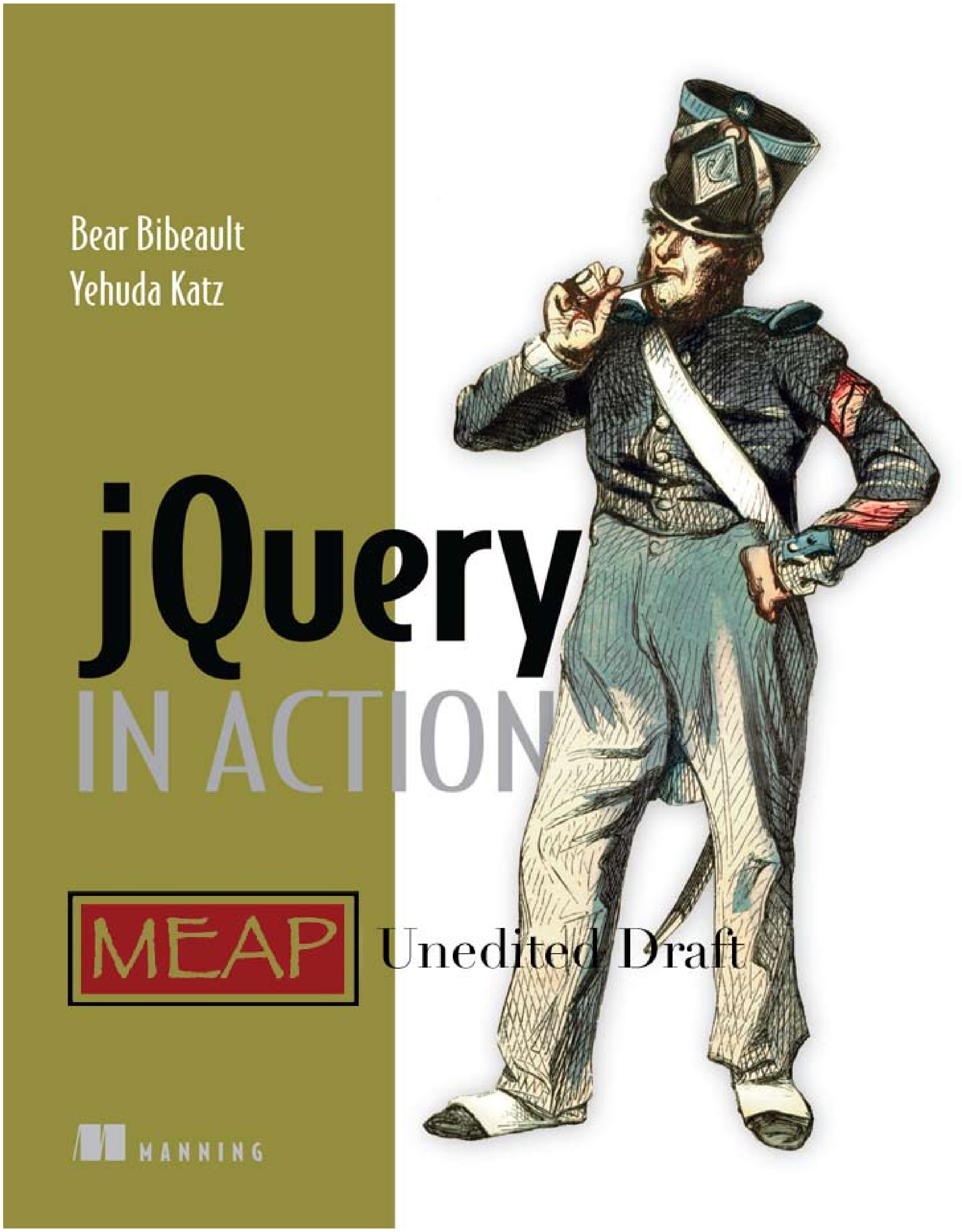


Bear Bibeault
Yehuda Katz

jQuery IN ACTION

MEAP Unedited Draft

MANNING

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=337

MEAP Edition
Manning Early Access Program

Copyright 2007 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=337

Contents

Chapter 1 - Introducing jQuery
Chapter 2 - Creating the wrapped element set
Chapter 3 - Bringing pages to life with jQuery
Chapter 4 - Events are where it happens
Chapter 5 - Animations and Effects
Chapter 6 - jQuery utility functions and flags
Chapter 7 - Creating your own plugins
Chapter 8 - Talk to your server with Ajax
Chapter 9 - The Core UI Plugin
Chapter 10 - Official Plugins
Appendix A - JavaScript that you need to know but might not!
Appendix B - Alphabetical API Reference

