SAMPLE CHAPTER

Bear Bibeault
Yehuda Katz

FOREWORD BY John Resig
Creator of jQuery

jQuery in Action

MANNING

jQuery in Action

by Bear Bibeault and Yehuda Katz

Sample Chapter 1

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. 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 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 such as 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 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), Extensible Hypertext Markup Language (XHTML), and good old straightforward JavaScript to manipulate page elements directly, making more rapid development a reality.

In this book, we're going to take an in-depth look at what jQuery has to offer us as page authors of Rich Internet Applications. Let's start by finding out what exactly jQuery brings to the page-development party.

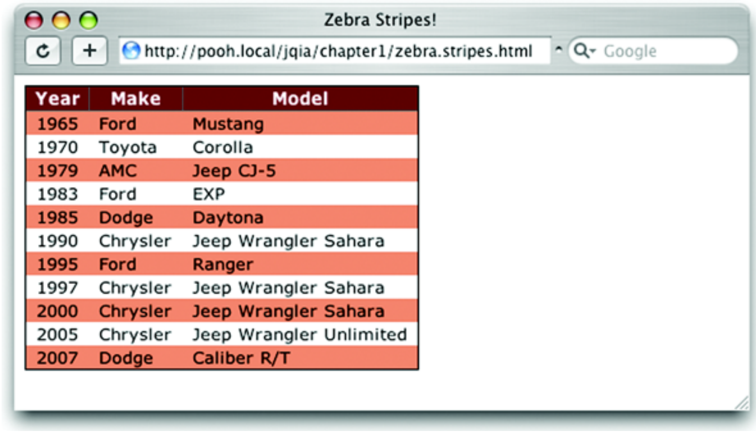
1.1 Why jQuery?

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 modifying their attributes.

Using raw JavaScript can result in dozens of lines of code for each of these tasks. The creators of jQuery specifically created the library to make common tasks trivial. For example, designers will use JavaScript to "zebra-stripe" tables—highlighting every other row in a table with a contrasting color—taking up to 10 lines of code or more. Here's how we accomplish it using jQuery:

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

Don't worry if that looks a bit cryptic to you right now. In short order, you'll understand how it works, and you'll be whipping out your own terse—but powerful—



Year	Make	Model
1965	Ford	Mustang
1970	Toyota	Corolla
1979	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 Wrangler Sahara
2005	Chrysler	Jeep Wrangler Unlimited
2007	Dodge	Caliber R/T

Figure 1.1 Adding "zebra stripes" to a table is easy to accomplish in one statement with jQuery!

jQuery statements to make your pages come alive. Let's briefly examine how this code snippet works.

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.

The real power in this jQuery statement comes from the *selector*, an expression for identifying target elements on a page that allows us to easily identify and grab the elements we need; in this case, every even `<tr>` element in all tables. You'll find the full source for this page in the downloadable source code for this book in file chapter1/zebra.stripes.html.

We'll study how to easily create these selectors; but first, let's examine how the inventors of jQuery think JavaScript can be most effectively used in our pages.

1.2 Unobtrusive JavaScript

Remember 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 and, perhaps, with more than a little shudder. The addition of CSS to our web development toolkits allows 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, but it also gives us the versatility to completely change the stylistic rendering of a page by swapping out different stylesheets.

