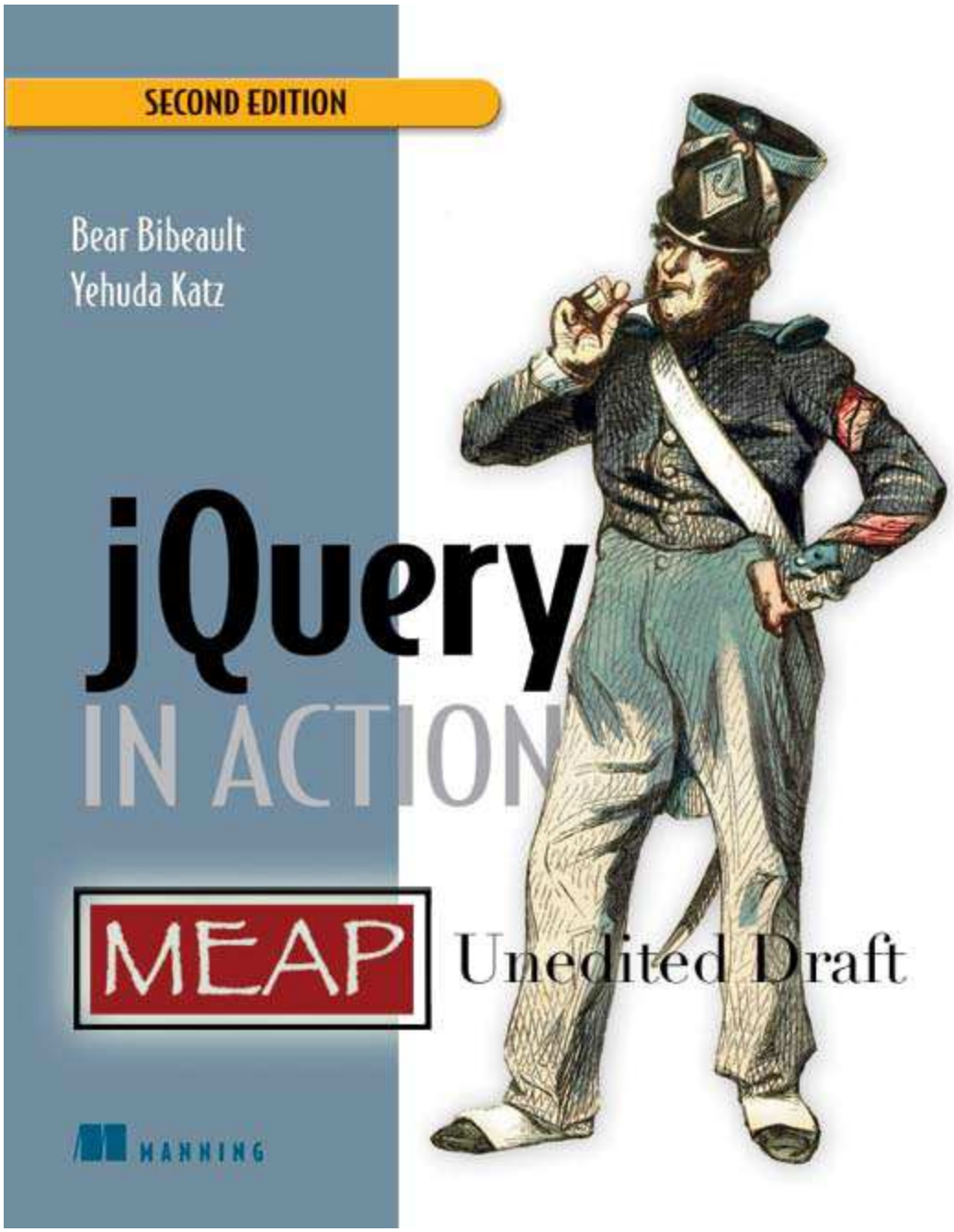

SECOND EDITION
Bear Bibeault
Yehuda Katz
jQuery
IN ACTION
MEAP
Unedited Draft
MANNING

MEAP Edition
Manning Early Access Program

Copyright 2009 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:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Contents

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

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 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 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 websites such as MSNBC, and well-regarded open source projects including SourceForge, Trac, and Drupal. Microsoft has elected to distribute jQuery with its Visual Studio tool, and Nokia uses jQuery on all its phones that include their *Web Run-Time* 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), 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 page authors of Rich Internet Applications. Let's start by finding out exactly what jQuery brings to the page-development party.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

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 inspecting 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, anyone who has dealt with radio groups in JavaScript can tell you that it's a lesson in tedium to discover which radio element of a radio group is currently checked and obtains 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.

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 to you 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 code snippet works.

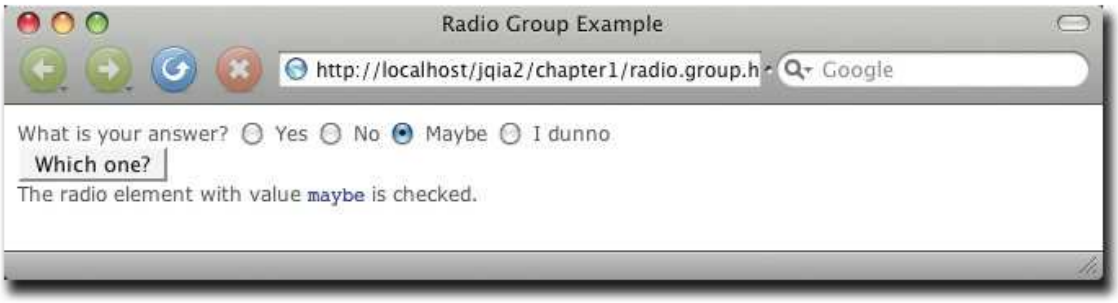


Figure 1.1 Determining which radio button is checked is easy to accomplish in one statement with jQuery!

We identify all elements that possess a `name` attribute with the value `someRadioGroup` (remember that radio groups are formed by naming all its 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, as 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*, an expression for identifying target elements on a page that allows us to easily locate and grab the elements that we need; in this case, the checked element in the radio group. You'll find the full source for this page in the downloadable source code for this book in file `chapter1/radio.group.html`.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

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

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 and, 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 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 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 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 just as beneficial (if not more so) to separate the *behavior* from the structure.

This strategy is known as *Unobtrusive JavaScript*, which was pioneered by the inventors of jQuery and is now embraced by every major JavaScript library, and helps page authors achieve this useful separation on their pages. As the library that spearheaded the movement, so to speak, jQuery's core is well optimized for producing Unobtrusive JavaScript quite productively. 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. We can click it all day long, and no behavior will result. Let's fix that.

But 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:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
<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.)

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 be examining, in the remainder of this chapter, how jQuery makes writing this example 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 *may* increase the line count of the script that needs to be written, and 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.

And, 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 while 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 type, attributes, or placement within the document. With jQuery, we'll be able to leverage our knowledge and that degree of power to vastly simplify our JavaScript.

jQuery places a high priority on ensuring that 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

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 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 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 collected group of elements.

In programming parlance, this type of construct is termed a *wrapper* because it wraps the collected 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 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 write

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

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");
```

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, selects the first one using an array index of `0`, and replaces the contents using an ordinary JavaScript property assignment to `innerHTML`.

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 chain-ability will continue to reduce the lines of code necessary to produce the results that 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.

```
$("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 that jQuery supports. We'll be covering jQuery selectors in great detail in section 2.2, and you can find a full list at `http://docs.jquery.com/Selectors`.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

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; in fact, 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 briefly introducing 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 be

```
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 are 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 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 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
};
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

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 have you noticed how many QuickTime and Flash videos are being embedded in web pages 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 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 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 (without waiting for external resources) has loaded. 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 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>")
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

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-1.3.2.min.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

Cueballs in code and text

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 as shown in figure 1.2.

Figure 1.2 A dynamically created and inserted element

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

We'll be investigating the full set of DOM manipulation functions in chapter 3, 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 requirements, 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 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 is accomplished by coding a new `disable()` function:

```
$.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 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 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 (and the Appendix is there to help with the JavaScript concept of the `this` keyword).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

For each element, we check whether the element has a `disabled` attribute, and if it does, 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, as well as introducing some of 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 with names that we might want to use on our pages. 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 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

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 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 operate upon with 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 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 1.3.2) 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 learn how to use jQuery selectors to quickly and easily identify the elements that we wish to act upon.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

2

Selecting the elements to act upon

This chapter covers:

- Selecting elements to be wrapped by jQuery using *selectors*
- Creating and placing new HTML elements in the DOM
- Manipulating the wrapped element set

In the previous chapter, we discussed the many ways that the jQuery function can be used. Its capabilities range from the selection of DOM elements to defining functions to be executed when the DOM is loaded.

In this chapter, we examine (in great detail) how the DOM elements to be acted upon are identified by looking at two of the most powerful and frequently used capabilities of jQuery's `$()` function: the selection of DOM elements via *selectors* and the creation of new DOM elements.

A good number of the capabilities required by DOM-Scripted Applications are achieved by manipulating the DOM elements that make up the pages. But before they can be manipulated, they need to be identified and selected. Let's begin our detailed tour of the many ways that jQuery lets us specify which elements are to be targeted for manipulation.

2.1 Selecting elements for manipulation

The first thing we need to do when using virtually any jQuery method (frequently referred to as jQuery *wrapper methods*) is to select some page elements to act upon. Sometimes, the set of elements we want to select will be easy to describe, such as "all paragraph elements on the page." Other times, they'll require a more complex description like "all list elements that have the class `listElement`, contain a link, and are first in the list."

Fortunately, jQuery provides a robust ***selector*** syntax with which we'll be able to easily specify virtually any set of elements elegantly and concisely. You probably already know a big chunk of the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

syntax: jQuery uses the CSS syntax you already know and love, and extends it with some custom means to select elements that help us to perform tasks both common and complex.

To help you learn about element selection, we've put together a ***Selectors Lab Page*** that's available within the downloadable code examples for this book. If you haven't yet downloaded the example code, now would be a great time to do so – the information in this chapter will be much easier to absorb by following along with the Lab exercises. Please see the book's front section for details on how to find and download this code.

The Selectors Lab allows us to enter a jQuery selector string and see (in real time!) which DOM elements get selected. The Selectors Lab can be found at `chapter2/lab.selectors.html` in the example code.

When displayed, the Lab should look as shown in figure 2.1 (if the panes don't appear correctly lined up, you may need to widen your browser window).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

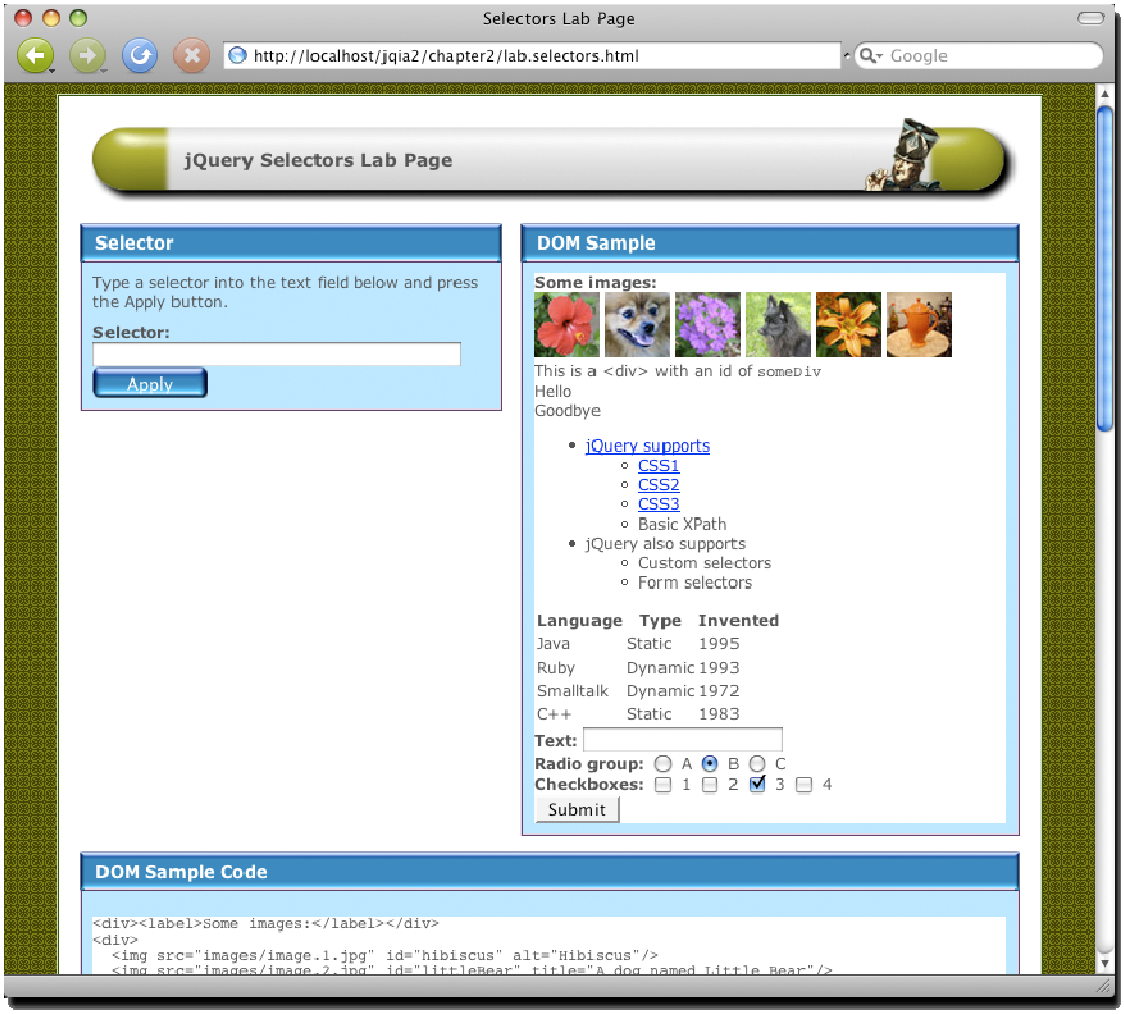


Figure 2.1 The Selectors Lab page allows us to observe the behavior of any selector we choose in real time.

The Selector pane at top left contains a text box and a button. To run a Lab "experiment", type a selector into the text box and click the Apply button. Go ahead and type the string `li` into the box, and click the Apply button.

The selector that you type (in this case `li`) is applied to the HTML fragment loaded into the DOM Sample pane at upper right. The Lab code that executes when Apply is clicked adds a class named

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`wrappedElement` to be applied to all matching elements. A CSS rules defined for the page causes all elements with that class to be highlighted with a red border and pink background. After clicking Apply, you should see the display shown in figure 2.2 in which all `<li>` elements in the DOM sample are highlighted.

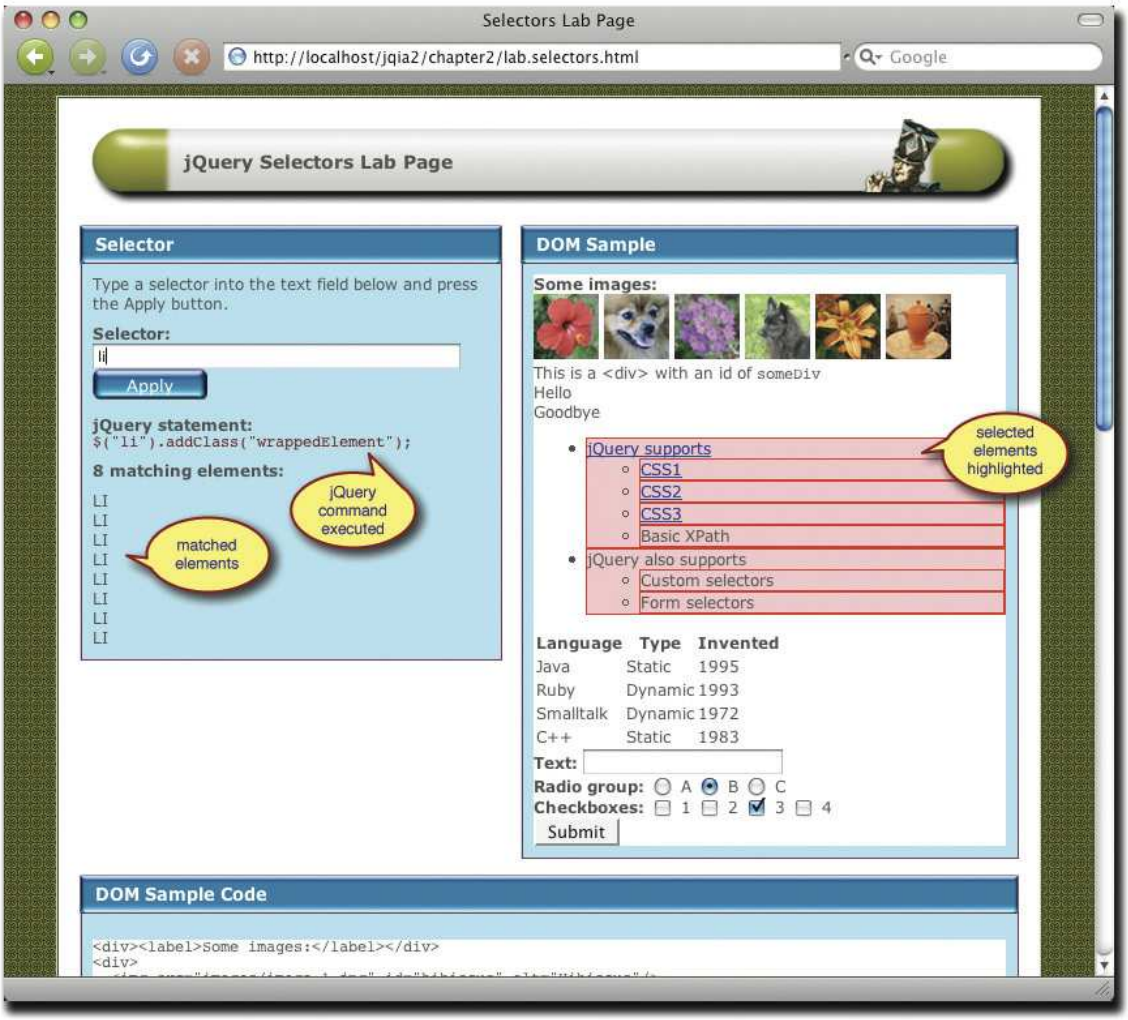


Figure 2.2 A selector value of `li` matches all `<li>` elements when applied as shown by the displayed results

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Note that the <li> elements in the sample fragment have been highlighted and that the executed jQuery statement, as well as the tag names of the selected elements, have been displayed below the Selector text box.