Chapter 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

Considered a "toy" language by serious web developers for most of its lifetime, JavaScript has regained its prestige in the past few years as a result of the renewed interest in Rich Internet Applications and Ajax technologies. As a result, the language has had to grow up quickly, with client-side developers tossing aside cut-and-paste JavaScript for the convenience of full-featured JavaScript libraries that solve difficult cross-browser problems once and for all and which have provided new and improved paradigms 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 websites like MSNBC and well-regarded open-source projects, including SourceForge, Trac, and Drupal.

Compared with other toolkits that focus heavily on clever JavaScript techniques, jQuery aims to change the way that web developers fundamentally think about creating rich functionality in their pages. Rather than spending a lot of time juggling the complexities of object-oriented JavaScript, designers can leverage their existing knowledge of CSS, XHTML and straight-forward JavaScript to manipulate page elements directly, making more rapid development a reality.

Let's find out what exactly jQuery brings to the page development party.

1.1 Why jQuery?

If you've spent any time at all trying to layer dynamic functionality onto your pages, you'll find that you are constantly following a common 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, or adding a CSS class to them, or animating them, or simply modifying their attributes.

Using raw JavaScript, each of those tasks can require dozens of lines of JavaScript code. The creators of jQuery, on the other hand, specifically created the library to make these common types of tasks trivial. For example, designers will commonly use JavaScript to "zebra-stripe" tables, highlighting every other row in a table with a contrasting color.

Using native JavaScript, that can take up to ten lines of code or more. Here's how you accomplish it using jQuery:

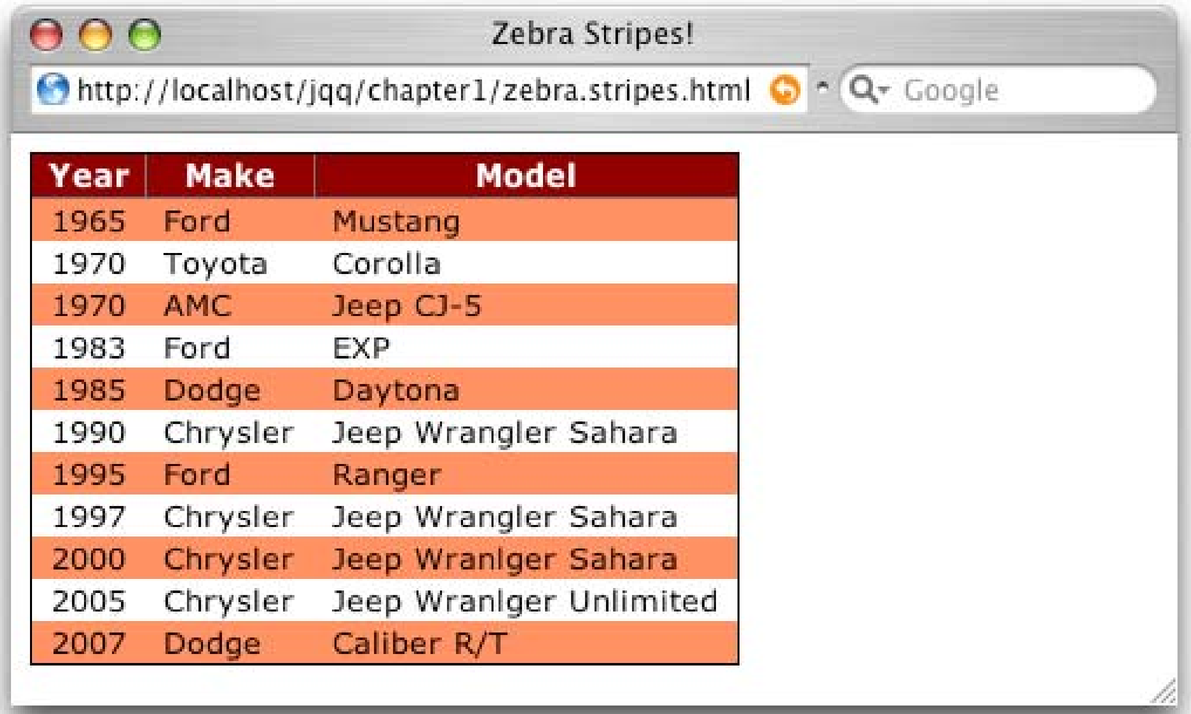
```
$("table tr:nth-child(even)").addClass("striped");
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=337

Don't worry if that looks a bit cryptic to you right now. In very short order, you'll be understanding how it works and whipping out your own terse, but powerful, jQuery statements to make your pages come alive. But first, let's briefly examine how this code snippet works.

Simply stated, we identify every even row (`<tr>` element) in all of the tables on the page and add the CSS class `striped` to each of those rows. By applying the desired background color to these rows via a CSS rule for class `striped`, a single line of JavaScript can improve the aesthetics of the entire page.

When applied to a sample table, the effect could be as shown in figure 1.1.



Year	Make	Model
1965	Ford	Mustang
1970	Toyota	Corolla
1970	AMC	Jeep CJ-5
1983	Ford	EXP
1985	Dodge	Daytona
1990	Chrysler	Jeep Wrangler Sahara
1995	Ford	Ranger
1997	Chrysler	Jeep Wrangler Sahara
2000	Chrysler	Jeep Wranlger Sahara
2005	Chrysler	Jeep Wranlger Unlimited
2007	Dodge	Caliber R/T

Figure 1.1: Adding "zebra stripes" to a table in just one statement!

Of course, the real power in this jQuery statement comes from the *selector*, which allowed us to easily identify and "grab" just the elements that we needed; in this case, every even `<tr>` element in all tables. You will find the full source for this page in the downloadable source code for this book in file `chapter1/zebra.stripes.html`.

We'll be studying how to create these selectors with ease, but first let's talk a little about how the inventors of jQuery view how JavaScript can be most effectively used in your pages.

1.2 Unobtrusive JavaScript

Remember the bad old days before CSS (Cascading Style Sheets), when we were forced to mix stylistic markup with the document structure markup in our HTML pages?

I'm sure that we all do, and with perhaps a little shudder. The addition of CSS to our web development toolkits allowed us to separate stylistic information from the document structure and give 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 stylesheets.

Few of us would voluntarily regress back to the days of applying style with HTML elements, and yet, markup such as the following is still quite common:

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

We can easily imagine that the style of this button element, to include the font of its caption, is provided by CSS rules loaded via a stylesheet rather than the use of the `<font>` tag and other deprecated style-oriented markup. But while this declaration does not mix style with structure, it does mix *behavior* with structure, by including the JavaScript that is to be executed when the button is clicked as part of the markup of the button element (which in this case, turns something named `xyz` red upon a click of the button).

For all the same reasons that it is desirable to segregate style and structure within an HTML document, it is also becoming recognized that separation of *behavior* from structure has just as many, if not more, benefits.

This movement is known as *Unobtrusive JavaScript* and the inventors of jQuery have focused that library strongly on helping page authors easily achieve this separation in their pages. Unobtrusive JavaScript, along with the legions of the jQuery-savvy, considers any JavaScript expressions or statements that are 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 "not correct".

"But how would we 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 will have noted, the button doesn't *do* anything.

Rather than embedding the button's behavior in its markup, we'll move it to a script block in the `<head>` section of the page, outside the scope of the document body, as follows:

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

  function makeItRed() {
    document.getElementById('xyz').style.color = 'red';
  }
</script>
```

We've placed script in the `onload` handler for the page to assign a function, `makeItRed()`, to the `onclick` attribute of the button element. We added this script in the `onload` handler because we need to make sure that the button element exists *before* we attempt to manipulate it. (In section 1.3.3 we'll see how jQuery provides a better place for us to put such code.)

If any of the code in this example looks funky to you, fear not! In chapter 4 we'll be taking a look at the JavaScript concepts that we'll need to use jQuery effectively. We'll also be examining, in the remainder of this very chapter, how jQuery will make writing the above code easier, shorter and more versatile all at the same time.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=337

Unobtrusive JavaScript, while a powerful technique to further add to the clear separation of responsibilities within a web application, does not come without its price. You might already have noticed that it took a few more lines of script to accomplish our goal than when we just shoved it into the button markup. Unobtrusive JavaScript may not only increase the amount of script that might need to be written, it also requires some discipline and the application of good coding patterns to the client-side script.

