML pages and performing operations upon 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, or placement within the document. With jQuery, we can employ that knowledge, and that degree of power, to vastly simplify our JavaScript.

jQuery places a high priority on ensuring that code will work consistently across all major browsers; many of the more difficult JavaScript problems, such as waiting until the page is loaded before performing page operations, have been silently solved for us.

Should we find that the library needs a bit more juice, jQuery has a simple but powerful built-in method for extending its functionality via *plugins*. Many new jQuery programmers find themselves putting this versatility into practice by extending jQuery with their own plugins on their first day.

Let's start by taking a look at how we can leverage our CSS knowledge to produce powerful, yet terse, code.

1.3.1 The jQuery wrapper

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 through the use of selectors, which concisely represent elements based upon their type, attributes, or position within the HTML document.

Those familiar with XML might be reminded of XPath 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.

For example, the selector

```
p a
```

refers to the group of all links (<a> elements) that are nested inside a <p> element. jQuery makes use of the same selectors, supporting not only the common selectors currently used in CSS, but also some that may not yet be fully implemented by all browsers, including some of the more powerful selectors defined in CSS3.

To collect a group of elements, we pass the selector to the jQuery function using the simple syntax

```
$(selector)
```

or

```
jQuery(selector)
```

Although you may find the `$()` notation strange at first, most jQuery users quickly become fond of its brevity. For example, to wrap the group of links nested inside any <p> element, we can use the following:

```
$("p a")
```

The `$()` function (an alias for the `jQuery()` function) returns a special JavaScript object containing an array of the DOM elements, in the order in which they are defined within the document, that match the selector. This object possesses a large number of useful predefined methods that can act on the collected group of elements.

In programming parlance, this type of construct is termed a *wrapper* because it wraps the collected elements with extended functionality. We'll use the term *jQuery wrapper* or *wrapped set* to refer to this set of matched elements that can be operated on with the methods defined by jQuery.

Let's say that we want to hide all elements that possess the class `notLongForThisWorld`. The jQuery statement is as follows:

```
$("div.notLongForThisWorld").hide();
```

A special feature of a large number of these methods, which we often refer to as jQuery *wrapper methods*, is that when they're done with their action (like a hide operation), they return the same group of elements, ready for another action. For example, say that we want to add a new class, `removed`, to each of the elements in addition to hiding them. We would write

```
$("div.notLongForThisWorld").hide().addClass("removed");
```

These jQuery *chains* can continue indefinitely. It's not uncommon to find examples in the wild of jQuery chains dozens of methods long. And because each method works on all of the elements matched by the original selector, there's no need to loop over the array of elements. It's all done for us behind the scenes!

Even though the selected group of objects is represented as a highly sophisticated JavaScript object, we can pretend it's a typical array of elements, if necessary. As a result, the following two statements produce identical results:

```
$("#someElement").html("I have added some text to an element");
```

and

```
$("#someElement")[0].innerHTML =
  "I have added some text to an element";
```

Because we've used an ID selector, only one element will match the selector. The first example uses the jQuery `html()` method, which replaces the contents of a DOM element with some HTML markup. The second example uses jQuery to retrieve an array of elements, selects the first one using an array index of `0`, and replaces the contents using an ordinary JavaScript property assignment to `innerHTML`.

If we wanted to achieve the same results with a selector that results in multiple matched elements, the following two fragments would produce identical results (though the latter example is not a recommended way of coding using jQuery):

```
$("div.fillMeIn")
  .html("I have added some text to a group of nodes");
```

and

```
var elements = $("div.fillMeIn");
for(var i=0;i<elements.length;i++)
  elements[i].innerHTML =
    "I have added some text to a group of nodes";
```

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 we want. Additionally, jQuery supports not only the selectors that you have already come to know and love, but also more advanced selectors—defined as part of the CSS specification—and even some custom selectors.

Here are a few examples:

Selector	Results
`$("p:even")`	Selects all even `<p>` elements
`$("tr:nth-child(1)")`	Selects the first row of each table
`$("body > div")`	Selects direct `<div>` children of `<body>`
`$("a[href$= 'pdf ']")`	Selects links to PDF files
`$("body > div:has(a)")`	Selects direct `<div>` children of `<body>`-containing links

That's powerful stuff!

You'll be able to leverage your existing knowledge of CSS to get up and running fast, and then learn about the more advanced selectors that jQuery supports. We'll be covering jQuery selectors in great detail in chapter 2, and you can find a full list on the jQuery site at http://docs.jquery.com/Selectors.

Selecting DOM elements for manipulation is a common need in web pages, but some things that we'll also need to do don't involve DOM elements at all. Let's take a brief look at what jQuery offers beyond element manipulation.

1.3.2 Utility functions

Even though wrapping elements to be operated upon is one of the most frequent uses of jQuery's `$()` function, that's not the only duty to which it's assigned. One of its additional duties is to serve as the *namespace prefix* for a handful of general-purpose utility functions. Because so much power is given to page authors by the jQuery wrapper created as a result of a call to `$()` with a selector, it's somewhat rare for most page authors to need the services provided by some of these functions. In fact, we won't be looking at the majority of these functions in detail until chapter 6 as a preparation for writing jQuery plugins. But you *will* see a few of these functions put to use in the upcoming sections, so we'll briefly introduce them here.