Few of us would voluntarily regress back to the days of applying style 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>
```

We can easily see that the style of this button element, including the font of its caption, is not applied via the use of the `<font>` tag and other deprecated style-oriented markup, but is determined by CSS rules 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 (which in this case turns some Document Object Model [DOM] element named `xyz` red upon a click of the button).

For all the same reasons that it's desirable to segregate style from structure within an HTML document, it's as beneficial (if not more so) to separate the *behavior* from the structure.

This movement is known as *Unobtrusive JavaScript*, and the inventors of jQuery have focused that library 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 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 would 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.

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 place the script in the `onload` handler for the page to assign a function, `makeItRed()`, to the `onclick` attribute of the button element. We add this script in the `onload` handler (as opposed to inline) 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 odd to you, fear not! Appendix A provides a look at the JavaScript concepts that you'll need to use jQuery effectively. We'll also be examining, in the remainder of this chapter, how jQuery makes writing the previous code easier, shorter, and more versatile all at the same time.

Unobtrusive JavaScript, though a powerful technique to further add to the clear separation of responsibilities within a web application, doesn't 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 placed it into the button markup. Unobtrusive JavaScript not only may increase the amount of script that needs to be written, but also requires some discipline and the application of good coding patterns to the client-side script.

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.

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'll find that making effective use of jQuery will enable us to accomplish much more on our pages by writing less code.

Without further ado, let's start taking a look at just 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 our 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 attributes or placement within the document. With jQuery, you'll be able to leverage your knowledge and that degree of power to vastly simplify your JavaScript.

jQuery places a high priority on ensuring our code will work in a consistent manner 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, 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 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 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 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 we examined earlier is a good example of a more powerful selector defined in CSS3.

To collect a group of elements, we use 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 retrieve the group of links nested inside a `<p>` element, we 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 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* because it wraps the matching element(s) 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 fade out all `<div>` elements with the CSS class `notLongForThisWorld`. The jQuery statement is as follows:

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

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

or

```
$("#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 method `html()`, which replaces the contents of a DOM element with some HTML markup. The second example uses 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 we want 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");
```

or

```
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 also more advanced selectors—defined as part of the CSS Specification—and even some custom selectors.

Here are a few examples.

```
$("p:even");
```

This selector selects all even `<p>` elements.

```
$("tr:nth-child(1)");
```

This selector selects the first row of each table.

```
$("body > div");
```

This selector selects direct `<div>` children of `<body>`.

```
$("a[href$=pdf]");
```

This selector selects links to PDF files.

```
$("body > div:has(a)")
```

This selector selects direct `<div>` children of `<body>`-containing links.

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 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.

Selecting DOM elements for manipulation is a common need in our pages, but some things that we also need to do don't involve DOM elements at all. Let's take a brief look at more that 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; we won't be looking at the majority of these functions in detail until chapter 6 as a preparation for writing jQuery plug-ins. But you *will* see 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 odd at first. Let's take, for example, the utility function for trimming strings. A call to it could be

```
$.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, looks a bit more familiar:

```
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 are actually *methods* of the `$()` function. We'll put aside this technical distinction and use the term *utility function* to describe these methods so as not to introduce conflicting terminology 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'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. In the zebra-striping example, the entire table must load before striping 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() {
  $("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 DOM tree is created but also waits until after all images and other external resources 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 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 the rich behaviors become available. This could make the whole Unobtrusive JavaScript movement 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 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 as follows:

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

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:

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

By passing a function to `$()`, 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 third-party code we might be using already uses the `onload` mechanism for its own purpose (not a best-practice approach).

We've seen another use of the `$()` function; now let's see yet something else that it can do for us.

1.3.4 Making DOM elements

It's become apparent by this point that 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

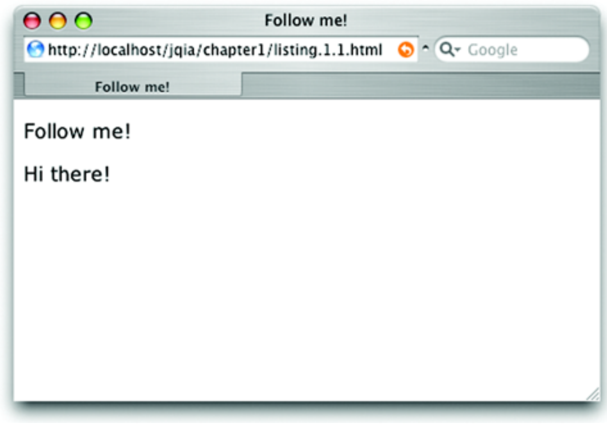
```
<html>
  <head>
    <title>Follow me!</title>
    <script type="text/javascript" src="../scripts/jquery-1.2.js">
    </script>
    <script type="text/javascript">
      $(function(){                                         ❶ Ready handler that creates HTML element
        $("<p>Hi there!</p>").insertAfter("#followMe");
      });
    </script>
  </head>

  <body>
    <p id="followMe">Follow me!</p>                          ❷ Existing element to be followed
  </body>
</html>
```