None of that is bad – in fact, 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 very *good* thing! But it *is* extra work. That is, without jQuery.

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 in order to do so. We will find, in fact, that making effective use of jQuery will enable us to accomplish much much more on our pages by writing *less* code.

Without further ado, let's start taking a look at how jQuery makes it 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 your HTML page and performing operations upon them. If you're familiar with CSS, you are already well aware of the power of selectors, which describe groups of elements by their attributes or placement within the document. With jQuery, you will be able to leverage your knowledge and that degree of power to vastly simplify your JavaScript.

jQuery places a high priority on ensuring that your code will work in a consistent manner across all major browsers, so you'll find that many of the more difficult JavaScript problems, such as waiting until the page is loaded before performing page operations, have been silently solved for you.

Should you find that the library needs a bit more juice, its developers have built in a simple but powerful method for extending its functionality. Many new jQuery programmers find themselves putting this versatility into practice by extending jQuery on their first day.

But first, let's take a look at how you can leverage your 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 attributes or position within the HTML document.

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 very same selectors, supporting not only the common selectors currently used in CSS, but also the more powerful ones not yet fully implemented by most browsers. The `nth-child` selector from the "zebra-striping" code that we examined earlier is a good example of a more powerful selector that is defined in CSS3.

To collect a group of elements, you use the simple syntax:

```
$(selector)
```

or

```
jQuery(selector)
```

Although you may find the `$()` notation a bit strange at first, most jQuery users quickly become quite fond of its brevity.

For example, to retrieve the group of links nested inside a `p` element, you would use

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

The `$()` function (simply an alias for `jQuery()` function), returns a special JavaScript object containing an array of the elements that match the selector. This object possesses a large number of useful predefined methods that can act on the group of elements.

In programming parlance, this type of construct is termed a *wrapper* in that it "wraps" the matching element(s) with extended functionality. We'll use the term *jQuery wrapper* to refer to the wrapped set of matched elements that can be operated on by the functions defined by jQuery

Let's say that you want to fade out all `div` elements with the CSS class `notLongForThisWorld`. The jQuery statement would be:

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

A special feature of these functions is that when they're done with their action (like fading out), they return the same group of elements, ready for another action. For example, say that you want to add a new CSS class, `removed`, to each of the elements in addition to fading them out. You could write:

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

These jQuery "chains" can continue indefinitely. It's not uncommon to find examples in the wild of jQuery chains dozens of commands long. And because each function 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 you behind the scenes!

Yet, even though the selected group of objects is represented as a highly sophisticated JavaScript object, you 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");

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

Because we have used an ID selector, only one element will match the selector. In the first example, we use the jQuery method `html()`, which replaces the contents of a page element with some HTML markup. In the second example, we use jQuery to retrieve an array of elements, select the first one using an array index of 0, and replace the contents using an ordinary JavaScript means.

If you wanted to achieve the same results with a selector that resulted in multiple matched elements, the following two fragments would produce identical results:

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

var elements = $("div. fillMeIn");
for(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, leveraging jQuery's chainability will continue to reduce the lines of code necessary to produce the results that you want. Additionally, jQuery supports not only the selectors that you have already come to know and love, but it also supports more advanced selectors -- defined as part of the CSS3 specification -- as well as basic XPath expressions, and even some custom selectors.

Here are a few examples:

```
// All even <p> elements
$("p:even");

// The first row of each table
$("tr:nth-child(1)");

// <div>s that are direct children of the <body> element
$("body > div");

// Links to PDF files
$("a[@href$=pdf]");

// <div>s that are direct children of the <body> element,
// and contain links
$("body > div[a]");
```

Powerful stuff!

You will 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 section 2.1, and you can find a full list at http://docs.jquery.com/Selectors.

Next, let's take a brief look at more that jQuery offers.

1.3.2 Utility functions

Even though wrapping elements to be operated upon is one of the most frequent uses of jQuery's `$()` function, that's not the only duty to which it is assigned. One of its additional duties is to serve as the *namespace prefix* for a handful of general-purpose utility functions. Because so much power is given to page authors by the jQuery wrapper created as a result of a call to `$()` with a selector, it's somewhat rare for most page authors to feel the need for the services provided by some of these functions. As such, we won't be looking at the majority of these functions in detail until chapter 8 as a preparation for examining writing jQuery plugins. But we *will* be seeing a few of these functions put to use in the upcoming sections, so we're introducing their concept here.