The HTML markup used to render the DOM Sample fragment is displayed in the lower pane labeled DOM Sample Code to help you experiment with writing selectors targeted at the elements in this sample.

We'll talk more about using this Lab as we progress through the chapter. But first, we must admit that we've been making a rather blatant over-simplification, and we're going to rectify that now.

2.1.1 Controlling the context

Up to this point, we've been acting as if there were only one argument passed to jQuery's $() function, but this was just a bit of hand waving to keep things simple at the start. In fact, for the variants in which a selector or an HTML fragment is passed to the $() function, a second argument is accepted. When the first argument is a selector, this second argument denotes the *context* of the operation.

As we will see with many of jQuery's methods, when an optional argument is omitted, a reasonable default is assumed. And so it is with the context argument. When a selector is passed as the first argument (we'll deal with passing HTML fragments later), the context defaults to applying that selector to every element in the DOM tree.

That's quite often exactly what we want, and so it's a nice default. But there may be times when we want to limit our search to a subset of the entire DOM. In such cases, we can provide a DOM element that serves as the root of the sub-tree to which the selector is applied.

The Selectors Lab is a good example of this scenario. When that page applies the selector that you typed into the text field, the selector is applied *only* to the subset of the DOM that is loaded into the DOM Sample pane.

In addition to using a DOM element reference as the context, we can also supply either a string that contains a jQuery selector, or a wrapped set of DOM elements. (So yes, that means that you can pass the result of one $() invocation to another -- don't let that make your head explode just yet; it's not as confusing as it may seem at first.)

When a selector or wrapped set is provided, the identified elements serve as the contexts for the application of the selector. As there can be multiple such elements, this is a nice way to provide disparate sub-trees in the DOM to serve as the contexts for the selection process.

Let's take the Lab Page as an example.

We'll assume that the selector string is stored in a variable conveniently named selector. When we apply this submitted selector, we only want to apply it to the sample DOM, which is contained within a element with an id value of sampleDOM.

Were we to code the call to the jQuery function as:

```
$(selector)
```

the selector would be applied to the entire DOM tree, including the form in which the selector was specified. That's not what we want. What we want is to limit the selection process to the sub-tree of the DOM rooted at the element with the id of sampleDOM; so instead we write:

```
$(selector,'div#sampleDOM')
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

which limits the application of the selector to the desired portion of the DOM.

OK, now that we know how to control where to apply selectors, let's see how to code them beginning with familiar territory: traditional CSS selectors.

2.1.2 Using basic CSS selectors

For applying styles to page elements, web developers have become familiar with a small, but powerful and very useful, group of selection expressions that work across all browsers. Those expressions include making selections by an element's ID, by CSS class names, by tag names, and by the hierarchy of the page elements within the DOM.

Table 2.1 provides some examples to give you a quick refresher:

Table 2.1 Some simple CSS selector examples

Example	Description
a	Matches all anchor (<a>) elements.
`#specialID`	Matches the element with the id value of `specialID`
`.specialClass`	Matches all elements with the class `specialClass`
`a#specialId.specialClass`	Matches the element with the id value `specialID` if it is an anchor tag and has class `specialClass`
`p a.specialClass`	Matches all anchor elements with the class `specialClass` that are descendants of <p> elements

We can mix and match the basic selector types to identify fairly fine-grained sets of elements.

With jQuery, we can easily select elements using the CSS selectors that we're already accustomed to using. To select elements using jQuery, we wrap the selector in `$()`, as in

```
$("p a.specialClass")
```

With a few exceptions, jQuery is fully CSS3 compliant, so selecting elements this way will come with no surprises; the same elements that would be selected in a style sheet by a standards-compliant browser will be selected by jQuery's selector engine. Note that jQuery does *not* depend upon the CSS implementation of the browser it's running within. Even if the browser doesn't implement a standard CSS selector correctly, jQuery will correctly select elements according to the rules of the World Wide Web Consortium (W3C) standard.

jQuery also lets us combine multiple selectors into a single expression using the comma operator. For example, to select all and all elements:

```
$('div,span')
```

For some exercise, go play with the Selectors Lab and run some experiments with some basic CSS selectors until you feel comfortable with them.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

These basic selectors are powerful, but sometimes we need even finer-grained control over which elements we want to match. jQuery meets this challenge and steps up to the plate with even more advanced selectors.

2.1.3 Using child, container, and attribute selectors

For more advanced selectors, jQuery uses the most up-to-date generation of CSS supported by Mozilla Firefox, Internet Explorer 7 and 8, Safari, and other modern browsers. These advanced selectors include selecting the direct children of some elements, elements that occur after other elements in the DOM, and even elements with attributes matching certain conditions.

Sometimes, we'll want to select only the direct children of a certain element. For example, we might want to select list elements directly under some list, but not list elements belonging to a sublist. Consider the following HTML fragment from the sample DOM of the Selectors Lab:

```
<ul class="myList">
  <li><a href="http://jquery.com">jQuery supports</a>
    <ul>
      <li><a href="css1">CSS1</a></li>
      <li><a href="css2">CSS2</a></li>
      <li><a href="css3">CSS3</a></li>
      <li>Basic XPath</li>
    </ul>
  </li>
  <li>jQuery also supports
    <ul>
      <li>Custom selectors</li>
      <li>Form selectors</li>
    </ul>
  </li>
</ul>
```

Suppose that we want to select the link to the remote jQuery site, but not the links to various local pages describing the different CSS specifications. Using basic CSS selectors, we might try something like `ul.myList li a`. Unfortunately, that selector would grab all links because they all descend from a list element.

You can verify this by entering the selector `ul.myList li a` into the Selectors Lab and clicking Apply. The results will be as shown in figure 2.3.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

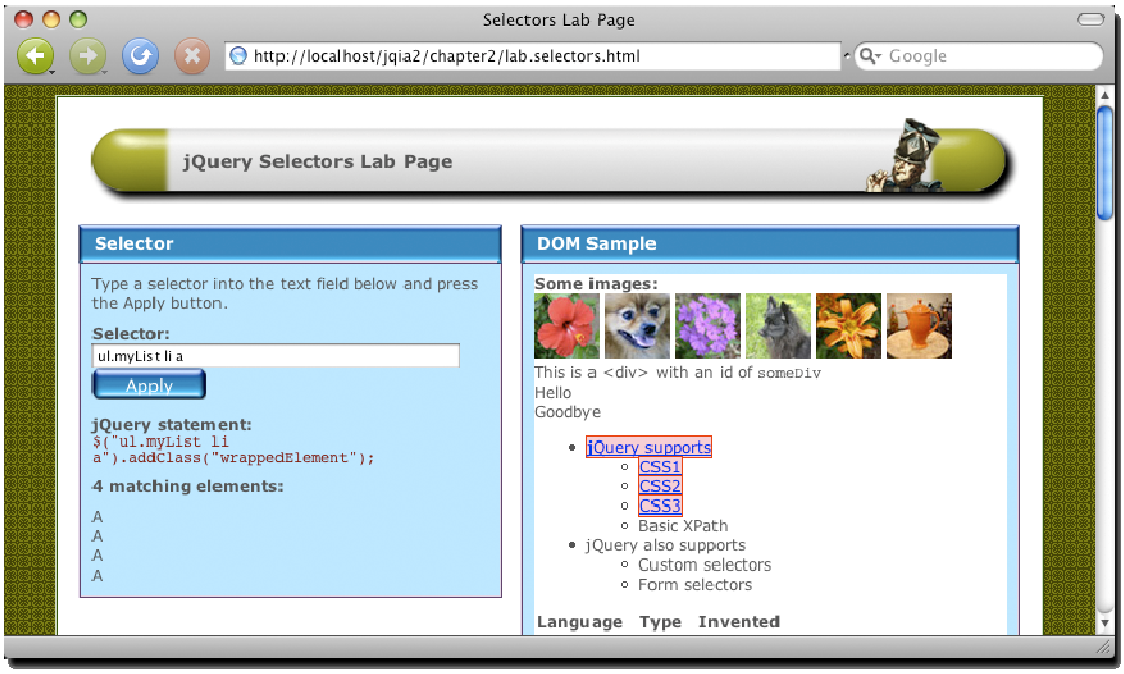


Figure 2.3 All anchor tags that are descendents, at any depth, of an `<li>` element are selected by `ul.myList li a`

A more advanced approach is to use *child selectors*, in which a parent and its *direct* child are separated by the right angle bracket character (>), as in

```
p > a
```

This selector matches only links that are *direct* children of a <p> element. If a link were further embedded, say within a within the <p>, that link would not be selected.

Going back to our example, consider a selector such as

```
ul.myList > li > a
```

This selector selects only links that are direct children of list elements, which are in turn direct children of `<ul>` elements that have the class `myList`. The links contained in the sublists are excluded because the `<ul>` elements serving as the parent of the sublists `<li>` elements don't have the class `myList`, as shown in the Lab results of figure 2.4.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

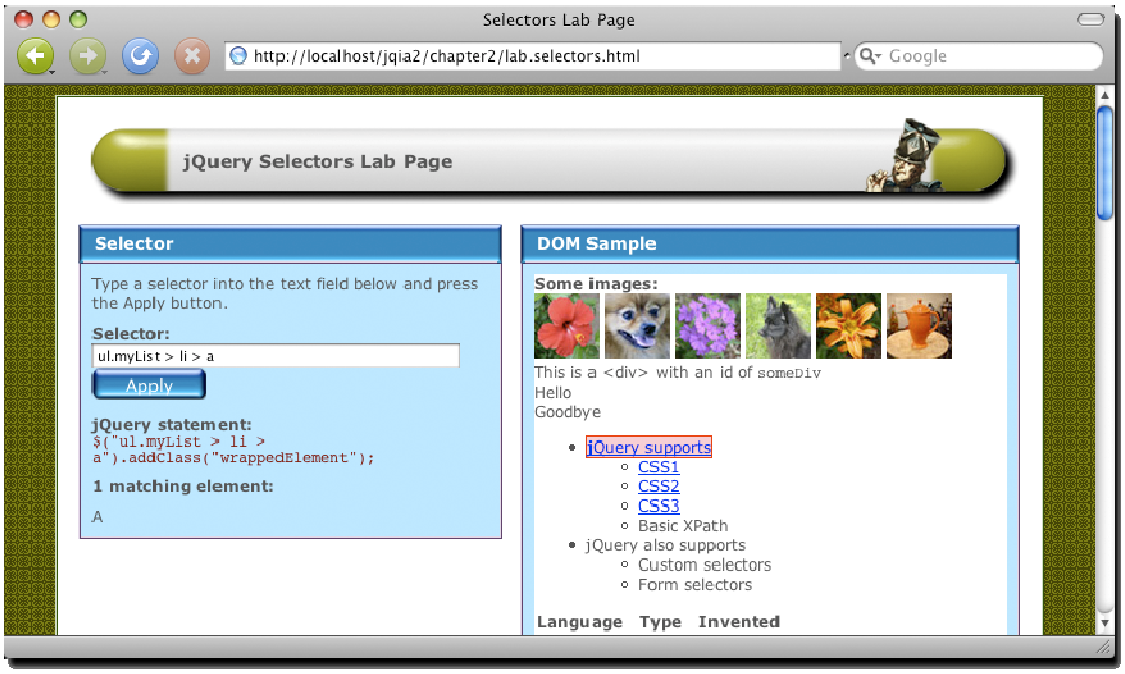


Figure 2.4 With the selector `ul.myList > li > a`, only the direct children of parent nodes are matched

Attribute selectors are also extremely powerful. Say that we want to attach a special behavior only to links that point to locations outside our site. Let's take another look at that portion of the Lab example that we previously examined:

```
<li><a href="http://jquery.com">jQuery supports</a>
    <ul>
      <li><a href="css1">CSS1</a></li>
      <li><a href="css2">CSS2</a></li>
      <li><a href="css3">CSS3</a></li>
      <li>Basic XPath</li>
    </ul>
</li>
```

What makes the link pointing to an external site unique is the presence of the string `http://` at the beginning of the value of the link's `href` attribute. We could select links with an `href` value starting with `http://` with the following selector:

```
a[href^='http://']
```

This matches all links with an `href` value beginning with the exact string `http://`. The caret character (^) is used to specify that the match is to occur at the beginning of a value. As this is the same

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

character used by most regular expression processors to signify matching at the beginning of a candidate string, it should be easy to remember.

Visit the Lab page again (from which the example HTML fragment was lifted), type `a[href^='http://']` into the text box, and click Apply. Note how only the jQuery link is highlighted.

There are other ways to use attribute selectors. To match an element that possesses a specific attribute, regardless of its value, we can use

```
form[method]
```

This matches any `<form>` element that has an explicit `method` attribute.

To match a specific attribute value, we use something like

```
input[type='text']
```

This selector matches all input elements with a type of `text`.

We've already seen the "match attribute at beginning" selector in action. Here's another:

```
div[title^='my']
```

This selects all `<div>` elements with `title` attributes whose value begins with `my`.

What about an "attribute ends with" selector? Coming right up:

```
a[href$='.pdf']
```

This is a useful selector for locating all links that reference PDF files.

And there's a selector for locating elements whose attributes contain arbitrary strings anywhere in the attribute value:

```
a[href*='jquery.com']
```

As we would expect, this selector matches all `<a>` elements that reference the jQuery site.

Table 2.2 shows the basic CSS selectors that we can use with jQuery.

Table 2.2 The basic CSS selectors supported by jQuery

Selector	Description
*	Matches any element.
E	Matches all elements with tag name E.
E F	Matches all elements with tag name F that are descendents of E.
E>F	Matches all elements with tag name F that are direct children of E.
E+F	Matches all elements with tag name F that are immediately preceded by sibling E.
E~F	Matches all elements with tag name F preceded by any sibling E.
E.C	Matches all elements with tag name E with class name C. Omitting E is the same as `*.C`.
E#I	Matches all elements with tag name E with the id of I. Omitting E is the same as `*.I`.
E[A]	Matches all elements with tag name E that have attribute A of any value.
E[A=V]	Matches all elements with tag name E that have attribute A whose value is exactly V.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

E[A^=V] Matches all elements with tag name E that have attribute A whose value starts with V.

E[A$=V] Matches all elements with tag name E that have attribute A whose value ends with V.

E[A!=V] Matches all elements with tag name E that have attribute A whose value does not match the value V, or that lack attribute A completely.

E[A*=V] Matches all elements with tag name E that have attribute A whose value contains V.

With all this knowledge in hand, head over to the Selectors Lab page, and spend some more time running experiments using selectors of various types from table 2.2. Try to make some targeted selections like the elements containing the text *Hello* and *Goodbye* (hint: you'll need to use a combination of selectors to get the job done).

As if the power of the selectors that we've discussed so far isn't enough, there are some more options that give us an even finer ability to slice and dice the page.

2.1.4 Selecting by position

Sometimes, we'll need to select elements by their position on the page or in relation to other elements. We might want to select the first link on the page, or every other paragraph, or the last list item of each list. jQuery supports mechanisms for achieving these specific selections.

For example, consider

```
a:first
```

This format of selector matches the first <a> element on the page.

What about picking every other element?