This example establishes an existing HTML paragraph element named `followMe` ❷ in the document body. In the script element within the `<head>` section, we establish a ready handler ❶ 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 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 you'll see that jQuery provides many means to manipulate the DOM to achieve 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 useful functions we'll find ourselves using again and again in our 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 needs in the core library. Recognizing also that page authors would each have their own unique needs, jQuery was designed to be easily extended with additional functionality.

But why extend jQuery versus writing standalone functions to fill in any gaps?

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

Let's look at a particular example: jQuery doesn't come with a predefined function to disable a group of form elements. And if we're using forms throughout our application, we might find it convenient to be able to use the following syntax:

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

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

```
$.fn.disable = function() {
  return this.each(function() {
    if (typeof this.disabled != "undefined") 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 by the time you make 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 function called `disable`. Inside that function, `this` 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 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, set it to `true`. We return the results 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, as well as introducing the official plugins that are freely available in chapter 9.

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 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 we're in the process of 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 such cohabitation of other libraries with jQuery on our pages.

First, they've 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 also names that you 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.

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 `$` alias. Other JavaScript libraries, most notably the popular Prototype library, use the `$` 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 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 its parameters.

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

- 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, 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?

You can obtain the latest version of jQuery from the jQuery site at http://jquery.com/. The version of jQuery that the code in this book was tested against (version 1.2.1) is included as part of the downloadable code.

In the chapters that follow, we'll explore all that jQuery has to offer us as page authors of Rich Internet Applications. We'll begin our tour in the next chapter as we bring our pages to life via DOM manipulation.

WEB DEVELOPMENT

jQuery in Action

Bear Bibeault • Yehuda Katz

"The best-thought-out and researched piece of literature on the jQuery library."

—FROM THE FOREWORD BY John Resig, Creator of jQuery

A really good web development framework anticipates your needs. jQuery does more—it practically reads your mind. Developers fall in love with this JavaScript library the moment they see 20 lines of code reduced to three. jQuery is concise and readable. Its unique "chaining" model lets you perform multiple operations on a page element in succession, as in

```
$("div.elements").addClass("myClass").load("ajax_url").fadeIn()
```

jQuery in Action is a fast-paced introduction and guide. It shows you how to traverse HTML documents, handle events, perform animations, and add Ajax to your web pages. The book's unique "lab pages" anchor the explanation of each new concept in a practical example. You'll learn how jQuery interacts with other tools and frameworks and how to build jQuery plugins. This book requires a modest knowledge of JavaScript and Ajax.

What's Inside

- Countless practical examples
- DOM manipulation and event handling
- Animation and UI effects
- Painless Ajax
- Based on jQuery 1.2

Bear Bibeault is a JavaRanch senior moderator and coauthor of Manning's *Ajax in Practice* and *Prototype and Scriptaculous in Action.* **Yehuda Katz** is a developer with Engine Yard. He heads the jQuery plugin development team and runs Visual jQuery.

"Solve your complex UI problems in no time. A must read."
—Jonathan Bloomer, Soap Creative

"A fantastic journey through jQuery."
—John C. Tyler
UBS Investment Bank

"A superior guide to the superior JavaScript library."
—Gregg Bolinger, VML

"This book is like jQuery itself —fast, effective, efficient."
—Eric Pascarello
Author of *Ajax in Action*

"*jQuery in Action* has become my best friend. A great read."
—Andrew Siemer, OTX Research

"*The Elements of Style* for JavaScript."
—Joshua Heyer, Trane Inc.

For more information, code samples, and to purchase an ebook visit manning.com/jQueryinAction

ISBN-13: 978-1933988351
ISBN-10: 1933988355
53999
9 781933 988351

MANNING $39.99 / Can $39.99