The notation for these functions may look a bit odd at first. Let's take, for example, the `trim()` utility function. A call to it could be:

```
$.trim(someString);
```

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

```
jQuery.trim(someString);
```

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

We'll explore one of these utility functions that helps us to extend jQuery in section 1.3.4, and one that helps jQuery peacefully coexist with other client-side libraries in section 1.3.5. But first, let's take a 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 and so you'll be 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 loaded before those operations execute. Thinking back to the "zebra-striping" example, the entire table must be loaded before striping can be applied.

Traditionally, the `onload` handler for the `window` instance is used for this purpose, which executes statements after the entire page is fully loaded. The syntax is typically something like:

```
window.onload = function() {
  $("table tr:nth-child(even)").addClass("even");
}
```

This causes the "zebra-striping" code to execute after the document is fully loaded. Unfortunately, the browser not only delays executing the `onload` code until after the Document Object Model (DOM) tree is created, it also waits until after all images and other external files are fully loaded, and the page is displayed in the browser window. As a result, visitors can experience a delay between the time that they can first see the page and the time that the `onload` script is executed.

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

A much better approach would be to *only* wait until the document structure is fully parsed and the browser has converted the HTML into its DOM tree form 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, but not external image resources, has loaded. The formal syntax to define such code (using our striping example) is:

```
$(document).ready(function() {
  $("table tr:nth-child(even)").addClass("even");
});
```

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

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=337

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

```
$(function() {
  $("table tr:nth-child(even)").addClass("even");
});
```

By simply passing a function to `$()`, we are instructing the browser to wait until the DOM has fully loaded (but only the DOM) before executing the code. Even better, you can use this technique multiple times within the *same* HTML document and the browser will execute all of the functions you specify. In contrast, the window's `onload` technique allows for only a single function.

Now let's see yet something else that the `$()` function can do for us.

1.3.4 Making DOM elements

It's become apparent by this point that the authors of jQuery avoiding introducing a bunch of global names into the JavaScript namespace by making the `$()` function (which you will recall is merely an alias for the `jQuery()` function) versatile enough to perform many duties. Well, there's yet one more that we need to examine.

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

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

But just creating a disembodied DOM element (or hierarchy of elements) isn't all that useful, so 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>
    <script type="text/javascript" src="scripts/jquery-1.1.2.js">
    </script>
    <script type="text/javascript">
      $(function(){                                          1
        $("<p>Hi there!</p>").insertAfter("#followMe");
      });
    </script>
  </head>