```
p:odd
```

This selector matches every odd paragraph element. As we might expect, we can also specify that evenly ordered elements be selected with

```
p:even
```

Another form

```
ul li:last-child
```

chooses the last child of parent elements. In this example, the last <li> child of each <ul> element is matched.

There are a whole slew of these selectors, some defined by CSS, others specific to jQuery, and they can provide surprisingly elegant solutions to sometimes tough problems. The CSS Specification refers to these types of selectors as *pseudo-classes*, but jQuery has adopted the crisper term *filters*, as each of these selectors filter a base selector. These filter selectors are easy to spot as they all begin with the colon (:) character, and remember, if you omit any base selector, it defaults to *.

See table 2.3 for a list of these positional filter selectors.

Table 2.3 The positional filter selectors supported by jQuery

Selector	Description
`:first`	The first match within the context. `li a:first` returns the first link that is a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

	descendant of a list item.
`:last`	The last match within the context. `li a:first` returns the last link that is a descendant of a list item.
`:first-child`	The first child element. `lifers-child` returns the first list item of each list.
`:last-child`	The last child element. `li:first-child` returns the last list item of each list.
`:only-child`	Returns all elements that have no siblings.
`:nth-child(n)`	The *n*th child element. `li:nth-child(2)` returns the second list item of each list.
`:nth-child(even|odd)`	Even or odd children. `li:nth-child(even)` returns the even list items of each list.
`:nth-child(Xn+Y)`	The *n*th child element computed by the supplied formula. If Y is 0, it may be omitted. `li:nth-child(3n)` returns every 3rd list item, whereas `li:nth-child(5n+1)` returns the item after every 5th element.
`:even`	Even elements. `li:even` returns every even list item.
`:odd`	Odd elements. `li:odd` returns every even list item.
`:eq(n)`	The nth matching element.
`:get(n)`	Matching elements after and excluding the *n*th matching element.
`:lt(n)`	Matching elements before and excluding the *n*th matching element.

There is one quick gotcha (isn't there always?). The `nth-child` selector starts counting from 1, whereas the other selectors start counting from 0. For CSS compatibility, `nth-child` starts with 1, but the jQuery custom selectors follow the more common programming convention of starting at 0. With some use, it becomes second nature to remember which is which, but it may be a bit confusing at first.

Let's dig in some more.

Consider the following table from the Lab's sample DOM, containing a list of some programming languages and some basic information about them:

```
<table id="languages">
  <thead>
    <tr>
      <th>Language</th>
      <th>Type</th>
      <th>Invented</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Java</td>
      <td>Static</td>
      <td>1995</td>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
    </tr>
    <tr>
      <td>Ruby</td>
      <td>Dynamic</td>
      <td>1993</td>
    </tr>
    <tr>
      <td>Smalltalk</td>
      <td>Dynamic</td>
      <td>1972</td>
    </tr>
    <tr>
      <td>C++</td>
      <td>Static</td>
      <td>1983</td>
    </tr>
  </tbody>
</table>
```

Let's say we want to get all of the table cells that contain the names of programming languages. Because they are all the first cells in their row, we can use

```
table#languages td:first-child
```

We can also easily use

```
table#languages td:nth-child(1)
```

But the first syntax would be considered pithier and more elegant.

To grab the language type cells, we change the selector to use `:nth-child(2)`, and for the year they were invented, we use `:nth-child(3)` or `:last-child`. If we want the absolute last table cell (the one containing the text 1983), we'd use `td:last`. Also, whereas `td:eq(2)` returns the cell containing the text 1995, `td:nth-child(2)` returns all of the cells giving programming language types. Again, remember that `:eq` is 0-based, but `:nth-child` is 1-based.

Before we move on, head back over to the Selectors Lab, and try selecting entries two and four from the list. Then, try to find three different ways to select the cell containing the text *1972* in the table. Also, try and get a feel for the difference between the `nth-child` selectors and the absolute position selectors.

Even though the CSS selectors we've examined so far are incredibly powerful, let's discuss ways of squeezing even more power out of jQuery's selectors.

2.1.5 Using CSS and custom jQuery filter selectors

The CSS selectors that we have seen so far give us a great deal of power and flexibility to match the desired DOM elements, but there are even more selectors defined that give us further ability to filter the selections.

As an example, we might want to select all check boxes that have been checked by the user. You might be tempted to try something along the lines of:

```
$('input[type=checkbox][checked]')
```

But trying to match by attribute will only check the *initial* state of the control as specified in the HTML markup. What we really want to check is the real-time active state of the controls. CSS offers a selector, `:checked`, that filters the set of matched elements to those that are in checked state. For example,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

whereas the selector `input` selects all `<input>` elements, the selector `input:checked` narrows the search to only `<input>` elements that are checked.

As if that wasn't enough, jQuery provides a whole handful of powerful custom filter selectors, not specified by CSS, that make identifying target elements even easier. For example, the custom `:checkbox` selector identifies all checkbox elements. Combining these custom selectors can be powerful; consider `:checkbox:checked` Or `:radio:checked`.

As we discussed earlier, jQuery supports the CSS filter selectors and also defines a number of custom selectors. They are described in table 2.4.

Table 2.4 The CSS and custom jQuery filter selectors

Selector	Description	CSS?
`:animated`	Selects only elements that are currently under animated control. Chapter 5 will cover animations and effects.	
`:button`	Selects only button elements (`input[type=submit]`, `input[type=reset]`, `input[type=button]`, or `button`).	
`:checkbox`	Selects only check box elements (`input[type=checkbox]`).	
`:checked`	Selects only checkboxes or radio elements in checked state.	X
`:contains(food)`	Selects only elements containing the text `food`.	
`:disabled`	Selects only element in disabled state.	X
`:enabled`	Selects only elements in enabled state.	X
`:file`	Selects only file input elements (`input[type=file]`).	
`:has(selector)`	Selects only elements that contain at least one element that matches the specified selector.	
`:header`	Selects only elements that are headers; for example: `<h1>` through `<h6>` elements.	
`:hidden`	Selects only elements that are hidden.	
`:image`	Selects only image input elements (`input[type=image]`).	
`:input`	Selects only form elements (`input`, `select`, `text area`, `button`).	
`:not(selector)`	Negates the specified selector.	X
`:parent`	Selects only elements that have children (including text), but not empty elements.	
`:password`	Selects only password elements (`input[type=password]`).	

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`:radio`	Selects only radio elements (`input[type=radio]`).
`:reset`	Selects only reset buttons (`input[type=reset]` or `button[type=reset]`).
`:selected`	Selects only `<option>` elements that are in selected state.
`:submit`	Selects ply submit buttons (`button[type=submit]` or `input[type=submit]`).
`:text`	Selects only text elements (`input[type=text]`).
`:visible`	Selects only elements that are visible.

Many of these CSS and custom jQuery filter selectors are form-related, allowing us to specify, rather elegantly, a specific element type or state. We can combine selector filters too. For example, if we want to select only enabled and checked check boxes, we could use

```
:checkbox:checked:enabled
```

Try out as many of these filters as you like in the Selectors Lab until you feel that you have a good grasp of their operation.

These filters are an immensely useful addition to the set of selectors at our disposal, but what about the *inverse* of these filters?

USING THE :NOT FILTER

If we want to negate a selector, let's say to match any input element that's *not* a check box, we use the `:not` filter.

For example, to select non-check box `<input>` elements, we use

```
input:not(:checkbox)
```

But be careful! It's easy to go astray and get some unexpected results!

For example, let's say that you wanted to select all images except for those whose `src` attribute contains the text "dog". You might quickly concoct the following selector:

```
$(':not(img[src*="dog"])')
```

But when you use this selector, you find that not only did you get all the image elements that don't reference "dog" in their `src`, you also get every element in the DOM that isn't an image element!

Whoops! Remember that when a base selector is omitted, it defaults to *, so our errant selector actually reads as "fetch all elements that aren't images that reference 'dog' in their src attributes", when what we intended was "fetch all image elements that don't reference 'dog' in their src attributes", which would be expressed as:

```
$('img:not([src*="dog"])')
```

Again, use the Lab page to conduct experiments until you are comfortable with how to use the `:not` filter to invert selections.

jQuery also adds a custom filter that we'll find helps us make selections using parent-child relationships.

jQuery 1.2 Warning

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

If you are still using jQuery 1.2, be aware that the filter selectors such as `:not()` and `:has()` can only accept other filter selectors. They cannot be passed selectors that contain element expressions. This restriction was lifted in jQuery 1.3.

USING THE :HAS FILTER

As we saw earlier, CSS defines a useful selector for selecting elements that are descendants of particular parents. For example, the selector:

```
div span
```

would select all `<span>` elements that are descendants of `<div>` elements.

But what if we wanted the opposite? What if we wanted to select all `<div>` elements that contained `<span>` elements?

That's the job of the `:has()` filter. Consider:

```
div:has(span)
```

which selects the `<div>` ancestor elements, as opposed to the `<span>` descendent elements.

This can be a powerful mechanism when we get to the point where we want to select elements that represent complex constructs. For example, let's say that we want to find which table row contains a particular image element that we can uniquely identify using its `src` attribute. We might use a selector such as:

```
$('tr:has(img[src$=puppy.png])')
```

which would return any table row element containing the identified image anywhere in its descendant hierarchy.

Trust that this, along with the other jQuery filters, will play a large part in the code we examine going forward.

As we've seen, jQuery gives us a large toolset with which to select existing elements on a page for manipulation via the jQuery methods, which we'll begin to examine in chapter 3. But before we look at the manipulation methods, let's see how to use the `$()` function to create *new* HTML elements.

2.2 Generating new HTML

Sometimes, we'll want to generate new fragments of HTML to insert into the page. With jQuery, it's a simple matter because, as we saw in chapter 1, the `$()` function can create elements from HTML strings in addition to selecting existing page elements. Consider

```
$("<div>Hello</div>")
```

This expression creates a new `<div>` element ready to be added to the page. We can run any jQuery methods that we could run on wrapped element sets of existing elements on the newly created fragment. This may not seem impressive on first glance, but when we throw event handlers, Ajax, and effects into the mix (as we will in the upcoming chapters), we'll see how it can come in mighty handy.

Note that if we want to create an empty `<div>` element, we can get away with a shortcut:

```
$("<div>")    #1
```

#1 Identical to $(" ") and $(" ")

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

It's almost embarrassingly easy to create such simple HTML elements, and thanks to the chain ability of jQuery methods, creating more complex elements isn't much harder. Let's say that we want to create an image element complete with multiple attributes, some styling, and let's make it clickable to boot!

Take a look at the code of Listing 2.1.

Listing 2.1 Dynamically creating a full-featured <img> element

```
$('<img>')                                                    #1
  .attr({                                                     #2
    src: 'images/little.bear.png',
    alt: 'Little Bear',
    title:'I woof in your general direction'
  })
  .css({                                                      #3
    cursor: 'pointer',
    border: '1px solid black',
    padding: '12px 12px 20px 12px',
    backgroundColor: 'white'
  })
  .appendTo('body')                                           #4
  .click(function(){                                          #5
    alert($(this).attr('title'));
  });
```

#1 Creates the basic <img> element
#2 Assigns various attributes
#3 Styles the image
#4 Attaches the element to the document
#5 Establishes a click handler

Cueballs in code and text

With a single jQuery statement, we have created the basic <img> element (#1), given it important attributes such as its source, alternate text and flyout title (#2), styled it to look like a printed photograph (#3), attached it to the DOM tree (#4), and established an event handler that issues an alert (garnered from the image's title) when the image is clicked (#5).

That's a pretty hefty statement – which we spread across multiple lines, and with logical indentation, for readability – but it also does a heck of a lot. Such statements are not uncommon in jQuery-enabled pages, and if you find it a bit overwhelming, not to worry, we'll be covering every method used in this statement over the next few chapters, and writing such compound statements will be second nature before much longer.

Figure 2.5 shows the result of this code, both when the page is first loaded (top), and after the image has been clicked upon (bottom).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

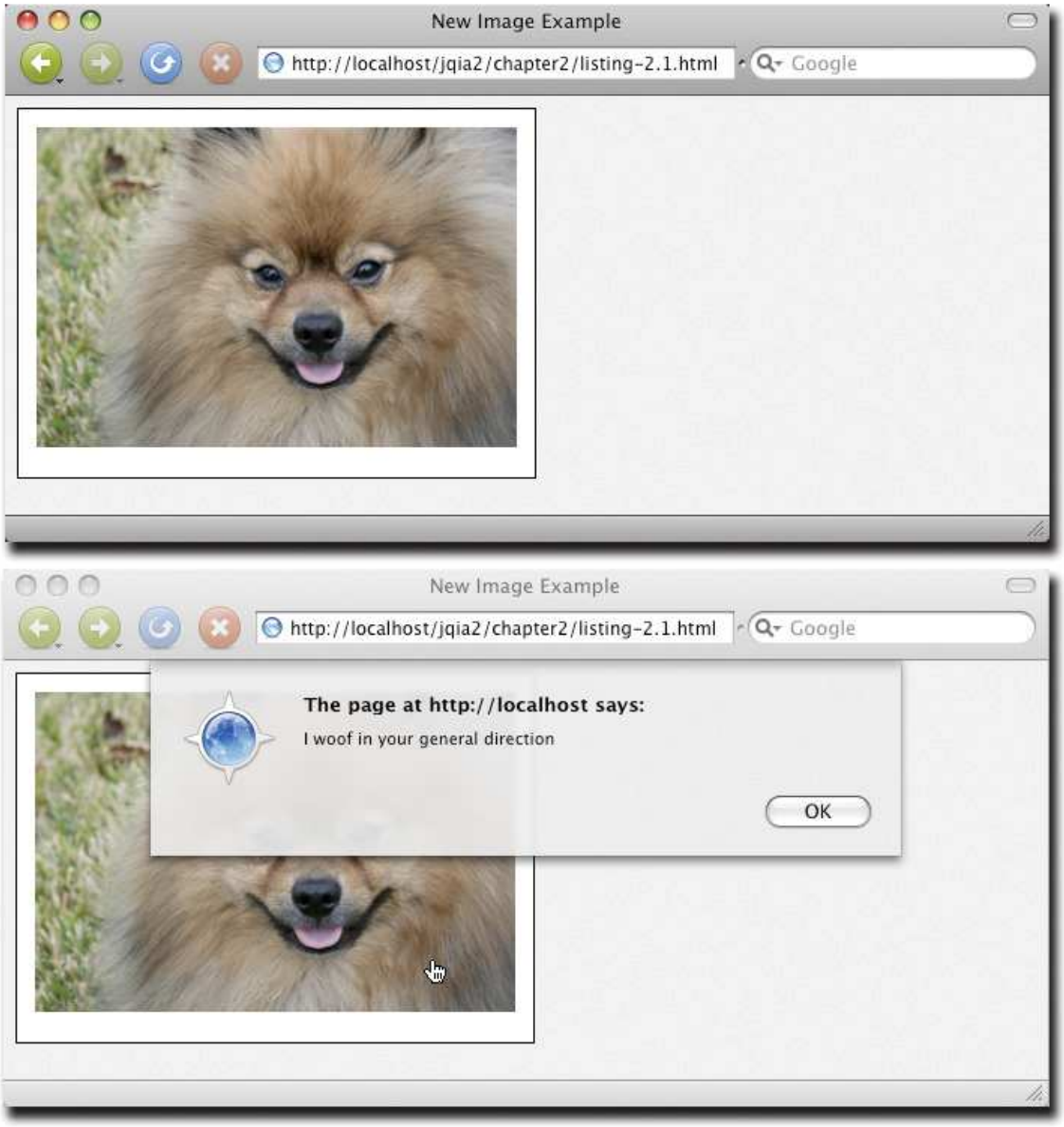


Figure 2.5 Creating complex elements on the fly, including this image, is easy as pie

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

The full code for this example can be found in the book's project code at `chapter2/listing-2.1.html`.

Up until now, we've applied wrapper methods to the entire wrapped set as created by the jQuery function when we pass a selector to it. But there may be times when we want to further manipulate that set before acting upon it.

2.3 Managing the wrapped element set

Once we've got a set of wrapped elements that we either identified by using a selector to match existing DOM elements, or that we created as new elements using HTML snippets (or a combination of both), we're ready to manipulate those elements using the powerful set of jQuery methods. We'll start looking at those methods in the next chapter, but what if we're not quite ready yet? What if we want to further refine the set of elements wrapped by the jQuery function?

In this section, we'll explore the many ways that we can refine, extend, or subset the set of wrapped elements that we wish to operate upon.

In order to visually help you in this endeavor, another Lab Page has been set up and included in the downloadable project code for this chapter: the jQuery Operations Lab, which you will find in `chapter2/lab.operations.html`. This page, which looks a lot like the Selectors Lab we employed earlier in this chapter, is shown in figure 2.6.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

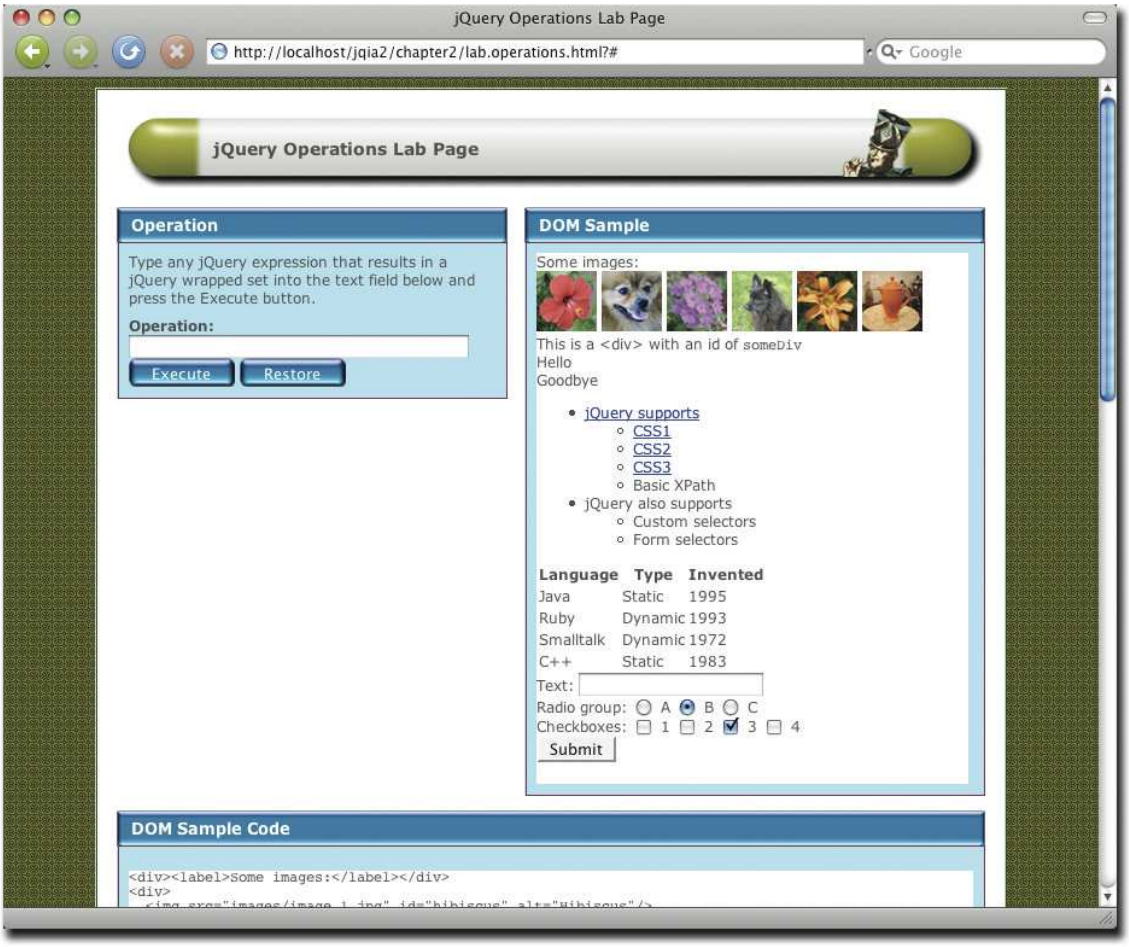


Figure 2.6 The jQuery Operations Lab helps us see how wrapped sets can be created and managed.

This new lab page not only looks like the Selectors Lab, it also operates in a similar fashion. Except in *this* Lab, rather than typing a selector, we can type in any *complete* jQuery operation that results in a wrapped set. The operation is executed in the context of the DOM Sample, and, as with the Selectors Lab, the results are displayed.

In a sense, the jQuery Operations Lab is a more general case of the Selectors Lab. Where the latter only allowed us to enter a single selector, the jQuery Operations Lab allows us to enter any expression

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

that results in a jQuery wrapped set. Because of the way jQuery chaining works, this expression can also include wrapper methods, making this a powerful Lab for examining the operations of jQuery.

Be aware that you need to enter *valid* syntax, as well as expressions that result in a jQuery wrapped set. Otherwise, you're going to be faced with a handful of unhelpful JavaScript errors.

Just to get a feel for the Lab, display it in your browser and enter the text
```
$('img').hide()
```
into the Operation field, then click the Execute button.

This operation is executed within the context of the DOM sample and you'll see how the images disappear from the sample. After any operation, to restore the DOM sample to its original condition, click the Restore button.

We'll see this new Lab in action as we work our way through the sections that follow, and you might even find it helpful in later chapters to test various jQuery operations.

2.3.1 Determining the size of the wrapped set

We mentioned before that the set of jQuery wrapped elements acts a lot like an array. This mimicry includes a `length` property, just like JavaScript arrays, that contains the number of wrapped elements.

Should we wish to use a method rather than a property, jQuery also defines the `size()` method, which returns the same information.

Consider the following statement:
```
$('#someDiv')
  .html('There are '+$('a').size()+' link(s) on this page.');
```
The jQuery expression embedded in the statement matches all elements of type <a> and returns the number of matched elements using the `size()` method. This is used to construct a text string, which is set as the content of an element with `id` of `someDiv` using the `html()` method (which we'll see in the next chapter).

The formal syntax of the `size()` method is as follows:

Method syntax: size

`size()`

Returns the count of elements in the wrapped set

Parameters

none

Returns

The element count

OK, so now we know how many elements we have. What if we want to access them directly?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

2.3.2 Obtaining elements from the wrapped set

Usually, once we have a wrapped set of elements, we can use jQuery methods to perform some sort of operation upon them as a whole; for example, hiding them all with the `hide()` method. But there may be times when we want to get our grubby little hands on a direct reference to an element or elements to perform raw JavaScript operations upon them.

Because jQuery allows us to treat the wrapped set as a JavaScript array, we can use simple array indexing to obtain any element in the wrapped list by position. For example, to obtain the first element in the set of all `<img>` elements with an `alt` attribute on the page, we can write