The notation for these functions may look odd at first. Let's take, for example, the utility function for trimming strings. A call to it could look like this:

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

If the `$.` prefix looks weird to you, remember that `$` is an identifier like any other in JavaScript. Writing a call to the same function using the `jQuery` identifier, rather than the `$` alias, may look a bit less odd:

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

Here it becomes clear that the `trim()` function is merely namespaced by `jQuery` or its `$` alias.

> **NOTE** Even though these elements are called the utility *functions* in jQuery documentation, it's clear that they're actually *methods* of the `$()` function (yes, in JavaScript, functions can have their own methods). We'll put aside this technical distinction and use the term *utility function* to describe these methods so as not to introduce terminology that conflicts with the online documentation.

We'll explore one of these utility functions that helps us to extend jQuery in section 1.3.5, and one that helps jQuery peacefully coexist with other client-side libraries in section 1.3.6. But first, let's look at another important duty that jQuery's `$()` function performs.

1.3.3 The document ready handler

When embracing Unobtrusive JavaScript, behavior is separated from structure, so we're performing operations on the page elements outside of the document markup that creates them. In order to achieve this, we 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 must load 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

```
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 after all external resources are fully loaded and the page is displayed in the browser window. This includes not only resources like images, but QuickTime and Flash videos embedded in web pages, and there are more and more of them these days. 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 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 (using our hiding example) is as follows:

```
jQuery(document).ready(function() {
  $("div.notLongForThisWorld").hide();
});
```

First, we wrap the `document` instance with the `jQuery()` function, and then we apply the `ready()` method, passing a function to be executed when the document is ready to be manipulated.

We called that the *formal syntax* for a reason; a shorthand form, used much more frequently, is as follows:

```
jQuery(function() {
  $("div.notLongForThisWorld").hide();
});
```

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

That's another use of the `$()` function; now let's look at yet something else it can do for us.

1.3.4 Making DOM elements

As you can see by now, the authors of jQuery avoided introducing a bunch of global names into the JavaScript namespace by making the `$()` function (which you'll recall is merely an alias for the `jQuery()` function) versatile enough to perform many duties. Well, there's one more duty that we need to examine.

We can create DOM elements on the fly by passing the `$()` function a string that contains the HTML markup for those elements. For example, we can create a new paragraph element as follows:

```
$("<p>Hi there!</p>")
```

But creating a disembodied DOM element (or hierarchy of elements) isn't all that useful; usually the element hierarchy created by such a call is then operated on using one of jQuery's DOM manipulation functions. Let's examine the code of listing 1.1 as an example.

Listing 1.1 Creating HTML elements on the fly

```
<html>
  <head>
    <title>Follow me!</title>
    <link rel="stylesheet" type="text/css"
          href="../styles/core.css"/>
    <script type="text/javascript" src="../scripts/jquery-1.4.js">
    </script>
    <script type="text/javascript">
      $(function(){
        $("<p>Hi there!</p>").insertAfter("#followMe");
      });
    </script>
  </head>

  <body>
    <p id="followMe">Follow me!</p>
  </body>
</html>
```

1 **Ready handler that creates HTML element**

2 **Existing element to be followed**

This example establishes an existing HTML paragraph element named `followMe` 2 in the document body. In the script element within the `<head>` section, we establish a ready handler 1 that uses the following statement to insert a newly created paragraph into the DOM tree after the existing element:

```
$("<p>Hi there!</p>").insertAfter("#followMe");
```

The result is shown in figure 1.3.

We'll be investigating the full set of DOM manipulation functions in chapter 3, where we'll see that jQuery provides many means to manipulate the DOM to create nearly any structure that we may desire.

Figure 1.3 Dynamically creating and inserting elements, usually requiring many lines of code, can be accomplished in a single line of jQuery code.

Now that you've seen the basic syntax of jQuery, let's take a look at one of the most powerful features of the library.

1.3.5 Extending jQuery

The `jQuery` wrapper function provides a large number of useful methods we'll use again and again in these pages. But no library can anticipate everyone's needs. It could be argued that no library should even try to anticipate every possible need; doing so could result in a large, clunky mass of code that contains little-used features that merely serve to gum up the works!

The authors of the jQuery library recognized this concept and worked hard to identify the features that most page authors would need and included only those in the core library. Recognizing also that page authors would each have their own unique requirements, jQuery was designed to be easily extended with additional functionality.

We could write our own functions to fill in any gaps, but once we've been spoiled by the jQuery way of doing things, we'll find that doing things the old-fashioned way is beyond tedious. By extending jQuery, we can use the powerful features it provides, particularly in the area of element selection.

Let's look at a specific example: jQuery doesn't come with a predefined function to disable a group of form elements. If we're using forms throughout an application, we might find it convenient to be able to write code such as the following:

```
$("form#myForm input.special").disable();
```

Fortunately, and by design, jQuery makes it easy to extend its set of methods by extending the wrapper returned when we call `$()`. Let's take a look at the basic idiom for how that's accomplished by coding a new `disable()` function:

```
$.fn.disable = function() {
  return this.each(function() {
    if (this.disabled == null) this.disabled = true;
  });
}
```

A lot of new syntax is introduced here, but don't worry about it too much yet. It'll be old hat once you've made your way through the next few chapters; it's a basic idiom that you'll use over and over again.

First, `$.fn.disable` means that we're extending the `$` wrapper with a method named `disable`. Inside that function, the `this` keyword is the collection of wrapped DOM elements that are to be operated upon.

Then, the `each()` method of this wrapper is called to iterate over each element in the wrapped collection. We'll be exploring this and similar methods in greater detail in chapter 3. Inside of the iterator function passed to `each()`, `this` is a reference to the specific DOM element for the current iteration. Don't be confused by the fact that `this` resolves to different objects within the nested functions. After writing a few extended functions, it becomes natural to remember that `this` refers to the function context of the current function. (The appendix is also there to explain the JavaScript concept of the `this` keyword.)

For each element, we check whether the element has a `disabled` attribute, and if it does, we set it to `true`. We return the result of the `each()` method (the wrapper) so that our brand new `disable()` method will support chaining, like many of the native jQuery methods. We'll be able to write

```
$("form#myForm input.special").disable().addClass("moreSpecial");
```

From the point of view of our page code, it's as though our new `disable()` method was built into the library itself! This technique is so powerful that most new jQuery users find themselves building small extensions to jQuery almost as soon as they start to use the library.

Moreover, enterprising jQuery users have extended jQuery with sets of useful functions that are known as *plugins*. We'll be talking more about extending jQuery in this way in chapter 7.

Testing for existence

You might have seen this common idiom for testing the existence of an item:

```
if (item) {
  //do something if item exists
}
else {
  //do something if item doesn't exist
}
```

The idea here is that if the item doesn't exist, the conditional expression will evaluate to `false`.

Although this works in most circumstances, the framers of jQuery feel that it's a bit too sloppy and imprecise and recommend the more explicit test used in the `$.fn.disable` example:

```
if (item == null) ...
```

This expression will correctly test for `null` or `undefined` items.

For a full list of the various coding styles recommended by the jQuery team, visit the jQuery documentation page at http://docs.jquery.com/JQuery_Core_Style_Guidelines.

Before we dive into using jQuery to bring life to our pages, you may be wondering if we're going to be able to use jQuery with Prototype or other libraries that also use the `$` shortcut. The next section reveals the answer to this question.

1.3.6 Using jQuery with other libraries

Even though jQuery provides a set of powerful tools that will meet most of our needs, there may be times when a page requires that multiple JavaScript libraries be employed. This situation could come about when we're transitioning an application from a previously employed library to jQuery, or we might want to use both jQuery and another library on our pages.

The jQuery team, clearly revealing their focus on meeting the needs of their user community rather than any desire to lock out other libraries, have made provisions for allowing jQuery to cohabitate with other libraries.

First, they've followed best-practice guidelines and have avoided polluting the global namespace with a slew of identifiers that might interfere not only with other libraries, but also with names we might want to use on our pages. The identifier `jQuery` and its alias `$` are the limit of jQuery's incursion into the global namespace. Defining the utility functions that we referred to in section 1.3.2 as part of the `jQuery` namespace is a good example of the care taken in this regard.

Although it's unlikely that any other library would have a good reason to define a global identifier named `jQuery`, there's that convenient but, in this particular case, pesky `$` alias. Other JavaScript libraries, most notably the Prototype library, use the `$` name for their own purposes. And because the usage of the `$` name in that library is key to its operation, this creates a serious conflict.

The thoughtful jQuery authors have provided a means to remove this conflict with a utility function appropriately named `noConflict()`. Anytime after the conflicting libraries have been loaded, a call to

```
jQuery.noConflict();
```

will revert the meaning of `$` to that defined by the non-jQuery library.

We'll cover the nuances of using this utility function in chapter 7.

1.4 Summary

We've covered a great deal of material in this whirlwind introduction to jQuery, in preparation for diving into using jQuery to quickly and easily enable the development of next-generation web applications.

jQuery is generally useful for any page that needs to perform anything but the most trivial of JavaScript operations, but 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 upon the parameters passed to it.

As we've seen, the `jQuery()` function can be used to do the following:

- Select and wrap DOM elements to be operated upon by wrapper methods
- Serve as a namespace for global utility functions
- Create DOM elements from HTML markup
- Establish code to be executed when the DOM is ready for manipulation

jQuery behaves like a good on-page citizen not only by minimizing its incursion into the global JavaScript namespace, but also by providing an official means to reduce that minimal incursion in circumstances when a name collision might still occur—namely when another library, such as Prototype, requires use of the `$` name. How's *that* for being user friendly?

In the chapters that follow, we'll explore all that jQuery has to offer us as developers of rich internet applications. We'll begin our tour in the next chapter as we learn how to use jQuery selectors to quickly and easily identify the elements that we wish to act upon.