  <body>
    <p id="followMe">Follow me!</p>                         2
  </body>
</html>
```

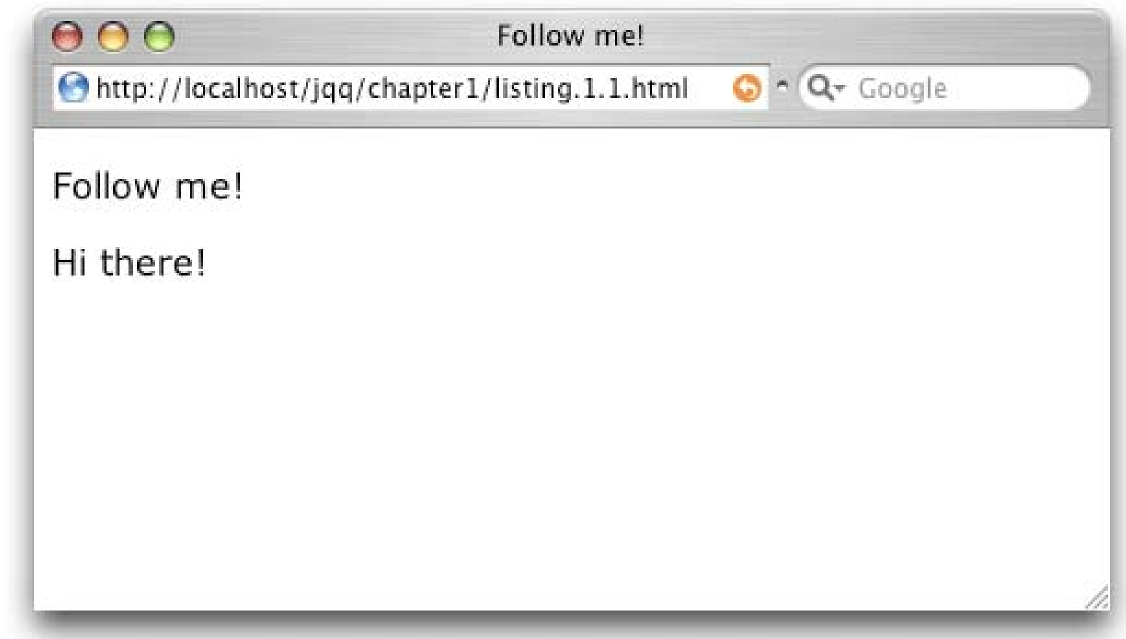
1 Ready handler that creates HTML element
2 Existing element to be followed

Replace #1 and #2 in following paragraph with cueballs

In this example, we establish 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 add a newly created paragraph to the DOM tree after the existing element:

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

The result is as shown in figure 1.2.



Follow me!

Hi there!

Figure 1.2: A dynamically created and inserted element

We'll be investigating the full set of DOM manipulation functions in chapter 2, where we'll see that jQuery provides many means to manipulate the DOM to achieve just about any structure that we may desire.

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 really useful functions that you will find yourself using again and again in your pages. But no library can anticipate everyone's needs. Indeed, 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 recognize this concept and have worked hard to identify the features that most page authors will need and included only those needs in the core library. Recognizing also that page authors will each have their own unique needs, jQuery has been designed to be easily extending with additional functionality.

But why *extend* jQuery versus just writing standalone function to fill in any gaps?

That's an easy one! By extending jQuery, you can use the powerful features it provides, particularly in the area of element selection.

Let’s look at a particular example: jQuery does not come with a predefined function to disable a group of form elements. And if you’re using forms throughout your application, as you probably are, you might find it convenient to be able to use the following syntax:

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

Fortunately, and quite by design, jQuery makes it really easy to extend its set of functions, by simply extending the “wrapper” that is returned when you call `$()`. Let’s take a look at the basic idiom for how that is accomplished:

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

There’s a lot of new syntax introduced here, but don’t worry about it too much yet. It’ll be “old hat” by the time you make way through the next few chapters and it’s a basic idiom that you’ll use over and over again.

First, `$.fn.disable` just means that you’re extending `$` with a function called `disable`. Inside that function, `this` is the collection of wrapped elements that are to be operated upon.

Then, the `each()` function of this wrapper is called to iterate over each element in the wrapped collection. We’ll be exploring this, and similar, functions in greater detail in chapter 2. Inside of the iterator function passed to `each()`, `this` is a pointer 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.

For each element, we check whether the element has a disabled attribute, and if it does, we set it to true. We return the results of the each() function (the wrapper), so that your brand new disable() function will support chaining just like all the native jQuery functions. So you’ll be able to do:

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

From the point of view of your page code, it’s just as though your new `disable()` function 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, as well as introducing the official plugins that are freely available, in chapter 9.

Before we dive in to using jQuery to bring life to your pages, you may be wondering if you’re going to be able to use jQuery with Prototype, or other libraries that also use the `$` shortcut. Let’s see if that’s possible.

1.3.6 Using jQuery with other libraries

Even though jQuery provides a set of powerful tools that will meet the majority of the needs for most page authors, there may be times when a page requires that multiple JavaScript libraries be employed. This situation could come about because you are in the process of transitioning an application from a previously employed library to jQuery, or you just might want to use both jQuery and another library on your 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 such co-habitation of other libraries with jQuery on your pages.

Firstly, they have followed best-practice guidelines and have avoided polluting the global namespace with a slew of identifiers that might interfere with not only other libraries, but with names that you yourself might want to use on the page. The identifiers `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.

While 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, `$` alias. Other JavaScript libraries, most notably the popular Prototype library, use the `$` name for their own purposes. And since the usage of the `$` name in that library is rather 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 further cover the nuances of using this utility function in section 7.2.

1.4 Summary

In this whirlwind introduction to jQuery we've covered a great deal of material in preparation for diving into using jQuery to quickly and easily enable Rich Internet Application development.

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

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

- Select and wrap DOM elements to operate upon
- 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 it incursion into the global JavaScript namespace, it even provides 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 remain, we'll be exploring all that jQuery has to offer us as page authors of Rich Internet Applications. We'll begin our tour in the next chapter with bringing our pages to life via DOM manipulation.