```
var imgElement = $('img[alt]')[0]
```

If we prefer to use a method rather than array indexing, jQuery defines the `get()` method for that purpose.

Method syntax: get

`get(index)`

Obtains one or all of the matched elements in the wrapped set. If no parameter is specified, all elements in the wrapped set are returned in a JavaScript array. If an `index` parameter is provided, the indexed element is returned.

Parameters

`index`	(Number) The index of the single element to return. If omitted, the entire set is returned in an array.

Returns

A DOM element or an array of DOM elements.

The fragment

```
var imgElement = $('img[alt]').get(0)
```

is equivalent to the previous example that used array indexing.

The `get()` method can also be used to obtain a plain JavaScript array of all the wrapped elements. Consider:

```
var allLabeledButtons = $('label+button').get();
```

This statement collects all the `<button>` elements on the page that are immediately preceded by `<label>` elements into a jQuery wrapper and then creates a JavaScript array of those elements to assign to the `allLabeledButtons` variable.

We can use an inverse operation to find the index of a particular element in the wrapped set. Let's say for some reason we want to know the ordinal index of an image with the `id` of `findMe` within the entire set of images in a page. We can obtain this value with

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
var n = $('img').index($('img#findMe')[0]);
```

The syntax of the `index()` method is as follows:

Method syntax: index

`index(element)`

Finds the passed element in the wrapped set and returns its ordinal index within the set. If the element isn't resident in the set, the value -1 is returned.

Parameters

`element` (Element) A reference to the element whose ordinal value is to be determined.

Returns

The ordinal value of the passed element within the wrapped set or -1 if not found.

Now, rather than obtaining direct references to elements, how would we go about adjusting the set of elements that are wrapped?

2.3.3 Slicing and dicing the wrapped element set

Once we have a wrapped element set, we may want to augment that set by adding to it or by reducing the set to a subset of the originally matched elements. jQuery gives us a large collection of methods to manage the set of wrapped elements. First, let's look at adding elements to a wrapped set.

ADDING MORE ELEMENTS TO A WRAPPED SET

Often, we may find ourselves in a situation where we want to add more elements to an existing wrapped set. This capability is most useful when we want to add more elements after applying some method to the original set. Remember, jQuery chaining makes it possible to perform an enormous amount of work in a single statement.

But first, let's examine a simpler situation. Let's say that we want to match all `<img>` elements that have either an `alt` or a `title` attribute. The powerful jQuery selectors allow us to express this as a single selector, such as

```
$('img[alt],img[title]')
```

But to illustrate the operation of the `add()` method, we could match the same set of elements with

```
$('img[alt]').add('img[title]')
```

Using the `add()` method in this fashion allows us to chain a bunch of selectors together into an *or* relationship, creating the union of the elements that satisfy both of the selectors.

Methods such as `add()` are also significant, and more flexible than aggregate selectors, within jQuery method chains because they do not actually augment to original wrapped set, but create a *new* wrapped set with the result. We'll see in just a bit how this can be extremely useful in conjunction with methods

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

such as `end()` (which we'll examine in section 2.3.6) that can be used to "back out" operations that augment original wrapped sets.

The syntax of the `add()` method: .

Method syntax: add

`add(expression)`

Creates a copy of the wrapped set and adds elements, specified by the `expression` parameter, to the new set. The expression can be a selector, an HTML fragment, a DOM element, or an array of DOM elements.

Parameters

`expression`	(String|Element|Array) Specifies what is to be added to the matched set. This parameter can be a jQuery selector, in which case any matched elements are added to the set. If an HTML fragment, the appropriate elements are created and added to the set. If a DOM element or an array of DOM elements, they are added to the set.

Returns

A copy of the original wrapped set with the additional elements.

Bring up the jQuery Operations Lab page in your browser, and enter the expression:

```
$('img[alt]').add('img[title]')
```

and click the Execute button.

This will execute the jQuery operation and result in the selection of all images with either an `alt` or `title` attribute.

Inspecting the HTML source for the DOM Sample reveals that all the images depicting flowers have `alt` attributes, the puppy images have `title` attributes, and the coffee pot image has neither. Therefore, we should expect that all images but the coffee pot would become part of the wrapped set. Figure 2.7 shows a screen capture of the results.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Figure 2.7 The expected image elements, those with an `alt` or `title` attribute, have been matched by the jQuery expression

We can see that five of the six images (all but the coffee pot) were added to the wrapped set. The red outline may be a bit hard to see in the print version of this book with grayscale figures, but if you have downloaded the project (which you should have) and are using it to follow along (which you should be), it's very evident.

Now let's take a look at a more realistic use of the `add()` method. Let's say that we want to apply a thick border to all `<img>` elements with `alt` attributes, and then apply a level of transparency to all `<img>` elements with either `alt` or `title` attributes. The comma operator (,) of CSS selectors won't help us with this one because we want to apply an operation to a wrapped set and *then* add more elements to it before applying another. We could easily accomplish this with multiple statements, but it would be more efficient and elegant to use the power of jQuery chaining to accomplish the task in a single expression, such as

```
$('img[alt]').addClass('thickBorder').add('img[title]').addClass('seeThrough')
```

In this statement, we create a wrapped set of all `<img>` elements with `alt` attributes, apply a predefined class that applies a thick border, add the `<img>` elements that have `title` attributes, and finally apply a class that establishes a level of transparency to the newly augmented set.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Enter this statement into the jQuery Operations Lab page (which has predefined the referenced classes), and view the results as shown in figure 2.8.

Figure 2.8 jQuery chaining allows us to perform complex operations in a single statement, as seen by these results

In these results, we can see that the flower images (those with `alt`) have thick borders, and all images but the coffee pot (the only one with neither an `alt` nor a `title`) are faded as a result of applying an opacity rule.

The `add()` method can also be used to add elements to an existing wrapped set given direct references to those elements. Passing an element reference, or an array of element references, to the `add()` method adds the elements to the wrapped set. If we assume that we have an element reference in a variable named `someElement`, it could be added to the set of all images containing an `alt` property with

```
$('img[alt]').add(someElement)
```

As if that weren't flexible enough, the `add()` method not only allows us to add existing elements to the wrapped set, we can also use it to add new elements by passing it a string containing HTML markup. Consider

```
$('p').add('<div>Hi there!</div>')
```

This fragment creates a wrapped set of all <p> elements in the document, and then creates a new , and adds it to the wrapped set. Note that doing so only adds the new element to the wrapped set; no action has been taken in this statement to add the new element to the DOM. We might then use the jQuery `appendTo()` method (patience, we'll be talking about such methods soon enough) to append the elements we selected, as well as the newly created HTML, to some part of the DOM.

Augmenting the wrapped set with `add()` is easy and powerful, but now let's look at the jQuery methods that let us *remove* elements from a wrapped set.

HONING THE CONTENTS OF THE WRAPPED SET

We saw that it's a simple matter to create wrapped sets from multiple selectors chained together with the `add()` method to form an *or* type of relationship. It's also possible to chain selectors together to form an

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

except relationship by employing the `not()` method. This is similar to the `:not` filter selector we discussed earlier, but can be employed in a similar fashion to the `add()` method to remove elements from the wrapped set anywhere within a jQuery chain of methods.

Let's say that we want to select all `<img>` elements in a page that sport a `title` attribute *except* for those that contain the text "puppy" in the `title` attribute value. We could come up with a single selector that expresses this condition (namely `img[title]:not([title*=puppy])`), but for the sake of illustration, let's pretend that we forgot about the `:not` filter. By using the `not()` method, which removes any elements from a wrapped set that match the passed selector expression, we can express an *except* type of relationship. To perform the described match, we can write

```
$('img[title]').not('[title*=puppy]')
```

Type this expression into the jQuery Operations Lab page, and execute it. You'll see that only the tan puppy image has the highlight applied. The black puppy, which is included in the original wrapped set because it possesses a `title` attribute, is removed by the `not()` invocation because its `title` contains the text "puppy".

Method syntax: not

`not(expression)`

Creates a copy of the wrapped set and removes elements from the new set according to the value of the `expression` parameter. If the parameter is a jQuery selector, any matching elements are removed. If an element reference is passed, that element is removed from the set.

Parameters

`expression`	(String|Element|Array) A jQuery selector expression, element reference, or array of element references defining what is to be removed from the wrapped set.

Returns

A copy of the original wrapped set without the removed elements.

As with `add()`, the `not()` method can also be used to remove individual elements from the wrapped set by passing a reference to an element or an array of element references. The latter is interesting and powerful because, remember, any jQuery wrapped set can be used as an array of element references.

At times, we may want to filter the wrapped set in ways that are difficult or impossible to express with a selector expression. In such cases, we may need to resort to programmatic filtering of the wrapped set items. We could iterate through all the elements of the set and use the `not(element)` method to remove the specific elements that do not meet our selection criteria. But the jQuery team didn't want us to have to resort to doing all that work on our own and so have defined the `filter()` method.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

The `filter()` method, when passed a function, invokes that function for each wrapped element and removes any element whose function invocation returns the value `false`. Each invocation has access to the current wrapped element via the function context (`this`) in the body of the filtering function.

For example, let's say that, for some reason, we want to create a wrapped set of all `<td>` elements that contain a numeric value. As powerful as the jQuery selector expressions are, such a requirement is impossible to express using them. For such situations, the `filter()` method can be employed, as follows:

```
$('td').filter(function(){return this.innerHTML.match(/^\d+$/)})
```

This jQuery expression creates a wrapped set of all `<td>` elements and then invokes the function passed to the `filter()` method for each, with the current matched elements as the `this` value for the invocation. The function uses a regular expression to determine if the element content matches the described pattern (a sequence of one or more digits), returning `false` if not. Every element whose filter function invocation returns `false` is not included in the returned wrapped set.

Method syntax: filter

`filter(expression)`

Creates a copy of the wrapped set and filters out elements from the new set using a passed selector expression, or a filtering function.

Parameters

`expression`	(String|Function) Specifies a jQuery selector used to remove all elements that do not match from the wrapped set, or a function that makes the filtering decision. This function is invoked for each element in the set, with the current element set as the function context for that invocation. Any element that returns an invocation of `false` is removed from the set.

Returns

A copy of the original wrapped set without the filtered elements.

Again, bring up the jQuery Operations Lab, type the previous expression in, and execute it. You will see that the table cells for the "Invented" column are the only `<td>` elements that end up being selected.

The `filter()` method can also be used with a passed selector expression. When used in this manner, it operates in the inverse manner than the corresponding `not()` method, removing any elements that do not match the passed selector. This isn't a super-powerful method, as it's usually easier to use a more restrictive selector in the first place, but it can be useful within a chain of jQuery methods. Consider, for example,

```
$('img').addClass('seeThrough').filter('[title*=dog]').addClass('thickBorder')
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

This chained statement selects all images and applies the `seeThrough` class to them and then reduces the set to only those image elements whose `title` attribute contains the string `dog` before applying another class named `thickBorder`. The result is that all the images end up semi-transparent, but only the tan dog gets the thick border treatment.

The `not()` and `filter()` methods give us powerful means to adjust a set of wrapped elements on the fly, based on just about any criteria regarding aspects of the wrapped elements. We can also subset the wrapped set, based on the position of the elements within the set. Let's see which methods allow us to do that.

OBTAINING SUBSETS OF THE WRAPPED SET

Sometimes we may wish to obtain a subset of the wrapped set based upon the position of the elements within the set. jQuery provides a method to do that named `slice()`. This method creates and returns a new set from any contiguous portion, or a slice, of an original wrapped set. The syntax for this method follows:

Method syntax: slice

`slice(begin,end)`

Creates and returns a new wrapped set containing a contiguous portion of the matched set.

Parameters

`begin`	(Number) The zero-based position of the first element to be included in the returned slice.
`end`	(Number) The optional zero-based index of the first element not to be included in the returned slice, or one position beyond the last element to be included. If omitted, the slice extends to the end of the set.

Returns

The newly-created wrapped set.

If we want to obtain a wrapped set that contains a single element from another set, based on its position in the original set, we could employ the `slice()` method, passing the zero-based position of the element within the wrapped set. For example, to obtain the third element, we write

```
$('*').slice(2,3);
```

This statement selects all elements on the page and then generates a new set containing the third element in the matched set.

Note that this is different from `$('*').get(2)`, which returns the third *element* in the wrapped set, not a wrapped set containing the element.

Therefore, a statement such as

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
$('*').slice(0,4);
```

selects all elements on the page and then creates a set containing the first four elements.

To grab elements from the end of the wrapped set, the statement

```
$('*').slice(4);
```

matches all elements on the page and then returns a set containing all but the first four elements.

TRANSLATING ELEMENTS OF THE WRAPPED SET

We'll often want to perform transformations on the elements of the wrapped set. For example, what if we wanted to collect all the `id` values, of the wrapped element, or the values of a wrapped set of form elements?

The `map()` method comes in might handy for such occasions:.

Method syntax: map

`map(callback)`

Invokes the callback function for each element in the wrapped set, and collects the returned values into a jQuery object instance.

Parameters

`callback`	(Function) A callback function that is invoked for each element in the wrapped set. Two parameters are passed to this function: the zero-based index of the element within the set, and the element itself. The element is also established as the function context (the `this` keyword).

Returns

The wrapped set of translated values.

For example to collect all the `id` values of all images on the page into a JavaScript array:

```
var allIds = $('div').map(function(){
  return (this.id=='d3') ? null : this.id;
}).get();
```

If any invocation of the callback function returns `null`, no corresponding entry is made in the returned set.

TRAVERSING THE WRAPPED SET ELEMENTS

While the `map()` method is useful for iterating over the elements of the wrapped set in order to collect values or translate the elements in some other way, we'll have many occasions where we'll want to iterate over the elements for more general purposes.

For these occasions the jQuery `each()` method is invaluable.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Method syntax: each

`each(iterator)`

Traverses all elements in the matched set invoking the passed iterator function for each.

Parameters

`iterator`	(Function) A function called for each element in the matched set. Two parameters are passed to this function: the zero-based index of the element within the set, and the element itself. The element is also established as the function context (the `this` keyword).

Returns

The wrapped set.

An example of using this method could be to easily set a property value onto all elements in a matched set. For example, consider:

```
$('img').each(function(n){
  this.alt='This is image['+n+'] with an id of '+this.id;
});
```

This statement will invoke the passed function for each image element on the page, modifying its `alt` property using the order of the element and its `id` value.

And we're not done yet! jQuery also gives us the ability to obtain subsets of a wrapped set based on the *relationship* of the wrapped items with other elements in the DOM. Let's see how.

2.3.4 Getting wrapped sets using relationships

jQuery allows us to get new wrapped sets from an existing set, based on the hierarchical relationships of the wrapped element to the other elements within the HTML DOM.

Table 2.5 shows these methods and their descriptions. Most of these methods accept an optional selector expression that any elements must match in order to be collected into the new set. If no such selector parameter is passed, all eligible elements are selected.

Table 2.5 Methods to obtain new wrapped set based on relationships

Method	Description
`children(expr)`	Returns a wrapped set consisting of all unique children of the wrapped elements.
`closest(expr)`	Returns a wrapped set containing the single nearest ancestor that matches the passed expression; if any.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`contents()`	Returns a wrapped set of the contents of the elements, which may include text nodes, in the wrapped set. (Frequently used to obtain the contents of `<iframe>` elements.)
`next(expr)`	Returns a wrapped set consisting of all unique next siblings of the wrapped elements.
`nextAll(expr)`	Returns a wrapped set containing all the following siblings of the wrapped elements.
`offsetParent()`	Returns a wrapped set containing the closest relatively or absolutely positioned parent of the first element in the wrapped set.
`parent(expr)`	Returns a wrapped set consisting of the unique direct parents of all wrapped elements.
`parents(expr)`	Returns a wrapped set consisting of the unique ancestors of all wrapped elements. This includes the direct parents as well as the remaining ancestors all the way up to, but not including, the document root.
`prev(expr)`	Returns a wrapped set consisting of all unique previous siblings of the wrapped elements.
`prevAll(expr)`	Returns a wrapped set containing all the previous siblings of the wrapped elements.
`siblings(expr)`	Returns a wrapped set consisting of all unique siblings of the wrapped elements.

All of the methods in Table 2.5, with the exception of `contents()` and `offsetParent()` accept a parameter containing a string that can be used to filter the results.

These methods give us a large degree of freedom to select elements from the DOM, based on relationships to the other DOM elements. But we're still not done. Let's see how jQuery deals further with wrapped sets.

2.3.5 Even more ways to use a wrapped set

As if all that were not enough, there are still a few more tricks that jQuery has up its sleeve to let us refine our collections of wrapped objects.

The `find()` method lets us search through the *descendants* of the elements in a wrapped set and returns a new set that contains all elements that match a passed selector expression. For example, given a wrapped set in variable `wrapped Set`, we can get another wrapped set of all citations (`<cite>` elements) within paragraphs that are descendents of elements in the original wrapped set with

```
wrappedSet.find('p cite')
```

Like many other jQuery wrapper methods, the `find()` method's power comes when it's used within a jQuery chain of operations.

Method syntax: find

`find(selector)`

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Returns a new wrapped set containing all elements that are descendants of the elements of the original set that match the passed selector expression.

Parameters

`selector` (String) A jQuery selector that elements must match to become part of the returned set.

Returns

The newly created wrapped set.

This method becomes very handy when we need to constrain a search for descendant elements in the middle of a jQuery method chains where we cannot employ any other context or constraining mechanism.

The last method that we'll examine in this section is one that allows us to test a wrapped set to see if it contains at least one element that matches a given selector expression. The `is()` method returns `true` if at least one element matches the selector, and `false` if not. For example:

```
var has Image = $('*').is('img');
```

This statement sets the value of the `hasImage` variable to `true` if the current DOM has an image element.

Method syntax: is

`is(selector)`

Determines if any element in the wrapped set matches the passed selector expression

Parameters

`selector` (String) The selector expression to test against the elements of the wrapped set

Returns

`true` if at least one element matches the passed selector; `false` if not

This is a highly optimized and fast operation within jQuery and can be used without hesitation in areas where performance is of high concern.

2.3.6 Managing jQuery chains

We've made a big deal about the ability to chain jQuery wrapper methods together to perform a lot of activity in a single statement, and will continue to do so, because it *is* a big deal. This chaining ability not only allows us to write powerful operations in a concise manner, but it also improves efficiency because wrapped sets do not have to be recomputed in order to apply multiple methods to them.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Depending upon the methods used in a method chain, multiple wrapped sets may be generated. For example, using the `clone()` method (which we'll explore in detail in chapter 3) generates a new wrapped set, which creates copies of the elements in the first set. If, once a new wrapped set is generated, we had no way to reference the original set, our ability to construct versatile jQuery method chains would be severely curtailed.

Consider the following statement:

```
$('img').clone().appendTo('#somewhere');
```

Two wrapped sets are generated within this statement: the original wrapped set of all the `<img>` elements in the DOM, and a second wrapped set consisting of copies of those elements. The `clone()` method returns this second set as its result, and it's that set that is operated on by the `appendTo()` method.

But what if we subsequently want to apply a method, such as adding a class name, to the original wrapped set *after* it's been cloned? We can't tack it onto the end of the existing chain; that would affect the clones, not the original wrapped set of images.

jQuery provides for this need with the `end()` method. This method, when used within a jQuery chain, will "back up" to a previous wrapped set and return it as its value so that subsequent operations will apply to that previous set.

Consider

```
$('img').clone().appendTo('#somewhere').end().addClass('beenCloned');
```

The `appendTo()` method returns the set of new clones, but by calling `end()` we back up to the previous wrapped set (the original images), which gets operated on by the `addClass()` method. Without the intervening `end()` method, `addClass()` would have operated on the set of clones.

Method syntax: end

`end()`

Used within a chain of jQuery methods to back up the current wrapped set to a previously returned set

Parameters

none

Returns

The previous wrapped set

It might help to think of the wrapped sets generated during a jQuery method chain as being held on a stack. When `end()` is called, the top-most (most recent) wrapped set is popped from the stack, leaving the previous wrapped set exposed for subsequent methods to operate upon.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Another handy jQuery method that modifies the wrapped set "stack" is `andSelf()`, which merges the two topmost sets on the stack into a single wrapped set.

Method syntax: andSelf

`andSelf()`

Merges the two previous wrapped sets in a method chain

Parameters

none

Returns

The merged wrapped set

Consider:

```
$('div').addClass('a').find('img').addClass('b').andSelf().addClass('c');
```

This statement selects all `<div>` elements, adds class `a` to them, creates a new wrapped set consisting of all `<img>` elements that are descendants of those `<div>` elements, applies the class `b` to them, then creates a third wrapped set that is merger of the `<div>` elements and their descendant `<img>` elements, applying the class `c` to them.

Whew! At the end of it all, the `<div>` elements end up with classes `a` and `c`, while the images that are descendants of those elements are given classes `b` and `c`.

We can see that jQuery provides the means to manage wrapper sets for just about any type of operations that we want to perform upon them

Summary

This chapter focused on creating and adjusting sets of elements (referred in this chapter and beyond as the ***wrapped set***) via the many means that jQuery provides for identifying elements on an HTML page.

jQuery provides a versatile and powerful set of ***selectors***, patterned after the selectors of CSS, for identifying elements within a page document in a concise but powerful syntax. These selectors include the CSS3 syntax currently supported by most modern browsers.

The creation of, and augmentation of, wrapped sets using HTML fragments to create new elements on the fly is another important feature that jQuery provides. These orphaned elements can be manipulated, along with any other elements in the wrapped set, and eventually attached to parts of the page document.

A robust set of methods to adjust the wrapped set in order to refine the contents of the set, either immediately after creation, or midway through a set of chained methods is available. Applying filtering criteria to an already existing set can also easily create new wrapped sets.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

All in all, jQuery gives us a lot of tools to make sure that we can easily and accurately identify the page elements that we wish to manipulate.

In this chapter, we covered a lot of ground without really *doing* anything to the DOM elements of the page. But now that we know how to select the elements that we want to operate upon, we're ready to start adding life to our pages with the power of the jQuery DOM manipulation methods.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

3

Bringing pages to life with jQuery

This chapter covers:

- Getting and setting element attributes
- Storing custom data on elements
- Manipulating element class names
- Setting the contents of DOM elements
- Storing and retrieving custom data on elements
- Getting and setting form element values
- Modifying the DOM tree

Remember those days (luckily, now fading into memory) when fledgling page authors would try to add pizzazz to their pages with counterproductive abominations such as marquees; blinking text; loud background patterns (that inevitably interfered with the readability of the page text); annoying animated GIFs; and, perhaps worst of all, unsolicited background sounds that would play upon page load (and only served to test how fast a user could close down the browser)?

We've come a long way since then.

Today's savvy web developers and designers know better, and use the power given to them by DOM Scripting (what us old-timers might once have called Dynamic HTML (DHTML)) to *enhance* a user's web experience, rather than showcase annoying tricks.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Whether it's to incrementally reveal content, create input controls beyond the basic set provided by HTML, or give users the ability to tune pages to their own liking, DOM manipulation has allowed many a web developer to amaze (not annoy) their users.

On an almost daily basis, many of us come across web pages that do something that makes us say, "Wow! I didn't know you could do that!" And being the commensurate professionals that we are (not to mention insatiably curious about such things), we immediately start looking at the source code to find out *how* they did it.

But rather than having to code up all that script ourselves, we'll find that jQuery provides a robust set of tools to manipulate the DOM, making those types of "Wow!" pages possible with a surprisingly small amount of code. Whereas the previous chapter introduced us to the many ways jQuery lets us select DOM elements into a wrapped set, this chapter puts the power of jQuery to work performing operations on those elements to bring life and that elusive "wow factor" to our pages.

3.1 Manipulating element properties and attributes

Some of the most basic components we can manipulate when it comes to DOM elements are the *properties* and *attributes* assigned to those elements. These properties and attributes are initially assigned to the JavaScript object instances that represent the DOM elements as a result of parsing their HTML markup, and can be changed dynamically under script control.

Let's make sure that we have our terminology and concepts straight.

Properties are intrinsic to JavaScript objects, and each has a name and a value. The dynamic nature of JavaScript allows us to create properties on JavaScript objects under script control. (Appendix A goes into great detail on this concept if you are new to JavaScript.)

Attributes are not a native JavaScript concept, but one that only applies to DOM elements. Attributes represent the values that are specified on the markup of DOM elements.

Consider the following HTML markup for an image element:

```
<img id="myImage" src="image.gif" alt="An image" class="someClass" title="This is an
image"/>
```

In this element's markup, the *tag name* is `img`, and the markup for `id`, `src`, `alt`, `class`, and `title` represents the element's *attributes*, each of which consists of a name and a value. This element markup is read and interpreted by the browser to create the JavaScript object instance that represents this element in the DOM. The attributes are gathered into a list, and this list is stored as a property named, reasonably enough, `attributes` on the DOM element instance. In addition to storing the attributes in this list, the object is given a number of *properties*, including some that represent the attributes of the element's markup.

As such, the attribute values are reflected not only in the `attributes` list, but also in a handful of properties.

Figure 3.1 shows a simplified overview of this process.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

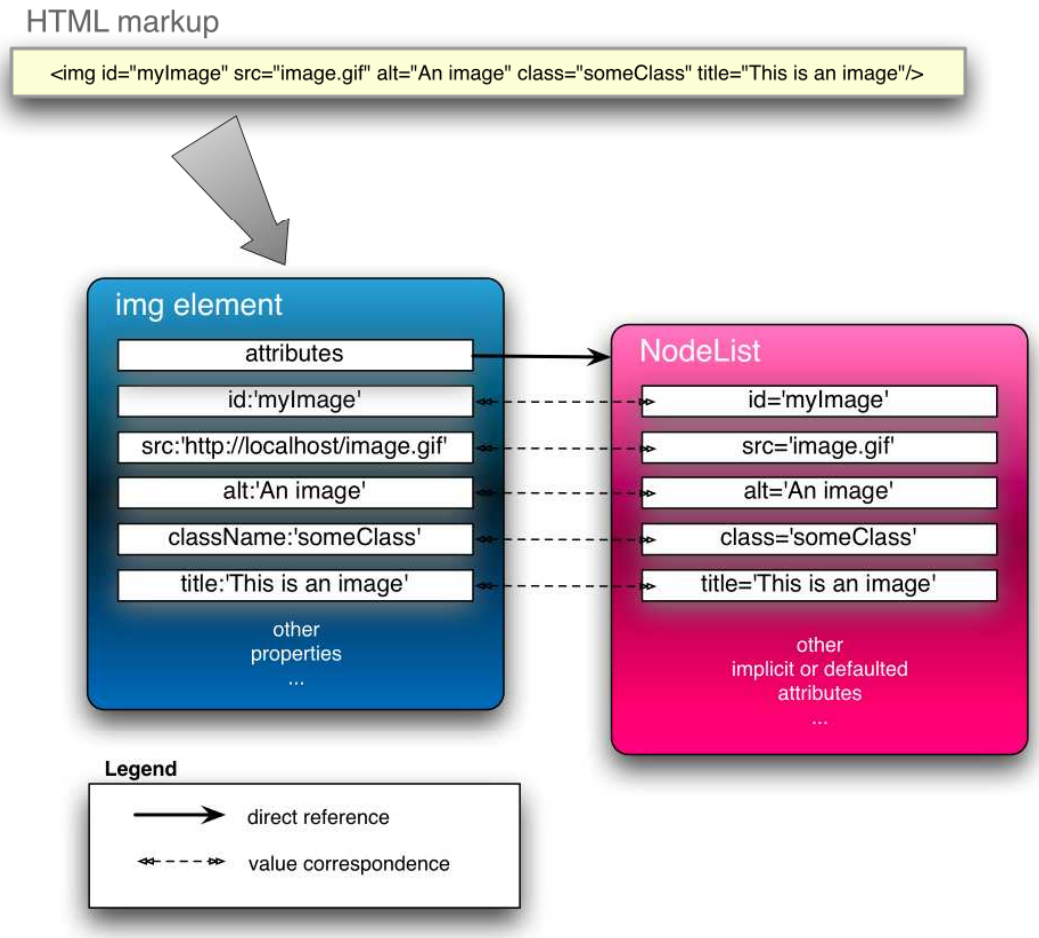


Figure 3.1 HTML markup is translated into DOM elements, including the attributes of the tag and properties created from them

There remains an active connection between the attribute values stored in the `attributes` list, and the corresponding properties. Changing an attribute value results in a change in the corresponding property value and vice versa. Even so, the values may not always be identical. For example, setting the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`src` attribute of the image element to `image.gif` will result in the `src` property being set to the full absolute URL of the image.

For the most part, the name of a JavaScript property matches that of any corresponding attribute, but there are some cases where they differ. For example, the `class` attribute in this example is represented by the `className` property.

jQuery gives us the means to easily manipulate an element's attributes and gives us access to the element instance so that we can also change its properties. Which of these we choose to manipulate depends on what we want to do and how we want to do it.

Let's start by looking at getting and setting element properties.

3.1.1 Manipulating element properties

jQuery doesn't possess a specific method to obtain or modify the properties of elements. Rather, we use the native JavaScript notation to access the properties and their values. The trick is in getting to the element references in the first place.

But it's not really tricky at all, as it turns out. As we saw in the previous chapter, jQuery gives us a number of ways to access the individual elements of the wrapped set. Some of these are:

- Using array indexing on the wrapped set, as in `$(whatever)[n]`
- The `get()` method which returns an individual element by index, or an array of the entire set of elements
- Getting passed an element reference in the callback of the `map()` method
- Getting passed an element reference in the callback of the each() method

As an example of using the `each()` method, we could use the following code to set the `id` property of every element in the DOM to a name composed of the element's tag name and position within the DOM:

```
$('*').each(function(n){
  this.id = this.tagName + n;
});
```

In this example, we obtain element references as the function context (`this`) of the callback function, and directly assign values to their `id` properties.

Dealing with attributes is a little less straightforward than dealing with properties in JavaScript, so jQuery provides more assistance for handling them. Let's look at how.

3.1.2 Fetching attribute values

As we'll find is true with many jQuery methods, the `attr()` method can be used either as a read or as a write operation. When jQuery methods can perform such bilateral operations, the number and types of parameters passed into the method determine which variant of the method is executed.

As one of these bilateral methods, the `attr()` method can be used to either fetch the value of an attribute from the first element in the matched set, or to set attribute values onto all matched elements.

The syntax for the fetch variant of the `attr()` method is as follows:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Method syntax: attr

`attr(name)`

Obtains the values assigned to the specified attribute for the first element in the matched set.

Parameters

`name` (String) The name of the attribute whose value is to be fetched.

Returns

The value of the attribute for the first matched element. The value `undefined` is returned if the matched set is empty or the attribute doesn't exist on the first element.

Even though we usually think of element attributes as those predefined by HTML, we can use `attr()` with custom attributes set through JavaScript or HTML markup. To illustrate this, let's amend the `<img>` element of our previous example with a custom markup attribute (highlighted in bold):

```
<img id="myImage" src="image.gif" alt="An image" class="someClass" title="This is an
image" data-custom="some value"/>
```

Note that we have added a custom attribute, unimaginatively named `data-custom`, to the element. We can retrieve that attribute's value, as if it were any of the standard attributes, with

```
$("#myImage").attr("data-custom")
```

Custom Attributes and HTML

Under HTML 4, using a nonstandard attribute name such as `data-custom`, although a common sleight-of-hand trick, will cause your markup to be considered invalid, and it will fail formal validation testing. Proceed with caution if validation matters to you.

HTML 5, on the other hand, formally recognizes and allows for such custom attributes, as long as the custom attribute name is prefixed with the string `data-`. Any attributes following this naming convention will be considered valid according to HTML 5's rules; those that do not will continue to be considered invalid

In anticipation of HTML 5, we have adopted the `data-` prefix in our example.

Attribute names are not case sensitive in HTML. Regardless of how an attribute such as `title` is declared in the markup, we can access (or set, as we shall see) attributes using any variants of case: `Title`, `TITLE`, `TiTlE`, or any other combinations are all equivalent. In XHTML, even though attribute names must be lowercase in the markup, we can retrieve them using any case variant.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

At this point you may be asking, "Why deal with attributes at all when accessing the properties is so easy (as seen in the previous section)?"

The answer to that question is that the jQuery `attr()` method is much more than a wrapper around the JavaScript `getAttribute()` and `setAttribute()` methods. In addition to allowing access to the set of element attributes, jQuery provides access to some commonly used properties that, traditionally, have been a thorn in the side of page authors everywhere due to their browser dependency.

This set of normalized-access names is shown in table 3.1.

Table 3.1 jQuery `attr()` normalized-access names

Normalized Name	Source name
`class`	`className`
`cssFloat`	`styleFloat` for IE, `cssFloat` for others
`float`	`styleFloat` for IE, `cssFloat` for others
`for`	`htmlFor`
`maxlength`	`maxLength`
`readonly`	`readOnly`
`styleFloat`	`styleFloat` for IE, `cssFloat` for others

In addition to these helpful shortcuts, the set variant of `attr()` has some of its own handy features. Let's take a look.

3.1.3 Setting attribute values

There are two ways to set attributes onto elements in the wrapped set with jQuery. Let's start with the most straightforward that allows us set a single attribute at a time (for all elements in the wrapped set). Its syntax is as follows:

Method syntax: attr

`attr(name,value)`

Sets the named attribute onto all elements in the wrapped set using the passed value.

Parameters

`name` (String) The name of the attribute to be set.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`value`	(Any|Function) Specifies the value of the attribute. This can be any JavaScript expression that results in a value, or it can be a function. See the following discussion for how this parameter is handled.

Returns

The wrapped set.

This variant of `attr()`, which may at first seem simple, is actually rather sophisticated in its operation.

In its most basic form, when the `value` parameter is any JavaScript expression that results in a value (including an array), the computed value of the expression is set as the attribute value.

Things get more interesting when the `value` parameter is a function reference. In such cases, the function is invoked for *each* element in the wrapped set, with the return value of the function used as the attribute value. When the function is invoked, it's passed a single parameter that contains the zero-based index of the element within the wrapped set. Additionally, the element is established as the `this` function context for the function invocation, allowing the function to tune its processing for each specific element—the main power of using functions in this way.

Consider the following statement:

```
$('*').attr('title',function(index) {
  return 'I am element ' + index + ' and my name is ' +
         (this.id ? this.id : 'unset');
});
```

This method will run through all elements on the page, setting the `title` attribute of each element to a string composed using the index of the element within the DOM and the `id` attribute of each specific element.

We'd use this means of specifying the attribute value whenever that value is dependent upon other aspects of the element, or whenever we have other reasons to set the values individually.

The second set variant of `attr()` allows us to conveniently specify multiple attributes at a time.

Method syntax: attr

`attr(attributes)`

Sets the attributes and values specified by the passed object onto all elements of the matched set

Parameters

`attributes`	(Object) An object whose properties are copied as attributes to all elements in the wrapped set

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Returns

The wrapped set

This format is a quick and easy way to set multiple attributes onto all the elements of a wrapped set. The passed parameter can be any object reference, commonly an object literal, whose properties specify the names and values of the attributes to be set. Consider:

```
$('input').attr(
  { value: '', title: 'Please enter a value' }
);
```

This statement sets the `value` of all `<input>` elements to the empty string, and sets the `title` to the string "`Please enter a value`".

Note that if any property value in the object passed as the `value` parameter is a function reference, it operates in a manner similar to that described for the previous format of `attr()`; the function is invoked for each individual element in the matched set.

WARNING

Internet Explorer won't allow the `name` or `type` attributes of `<input>` elements to be changed. If you want to change the name or type of `<input>` elements in Internet Explorer, you must replace the element with a new element possessing the desired name or type.

Now that we know how to get and set attributes, what about getting rid of them?

3.1.4 Removing attributes

In order to remove an attribute from DOM elements, jQuery provides the `removeAttr()` method. Its syntax is as follows:

Method syntax: removeAttr

`removeAttr(name)`

Removes the specified attribute from every matched element

Parameters

`name` (String) The name of the attribute to be removed

Returns

The wrapped set

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Note that removing an attribute doesn't remove any corresponding property from the JavaScript DOM element, though it may cause its value to change. For example, removing a `readonly` attribute from an element would cause the value of the element's `readOnly` property to flip from `true` to `false`, but the property itself isn't removed from the element.

Now let's look at some examples of how we might use this knowledge on our pages.

3.1.5 Fun with attributes

Let's say that we want to make all links on our site that pointed to external domains open in a new window. This is fairly trivial if we're in total control of the entire markup and can add a `target` attribute, as shown:

```
<a href="http://external.com" target="_blank">Some External Site</a>
```

That's all well and good, but what if we're not in control of the markup? We could be running a Content Management System or a wiki, where end users will be able to add content, and we can't rely on them to add the `target="_blank"` to all external links. First, let's try and determine what we want: we want all links whose `href` attribute begins with "`http://`" to open in a new window (which we have determined can be done by setting the `target` attribute to `_blank`).

Well, we can use the techniques we've learned in this section to do this concisely, as follows:

```
$("a[href^='http://']").attr("target","_blank");
```

First, we select all links with an `href` attribute starting with `http://` (which indicates that the reference is external). Then, we set their `target` attribute to `_blank`. Mission accomplished with a single line of jQuery code!

Another excellent use for jQuery's attribute functionality is helping to solve a long-standing issue with web applications (rich and otherwise): the Dreaded Double Submit Problem. This is a common problem in web applications when the latency of form submissions, sometimes several seconds or longer, gives users an opportunity to press the submit button multiple times, causing all manner of grief for the server-side code.

For the client-side of the solution (the server-side code should still be written in a paranoid fashion), we'll hook into the form's `submit` event and disable the submit button after its first press. That way, users won't get the opportunity to click the submit button more than once and will get a visual indication (assuming that disabled buttons appear so in their browser) that the form is in the process of being submitted. Don't worry about the details of event handling in the following example (we'll get more than enough of that coming up in chapter 4), but concentrate on the use of the `attr()` method:

```
$("form").submit(function() {
  $(":submit",this).attr("disabled", "disabled");
});
```

Within the body of the event handler, we grab all submit buttons that are inside our form with the `:submit` selector and modify the `disabled` attribute to the value `"disabled"` (the official W3C-recommended setting for the attribute). Note that when building the matched set, we provide a context value (the second parameter) of `this`. As we'll find out when we dive into event handing in chapter 4, the `this` pointer always refers to the page element for which the event was triggered while operating inside event handlers; in this case, the form instance.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

When is "enabled" not enabling?

Do not be fooled into thinking that you can substitute the value "enabled" for "disabled" as follows:

```
$(whatever).attr("disabled","enabled");
```

and expect the element to become enabled. This code will *still* disable the element!

According to W3C rules it is the *presence* of the `disabled` attribute, not its value, that places the element in disabled state. So it really doesn't matter what the value is, if the `disabled` attribute is present, the element is disabled.

So, to re-enable the element, we would either remove the attribute, or use a convenience that jQuery provides for us: if we provide the Boolean values `true` or `false` as the attribute value (*not* the strings "true" or "false"), jQuery will do the right thing under the covers; removing the attribute for `false`, and adding it for `true`.

WARNING

Disabling the submit button(s) in this way doesn't relieve the server-side code from its responsibility to guard against double submission or any other types of validation. Adding this type of feature to the client code makes things nicer for the end user and helps prevent the double-submit problem under normal circumstances. It doesn't protect against attacks or other hacking attempts, and server-side code must continue to be on its guard.

Attributes and element properties are useful concepts for data as defined by HTML and the W3C, but in the course of page authoring, we frequently need to store our own custom data. Let's see what jQuery can do for us on that front.

3.1.6 Storing custom data on elements

Let's just come right out and say it: global variables suck.

Except for the infrequent, truly global values, it's hard to imagine a worse place to store information that we'll need while defining and implementing the complex behavior of our rich pages. Not only do we run into scope issues, they also don't scale well when we might have multiple operations occurring simultaneously (menus opening and closing, Ajax requests firing, animations executing, and so on).

The functional nature of JavaScript can help mitigate this through the use of closures, but closures can only take us so far and aren't appropriate for every situation.

Since our page behaviors are so element-focused, it makes sense to leverage the elements themselves as storage scopes. Again, the nature of JavaScript, with its ability to dynamically create custom properties on objects, can help us out here. But we must proceed with caution. Being that DOM elements are represented by JavaScript object instances, they, like all other object instances, can be extended with custom properties of our own choosing. But there be dragons there!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

These custom properties, so-called *expandos*, are not without risk. Particularly, it can be easy to create circular references that can lead to serious memory leaks. In traditional web applications, where the DOM is dropped frequently as new pages are loaded, memory leaks may not be as big of an issue. But for us, as authors of highly interactive web applications, employing lots of script on pages that may hang around for quite some time, memory leaks can be a huge problem.

jQuery comes to our rescue by providing a means to tack data onto any DOM element that we choose, in a controlled fashion, and without relying upon potentially problematic expandos.

We can place any arbitrary JavaScript value, even arrays and objects, onto DOM elements by use of the cleverly named `data()` method. The syntax is:

Method syntax: data

`data(name,value)`

Adds the passed value to the jQuery-managed data store for all wrapped elements.

Parameters

`name`	(String) The name of the data to be stored.
`value`	(Object) The value to be stored.

Returns

The wrapped set

Data that's write-only isn't particularly useful, so a means to retrieve the named data must be available. It should be no surprise that the `data()` method is once again used. The syntax for retrieving data using the `data()` method is:

Method syntax: data

`data(name)`

Retrieves any previously stored data with the specified name on the first element of the wrapped set.

Parameters

`name`	(String) The name of the data to be retrieved.

Returns

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

The retrieved data, or `undefined` if not found.

In the interests of proper memory management, jQuery also provides a way to dump any data that may no longer be necessary with the `removeData()` method:

Method syntax: removeData

`removeData(name)`

Removes any previously stored data with the specified name on all elements of the wrapped set.

Parameters

`name`	(String) The name of the data to be removed.

Returns

The wrapped set.

Note that is not necessary to remove data "by hand" when removing element from the DOM using jQuery methods. jQuery will smartly handle that for us.

We mentioned the `className` property much earlier in this section as an example of the case where markup attribute names differ from property names; but, truth be told, class names are a bit special in other respects as well, and are handled as such by jQuery. The next section will describe a better way to deal with class names than by directly accessing the `className` property or using the `attr()` method.

3.2 Changing element styling

If we want to change the styling of an element, we have two options. We can add or remove a class, causing any existing style sheets to restyle the element based on its new classes. Or we can operate on the DOM element itself, applying styles directly.

Let's look at how jQuery makes it simple to make changes to an element's style via classes.

3.2.1 Adding and removing class names

The class name attributes and properties of DOM elements are unique in their format and semantics and are also important to the creation of interactive interfaces. The addition of class names to and removal of class names from an element is one of the primary means by which their stylistic rendering can be modified dynamically.

One of the aspects of element class names that make them unique—and a challenge to deal with—is that each element can be assigned any number of class names. In HTML, the `class` attribute is used to supply these names as a space-delimited string. For example:

```
<div class="someClass anotherClass yetAnotherClass"></div>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Unfortunately, rather than manifesting themselves as an array of names in the DOM element's corresponding `className` property, the class names appear as that same space-delimited string. How disappointing, and how cumbersome! This means that whenever we want to add class names to or remove class names from an element that already has class names, we need to parse the string to determine the individual names when reading it and be sure to restore it to valid space-delimited format when writing it.

Although it's not a monumental task to write code to handle all that, it's always a good idea to abstract such details behind an API that hides the mechanical details of such operations. Luckily, jQuery has already done that for us.

Adding class names to all the elements of a matched set is an easy operation with the following `addClass()` method:

Method syntax: addClass

`addClass(names)`

Adds the specified class name or class names to all elements in the wrapped set

Parameters

`names`	(String) A string containing the class name to add or, if multiple class names are to be added, a space-delimited string of class names

Returns

The wrapped set

Removing class names is just as straightforward with the following `removeClass()` method:

Method syntax: removeClass

`removeClass(names)`

Removes the specified class name or class names from each element in the wrapped set

Parameters

`names`	(String) A string containing the class name to remove or, if multiple class names are to be removed, a space-delimited string of class names

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Returns

The wrapped set

Often, we may want to switch a set of styles back and forth, perhaps to indicate a change between two states or for any other reasons that make sense with our interface. jQuery makes it easy with the `toggleClass()` method.

Method syntax: toggleClass

`toggleClass(name)`

Adds the specified class name if it doesn't exist on an element, or removes the name from elements that already possess the class name. Note that each element is tested individually, so some elements may have the class name added, and others may have it removed.

Parameters

`name` (String) A string containing the class name to toggle.

Returns

The wrapped set.

One situation where the `toggleClass()` method is most useful is when we want to switch visual renditions between elements quickly and easily. Let's consider a "zebra-striping" example in which we want to give alternating rows of a table different colors. And imagine that we have some valid reason to swap the colored background from the odd rows to the even rows (and perhaps back again) when certain events occurred? The `toggleClass()`method would make it almost trivial to add a class name to every other row, while removing it from the remainder.

Let's give it a whirl. In the file `chapter3/zebra.stripes.html`, you'll find a page that presents a table of vehicle information. Within the script defined for that page, a function is defined as follows:

```
function swapThem() {
  $('tr').toggleClass('striped');
}
```

This function uses the `toggleClass()`method to toggle the class named `striped` for all `<tr>` elements. We also defined the following ready handler:

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

The first statement in the body of this handler applies the class `striped` to every other row of the table using the `nth-child` selector that we learned about in the previous chapter. The second statement establishes event handlers for mouse over and mouse out events that both call the same `swapThem`

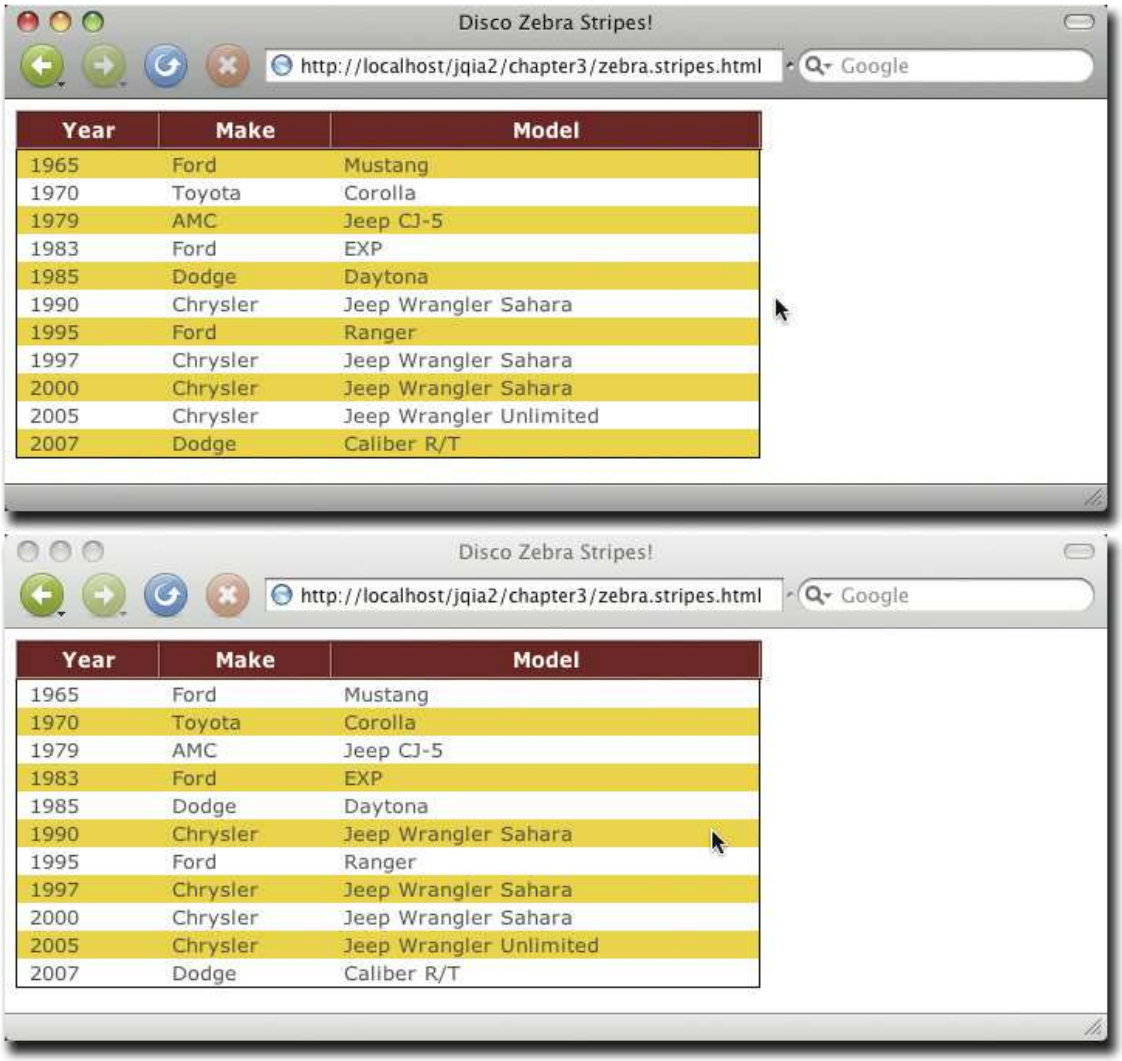
©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

function. We'll be learning all about event handling in the next chapter, but for now the important point is that whenever the mouse enters or leaves the table, the following line of code ends up being executed:

```
$('tr').toggleClass('striped');
```

The result is that every time the mouse cursor enters or leaves the table, all `<tr>` elements with the class `striped` will have the class removed, and all `<tr>` elements without the class will have it added. This (somewhat annoying) activity is shown in the two parts of figure 3.2.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Figure 3.2 The presence or absence of the `striped` class is toggled whenever the mouse cursor enters or leaves the table

Toggling a class based upon whether the elements already possess the class or not is a very common operation, but so is toggling the class based on some other arbitrary condition. For this more general case, jQuery provides another variant of the `toggleClass()` method that lets us add or remove a class based upon an arbitrary Boolean expression:

Method syntax: toggleClass

`toggleClass(name,switch)`

Adds the specified class name if the `switch` expression evaluates to `true`, and removes the class if the switch expression evaluates to `false`.

Parameters

`name`	(String) A string containing the class name to toggle.
`switch`	(Boolean) A control expression whose value determines if the class will be added to the elements (if `true`) or removed (if `false`).

Returns

The wrapped set.

Manipulating the stylistic rendition of elements via CSS class names is a powerful tool, but sometimes we want to get down to the nitty-gritty styles themselves as declared directly on the elements. Let's see what jQuery offers us for that.

3.2.2 Getting and setting styles

Although modifying the class of an element allows us to choose which predetermined set of defined style sheet rules should be applied, sometimes we want to override the style sheet altogether. Applying styles directly on the elements themselves will automatically override style sheets, giving us more fine-grained control over individual elements and their styles.

The `css()` method works similarly to the `attr()` method, allowing us to set an individual CSS property by specifying its name and value, or a series of elements by passing in an object. First, let's look at specifying a single name and value.

Method syntax: css

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`css(name,value)`

Sets the named CSS style property to the specified value for each matched element.

Parameters

`name`	(String) The name of the CSS property to be set.
`value`	(String|Number|Function) A string, number, or function containing the property value. If a function is passed as this parameter, it will be invoked for each element of the wrapped set with its return value serving as the value for the CSS property. The `this` property for each function invocation will be set to the element being evaluated.

Returns

The wrapped set.

As described, the `value` argument accepts a function in a similar fashion to the `attr()` commands. This means that we can, for instance, expand the width of all elements in the wrapped set by 20 pixels as follows:

```
$("div.expandable").css("width",function() {
   return $(this).width() + 20;
});
```

Don't worry that we haven't discussed the `width()` method yet. It does exactly what you would expect (namely, return the width of the element as a number), and we'll discuss it in more detail shortly.

One interesting side note – and yet another example of how jQuery makes our lives easier –is that the normally problematic `opacity` property will work perfectly across browsers by passing in a value between 0.0 and 1.0; no more messing with IE alpha filters, `-moz-opacity`, and the like!

Next, let's look at using the shortcut form of the `css()` method, which works exactly as the shortcut version of `attr()` worked.

Method syntax: css

`css(properties)`

Sets the CSS properties specified as keys in the passed object to their associated values for all matched elements

Parameters

`properties`	(Object) Specifies an object whose properties are copied as CSS properties to all elements in the wrapped set

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Returns

The wrapped set

We've already seen how useful this variant of this method can be in the code of Listing 2.1 that we examined in the previous chapter. To save you some page-flipping, here's the relevant passage again:

```
$('<img>')
  .attr({
    src: 'images/little.bear.png',
    alt: 'Little Bear',
    title:'I woof in your general direction'
  })
  .css({
    cursor: 'pointer',
    border: '1px solid black',
    padding: '12px 12px 20px 12px',
    backgroundColor: 'white'
  })
  …
```

As in the shortcut version of the `attr()` command, we can use functions as values to any CSS property in the `properties` parameter object, and they will be called on each element in the wrapped set to determine the value that should be applied.

Lastly, we can use `css()` with a name passed in to retrieve the computed style of the property associated with that name. When we say *computed* style, we mean the style after all linked, embedded, and inline CSS has been applied. Impressively, this works perfectly across all browsers, even for `opacity`, which returns a string representing a number between 0.0 and 1.0.

Method syntax: css

`css(name)`

Retrieves the computed value of the CSS property specified by `name` for the first element in the wrapped set

Parameters

`name`	(String) Specifies the name of a CSS property whose computed value is to be returned

Returns

The computed value as a String

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Keep in mind that this variant of the `css()`method always returns a string, so if you need a number or some other type, you'll need to parse the returned value.

That's not always convenient, so for a small set of CSS values that are commonly accessed, jQuery thoughtfully provides convenience methods that easily access these values and converts them to the most commonly used types.

GETTING AND SETTING DIMENSIONS

When it comes to CSS styles that we want to set or get on our pages, is there a more common set of properties than the element's width or height? Probably not, so jQuery makes it easy for us to deal with the dimensions of the elements as numeric values rather than strings.

Specifically, we can get (or set) the width and height of an element as a number by using the convenient `width()` and `height()` methods. To set the width or height:

Method syntax: width and height

`width(value)`
`height(value)`

Sets the width or height of all elements in the matched set

Parameters

`value` (Number) The value to be set, in pixels

Returns

The wrapped set

Keep in mind that these are shortcuts for the more verbose `css()` function, so

```
$("div.myElements").width(500)
```

is identical to

```
$("div.myElements").css("width",500)
```

To retrieve the width or height:

Method syntax: width and height

`width()`
`height()`

Retrieves the width or height of the first element of the wrapped set

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Parameters

none

Returns

The computed width or height as a Number.

The fact that the width and height values are returned from these functions as numbers isn't the only convenience that these commands bring to the table. If you've ever tried to find the width or height of an element by looking at its `style.width` or `style.height` property, you were confronted with the sad fact that these properties are only set by the corresponding `style` attribute of that element; to find out the dimensions of an element via these properties, you have to set them in the first place. Not exactly a paragon of usefulness!

The `width()` and `height()` commands, on the other hand, compute and return the size of the element. Although knowing the precise dimensions of an element in simple pages that let their elements lay out wherever they end up isn't usually necessary, knowing such dimensions in highly interactive scripted pages is crucial to be able to correctly place active elements such as context menus, custom tool tips, extended controls, and other dynamic components.

Let's put them to work. Figure 3.3 shows a sample page that was set up with two primary elements: a `<div>` serving as a "test subject" that contains a paragraph of text (also with a border and background color for emphasis) and a second `<div>` in which to display the dimensions.

The dimensions of the test subject aren't known in advance because no style rules specifying dimensions are applied. The width of the element is determined by the width of the browser window, and its height depends on how much room will be needed to display the contained text. Resizing the browser window will cause both dimensions to change.

In our page, we define a function that will use the `width()` and `height()` commands to obtain the dimensions of the test subject `<div>` (named `testSubject`) and display the resulting values in the second `<div>` (named `display`).

```
function displayDimensions() {
  $('#display').html(
    $('#testSubject').width()+'x'+$('#testSubject').height()
  );
```

We call this function in the ready handler of the page, resulting in the initial display of the values 589 and 60 for that particular size of browser window, as shown in the upper portion of figure 3.3.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

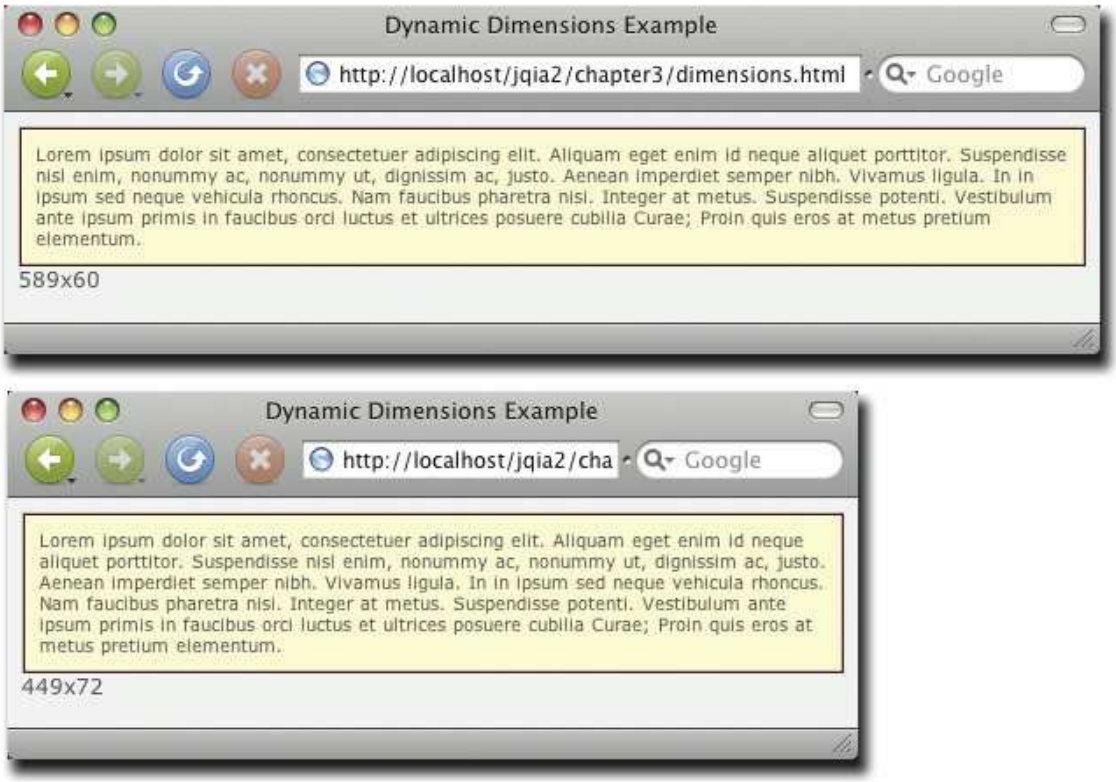


Figure 3.3 The width and height of the test element aren't fixed and depend on the width of the browser window

We also add a call to the same function in a resize handler that we establish on the window that updates the display whenever the browser window is resized, as shown in the lower portion of figure 3.3.

This ability to determine the computed dimensions of an element at any point is crucial to accurately positioning dynamic elements on our pages.

The full code of this page is shown in listing 3.1 and can be found in the file `chapter3/dimensions.html`.

Listing 3.1 Dynamically tracking and displaying the dimensions of an element

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Dynamic Dimensions Example</title>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
    <link rel="stylesheet" type="text/css" href="../styles/examples.css"/>
    <style type="text/css">
      body {
        background-color: #eeeeee;
      }
      #testSubject {
        background-color: #ffffcc;
        border: 2px ridge maroon;
        padding: 8px;
        font-size: .85em;
      }
    </style>
    <script type="text/javascript" src="../scripts/jquery-1.3.2.min.js"></script>
    <script type="text/javascript">
      $(function(){
        $(window).resize(displayDimensions);                              A
        displayDimensions();                                              B
      });

      function displayDimensions() {                                      C
        $('#display').html(
          $('#testSubject').width()+'x'+$('#testSubject').height()
        );
      }
    </script>
  </head>

  <body>
    <div id="testSubject">                                                D
      Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
      Aliquam eget enim id neque aliquet porttitor. Suspendisse
      nisl enim, nonummy ac, nonummy ut, dignissim ac, justo.
      Aenean imperdiet semper nibh. Vivamus ligula. In in ipsum
      sed neque vehicula rhoncus. Nam faucibus pharetra nisi.
      Integer at metus. Suspendisse potenti. Vestibulum ante
      ipsum primis in faucibus orci luctus et ultrices posuere
      cubilia Curae; Proin quis eros at metus pretium elementum.
    </div>
    <div id="display"></div>                                              E
  </body>
</html>
```

#A Establishes resize handler that invokes display function
#B Invokes reporting function in document ready handler
#C Displays width and height of test subject
#D Declares test subject with dummy text
#E Displays dimensions in this area

In addition to the very convenient `width()` and `height()` methods, jQuery also provides similar methods for getting more particular dimension values as described in table 3.2.

Table 3.2 Additional jQuery Dimension-related methods

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Method	Description
`innerHeight()`	Returns the "inner height" of the first matched element, which excludes the border but includes the padding.
`innerWidth()`	Returns the "inner width" of the first matched element, which excludes the border but includes the padding.
`outerHeight(margin)`	Returns the "outer height" of the first matched element, which includes the border and, optionally, the padding. The `margin` parameter causes the padding to be included if it is `true`, or omitted.
`outerWidth(margin)`	Returns the "outer width" of the first matched element, which includes the border and, optionally, the padding. The `margin` parameter causes the padding to be included if it is `true`, or omitted.

When dealing with the window or document elements, it recommended to avoid the inner and outer methods and use `width()` and `height()`.

We're not done yet; jQuery also gives use easy support for positions and scrolling values.

POSITIONS AND SCROLLING

jQuery provides two methods for getting the position of an element. Both of these elements return a JavaScript object that contains two properties: `top` and `left`, which, not surprisingly, indicate the top and left values of the element.

The two methods differ in where they consider the origin from which their relative computed values will measured. One of these methods, `offset()`, returns the position relative to the document:

Method syntax: offset

`offset()`

Returns the position (in pixels) of the first element in the wrapped set relative to the document origin.

Parameters

none

Returns

A JavaScript object with `left` and `top` properties as floats (usually rounded to the nearest integer) depicting the position in pixels relative to the document origin.

The other method, `position()`, returns values relative to an element's closest offset parent:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Method syntax: position

`position()`

Returns the position (in pixels) of the first element in the wrapped set relative to the element's closest offset parent.

Parameters

none

Returns

A JavaScript object with `left` and `top` properties as integers depicting the position in pixels relative to the closest offset parent.

The *offset parent* of an element is the nearest ancestor that has an explicit positioning rule of either `relative` or `absolute` set.

Both `offset()` and `position()` can only be used for visible elements, and it is recommended that pixel values be used for all padding, border and margins to obtain accurate results.

In addition to element positioning, jQuery gives us the ability to get, and to set, the scroll position of an element. Table 3.3 describes these methods:

Table 3.3 The jQuery Scroll Control Methods

Method	Description
`scrollLeft()`	Returns the horizontal scroll offset of the first matched element.
`scrollLeft(value)`	Sets the horizontal scroll offset for all matched elements.
`scrollTop()`	Returns the vertical scroll offset of the first matched element.
`scrollTop(value)`	Sets the vertical scroll offset for all matched elements.

All methods in table 3.3 work with both visible and hidden elements

Now that we've explored manipulating the styles on a wrapped set of elements, let's take a look at a couple of related style-oriented activities that you might want to accomplish, and how to achieve them.

3.2.3 More on class names

It's extremely common to need to determine whether an element has a particular class. With jQuery, we can do that by calling the `hasClass()` method.

```
$("p:first").hasClass("surpriseMe")
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

This method will return `true` if any element in the matched set has the specified class. The syntax of this method is as follows:

Method syntax: hasClass

`hasClass(name)`

Determines if any element of the matched set possesses the passed class name

Parameters

`name` (String) The class name to be checked

Returns

Returns `true` if any element in the wrapped set possesses the passed class name; `false` otherwise.

Recalling the `is()` method from chapter 2, we could achieve the same thing with

```
$("p:first").is(".surpriseMe")
```

In fact, jQuery's inner workings implement the `hasClass()` function exactly that way! But arguably, the `hasClass()`method makes for more readable code.

Another commonly desired ability is to obtain the list of classes defined for a particular element as an array instead of the cumbersome space-separated list. We could try to achieve that by writing

```
$("p:first").attr("class").split(" ");
```

Recall that the `attr()`method will return `undefined` if the attribute in question doesn't exist, so this statement will throw an error if the <p> element doesn't possess any class names. We could solve this by first checking for the attribute, and if we wanted to wrap the entire thing in a repeatable, useful jQuery extension, we could write

```
$.fn.getClassNames = function() {
  if (name = this.attr("className")) {
    return name.split(" ");
  }
  else {
    return [];
  }
};
```

Don't worry about the specifics of the syntax for extending jQuery; we'll go into that in more detail in chapter 7. What's important is that once we define such an extension, we can use `getClassNames()` anywhere in our script to obtain an array of class names or an empty array if an element has no classes. Nifty!

Now that we've learned how to get and set the styles of elements, let's discuss different ways for modifying their contents.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

3.3 Setting element content

When it comes to modifying the contents of elements, there's an ongoing debate regarding which technique is better: using DOM API methods or changing their inner HTML. In most cases, modifying an element's HTML is easier and more effective, so jQuery gives us a number of methods to do so.

3.3.1 Replacing HTML or text content

First up is the simple `html()` method, which allows us to retrieve the HTML contents of an element when used without parameters or, as we've seen with other jQuery functions, to set its contents when used with a parameter.

Here's how to get the HTML content of an element:

Method syntax: html

`html()`

Obtains the HTML content of the first element in the matched set.

Parameters

none

Returns

The HTML content of the first matched element. The returned value is identical to accessing the `innerHTML` property of that element.

And here's how to set the HTML content of all matched elements:

Method syntax: html

`html(text)`

Sets the passed HTML fragment as the content of all matched elements

Parameters

`text` (String) The HTML fragment to be set as the element content

Returns

The wrapped set

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

We can also set or get only the text contents of elements. The `text()`command, when used without parameters, returns a string that's the concatenation of all text. For example, let's say we have the following HTML fragment:

```
<ul id="theList">
  <li>One</li>
  <li>Two</li>
  <li>Three</li>
  <li>Four</li>
</ul>
```

The statement

```
var text = $('#theList').text();
```

results in variable `text` being set to `OneTwoThreeFour`.

Method syntax: text

`text()`

Concatenates all text content of the wrapped elements and returns it as the result of the command

Parameters

none

Returns

The concatenated string

We can also use the `text` method to set the text content of the wrapped elements. The syntax for this format is as follows:

Method syntax: text

`text(content)`

Sets the text content of all wrapped elements to the passed value. If the passed text contains angle brackets (< and >), these characters are replaced with their equivalent HTML entities.

Parameters

`content`	(String) The text content to be set into the wrapped elements. Any angle bracket characters are escaped as HTML entities.

Returns

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

The wrapped set.

Note that setting the inner HTML or text of elements using these commands will replace contents that were previously in the elements, so use these commands carefully. If you don't want to bludgeon all of an element's previous content, a number of other methods will leave the contents of the elements as they are but modify their contents or surrounding elements. Let's look at them.

3.3.2 Moving and copying elements

Manipulating the DOM of a page without the necessity of a page reload opens a world of possibilities for making our pages dynamic and interactive. We've already seen a glimpse of how jQuery let's us create DOM elements on the fly. These new elements can be attached to the DOM in a variety of ways, and we can also move or copy existing elements.

To add content to the end of existing content, the `append()` method is available.

Method syntax: append

`append(content)`

Appends the passed HTML fragment or elements to the content of all matched elements.

Parameters

`content` (String|Element|jQuery) A string, element, or wrapped set to append to the elements of the wrapped set.

Returns

The wrapped set.

This method accepts a string containing an HTML fragment, a reference to an existing or newly created DOM element, or a jQuery wrapped set of elements.

Consider the following simple case:

```
$('p').append('<b>some text<b>');
```

This statement appends the HTML fragment created from the passed string to the end of the existing content of all <p> elements on the page.

A more semantically complex use of this method identifies already-existing elements of the DOM as the items to be appended. Consider the following:

```
$("p.appendToMe").append($("a.appendMe"))
```

This statement moves all links with the class `appendMe` to the end of the child list of all <p> elements with the class `appendToMe`. If there are multiple targets for the operation, the original element is cloned as many times as is necessary and appended to the children of each target. In all cases the original is removed from its initial location.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

This operation is semantically a *move* if one target is identified; the original source element is removed from its initial location and appears at the end of the target's list of children. It can also be a "copy and move" operation if multiple targets are identified, creating enough copies of the original so that each target can have one appended to its children.

In place of a full-blown wrapped set, we can also reference a specific DOM element, as shown:

```
$("p.appendToMe").append(someElement);
```

While it's a common operation to add elements to the end of an elements content – we might be adding a list item to the end of a list, a row to the end of a table, or simply adding a new element to the end of the document body – we might also have a need to add a new or existing element to the *start* of the target element's contents.

When such a need arises, the `prepend()` method will do the trick.

Method syntax: prepend

`prepend(content)`

Prepends the passed HTML fragment or elements to the content of all matched elements.

Parameters

`content` (String|Element|jQuery) A string, element, or wrapped set to append to the elements of the wrapped set.

Returns

The wrapped set.

Sometimes, rather than the beginning or end of an element's content, we might wish to place elements with more precision. jQuery allows us to place new or existing elements anywhere in the DOM by identifying a target element that the source elements are to placed before, or are to be placed after.

Not surprisingly, the methods are named `before()` and `after()`. Their syntax should seem familiar by now.

Method syntax: before

`before(content)`

Inserts the passed HTML fragment or elements into the DOM as a sibling of the target positioned before the targets. The target wrapped elements must already be part of the DOM.

Parameters

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`content` (String|Element|Object) A string, element, or wrapped set to insert into the DOM before the elements of the wrapped set.

Returns

The wrapped set.

Method syntax: after

`after(content)`

Inserts the passed HTML fragment or elements into the DOM as a sibling of the target positioned after the targets. The target wrapped elements must already be part of the DOM.

Parameters

`content` (String|Element|jQuery) A string, element, or wrapped set to insert into the DOM after the elements of the wrapped set.

Returns

The wrapped set.

These operations are key to manipulating the DOM effectively in our pages, so a Move and Copy Lab Page has been provided so that we can play around with these operations until they are thoroughly understood. This lab is available at `chapter3/move.and.copy.lab.html` and its initial display is as shown in figure 3.4.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

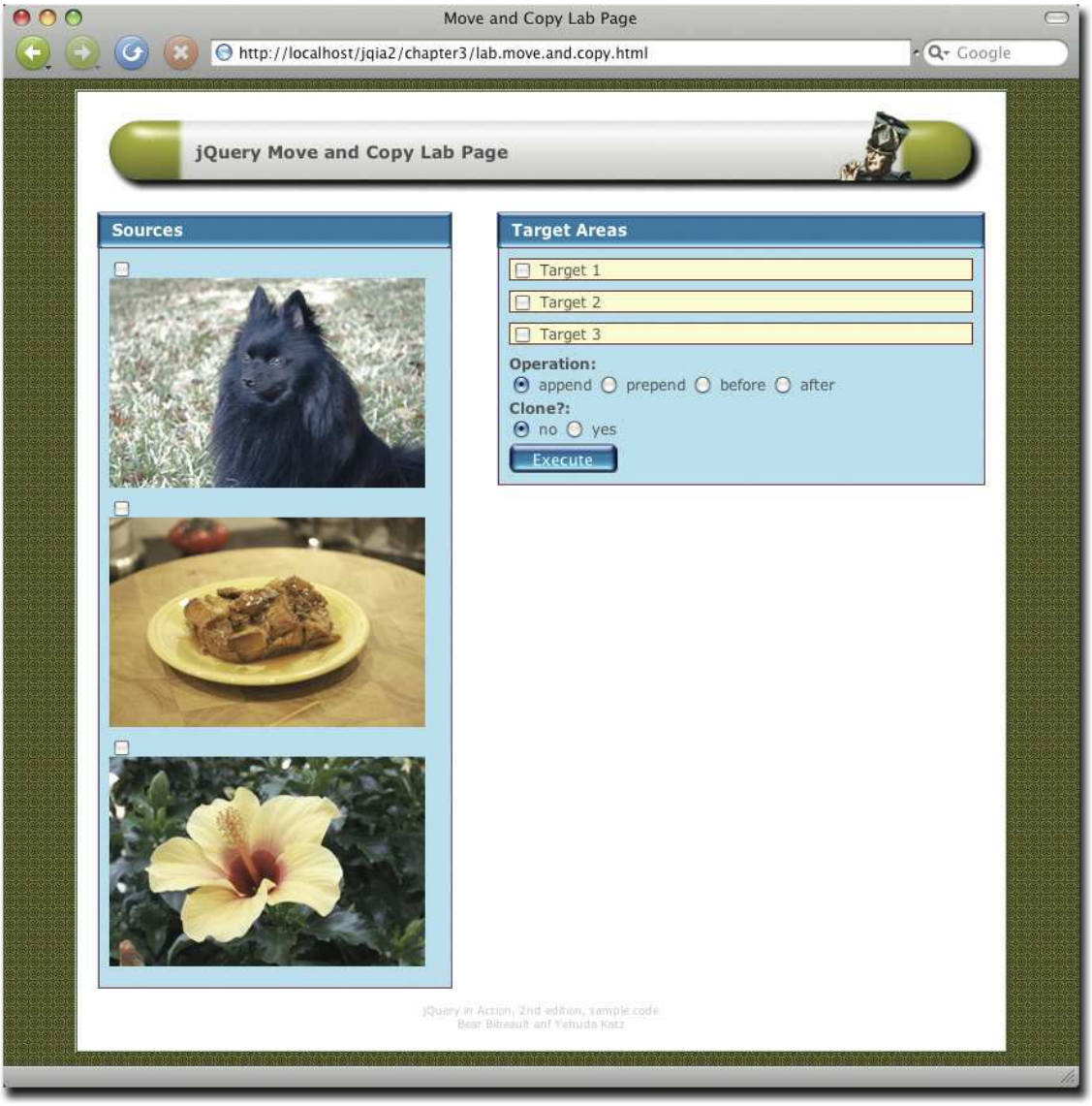


Figure 3.4 The Move and Copy Lab will let us inspect the operation of the DOM manipulation methods

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

The left pane of this Lab Page contains three images that can serve as sources for our move/copy experiments. Select one or more of the images by checking their corresponding check boxes.

Targets for the move/copy operations are in the right pane and are also selected via checkboxes. Controls at the bottom of the pane allow us to select one of the four operations to apply: append, prepend, before or after. (Ignore "clone" for now, we'll attend to that later.)

The Execute button causes any source images you have selected to be applied to a wrapped set of the selected set of targets using the specified operation. After execution, the Execute button is replaced with a Restore button that we will use to put everything back into place so we can run another experiment.

Let's run an "append" experiment.

Select the dog image, and then select Target 2. Leaving the append operation selected, click Execute. The display of figure 3.5 results.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

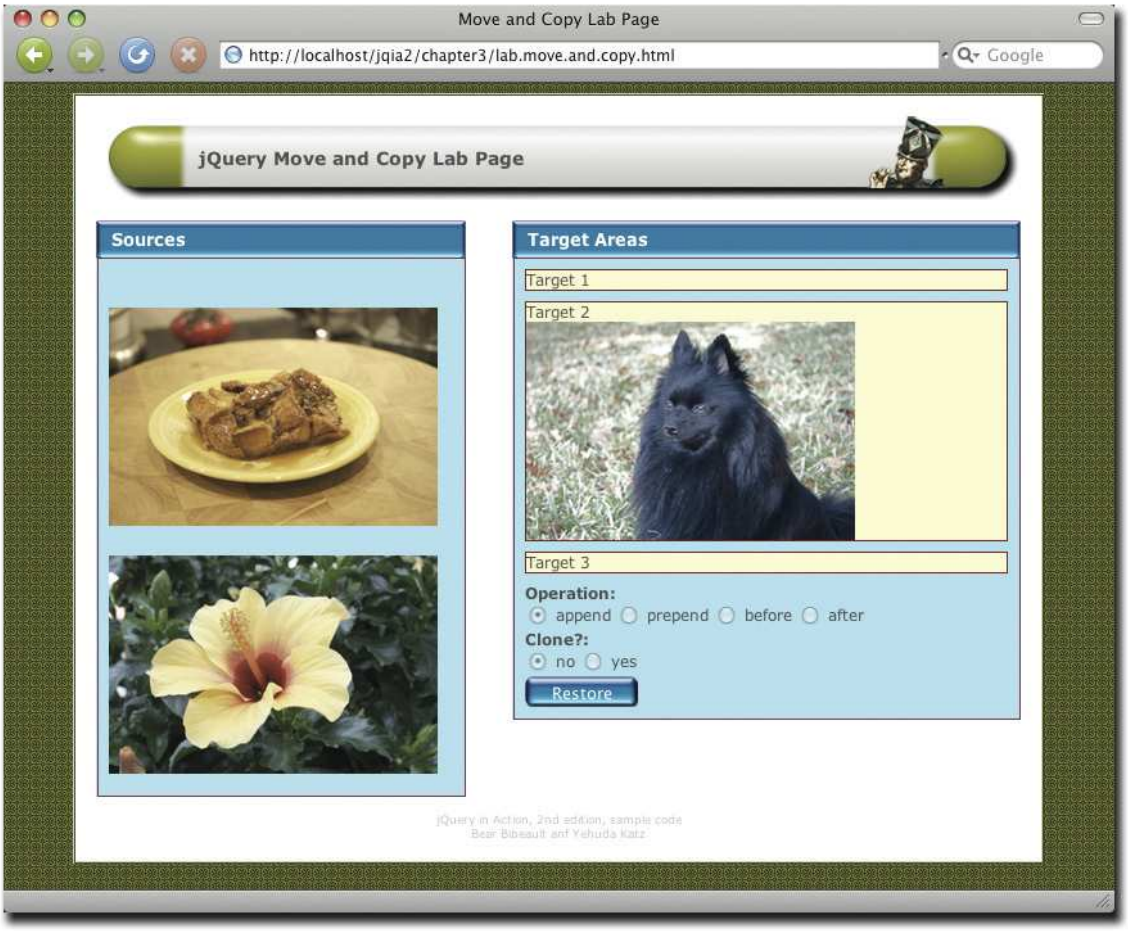


Figure 3.5 Cozmo has been added to the end of Target 2 as a result of the append operation

Use the Move and Copy Lab to try various combinations of sources, targets, and the four operations until you have a good feel for how they operate.

Sometimes, it might make the code more readable if we could reverse the order of the elements passed to these operations. If we want to move or copy an element from one place to another, another approach would be to wrap the source elements (rather than the target elements), and to specify the targets in the parameters of the method. Well, jQuery lets us do that too by providing analogous operations to the four that we just examined that reverse the order in which sources and targets are

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

specified. They are `appendTo()`, `prependTo()`, `insertBefore()`, and `insertAfter()`, and their syntax is as follows:

Method syntax: appendTo

`appendTo(targets)`

Adds all elements in the wrapped set to the end of the content of the specified target(s).

Parameters

`target` (String|Element) A string containing a jQuery selector or a DOM element. Each element of the wrapped set will be appended to the content of the target.

Returns

The wrapped set.

Method syntax: prependTo

`prependTo(targets)`

Adds all elements in the wrapped set to the beginning of the content of the specified target(s).

Parameters

`target` (String|Element) A string containing a jQuery selector or a DOM element. Each element of the wrapped set will be prepended to the content of the target.

Returns

The wrapped set.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Method syntax: insertBefore

`insertBefore(targets)`

Adds all elements in the wrapped set to the DOM just prior to the specified target(s).

Parameters

`target` (String|Element) A string containing a jQuery selector or a DOM element. Each element of the wrapped set will be added before the targets.

Returns

The wrapped set.

Method syntax: insertAfter

`insertAfter (targets)`

Adds all elements in the wrapped set to the DOM just after the specified target(s).

Parameters

`target` (String|Element) A string containing a jQuery selector or a DOM element. Each element of the wrapped set will be added after the targets.

Returns

The wrapped set.

There's one more thing we need to address before we move on…

Remember back in the previous chapter when we showed how to create new HTML fragments with the jQuery `$()` wrapper function? Well, that becomes a really useful trick when paired with the `appendTo()`, `prependTo()`, `insertBefore()`, and `insertAfter()` commands. ReConsider the following:

```
$('<p>Hi there!</p>').insertAfter('p img');
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

This statement creates a friendly paragraph and inserts a copy of it after every image element within a paragraph element. This is an idiom that we've already seen in Listing 2.1, and that we'll use again and again on our pages.

Sometimes, rather than inserting elements *into* other elements, we want to do the opposite. Let's see what jQuery offers for that.

3.3.3 Wrapping elements

Another type of DOM manipulation that we'll often need to perform is to wrap an element (or series of elements) in some markup. For example, we might want to wrap all links of a certain class inside a `<div>`. We can accomplish such DOM modifications by using jQuery's `wrap()` method. Its syntax is as follows:

Method syntax: wrap

`wrap(wrapper)`

Wraps the elements of the matched set with the passed HTML tags or a clone of the passed element.

Parameters

`wrapper` (String|Element) The opening and closing tags of the element with which to wrap each element of the matched set, or an element to be cloned and server as the wrapper.

Returns

The wrapped set.

To wrap each link with the class `surprise` in a `<div>` with the class `hello`, we could write

```
$("a.surprise").wrap("<div class='hello'></div>")
```

If we wanted to wrap the link in a clone of the first `<div>` element on the page:

```
$("a.surprise").wrap($("div:first")[0]);
```

When multiple elements are collected in a matched set, the `wrap()` method operates on each one individually. If we'd rather wrap all the elements in the set as a unit, we can use the `wrapAll()` method instead:

Method syntax: wrapAll

`wrapAll(wrapper)`

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Wraps the elements of the matched set, as a unit, with the passed HTML tags or a clone of the passed element.

Parameters

`wrapper` (String|Element) The opening and closing tags of the element with which to wrap each element of the matched set, or an element to be cloned and server as the wrapper.

Returns

The wrapped set

Sometimes we may not want to wrap the elements that are in a matched set, but rather their *contents*. For just such cases, the `wrapInner()` method is available:

Method syntax: wrapInner

`wrapInner(wrapper)`

Wraps the contents, to include text nodes of the elements in the matched set with the passed HTML tags or a clone of the passed element.

Parameters

`wrapper` (String|Element) The opening and closing tags of the element with which to wrap each element of the matched set, or an element to be cloned and server as the wrapper.

Returns

The wrapped set

Now that we know how to create, wrap, copy, and move elements, we may wonder how we make them go away.

3.3.4 Removing elements

Just as important as the ability to add, move or copy elements in the DOM, is the ability to remove elements that may no longer be needed.

If we want to empty or remove a set of elements, this can be accomplished with the `remove()` method whose syntax is as follows:

Method syntax: remove

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`remove()`

Removes all elements in the wrapped set from the page DOM

Parameters

none

Returns

The wrapped set

Note that, as with many other jQuery methods, the wrapped set is returned as the result of this method. The elements that were removed from the DOM are still referenced by this set (and hence not yet eligible for garbage collection) and can be further operated upon using other jQuery commands including the likes of `appendTo()`, `prependTo()`, `insertBefore()`, `insertAfter()`, and any other similar behaviors we'd like.

To empty DOM elements of their contents, we can use the `empty()` method. Its syntax is as follows:

Method syntax: empty

`empty()`

Removes the content of all DOM elements in the matched set

Parameters

none

Returns

The wrapped set

Sometimes, we don't want to move elements, but to copy them…

3.3.5 Cloning elements

One more way that we can manipulate the DOM is to make copies of elements to attach elsewhere in the tree. jQuery provides a handy wrapper method for doing so with its `clone()` method.

Method syntax: clone

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

clone(copyHandlers)

Creates copies of the elements in the wrapped set and returns a new wrapped set that contains them. The elements and any children are copied. Event handlers are optionally copied depending upon the setting of the `copyHandlers` parameter.

Parameters

`copyHandlers`	(Boolean) If `true`, event handlers are copied. If `false`, or omitted, handlers are not copied.

Returns

The newly created wrapped set.

Making a copy of existing elements with `clone()` isn't useful unless we do something with the carbon copies. Generally, once the wrapped set containing the clones is generated, another jQuery method is applied to stick them somewhere in the DOM. For example:

```
$('img').clone().appendTo('fieldset.photo');
```

This statement makes copies of all image elements and appends them to all `<fieldset>` elements with the class name `photo`.

A slightly more complex example is as follows:

```
$('ul').clone().insertBefore('#here');
```

This method chain performs a similar operation but the targets of the cloning operation—all `<ul>` elements—are copied, *including* their children (it's likely that any `<ul>` element will have a number of `<li>` children).

One last example:

```
$('ul').clone().insertBefore('#here').end().hide();
```

This statement performs the same operation as the previous example, but after the insertion of the clones, the `end()` method is used to select the original wrapped set (the original targets) and hide them. This emphasizes how the cloning operation creates a new set of elements in a new wrapper.

In order to see the clone operation in action, return to the Move and Copy Lab Page. Just above the Execute button is a set of radio buttons that allow us to specify a cloning operation as part of the main DOM manipulation operation. When the `yes` radio button is selected, the sources are cloned before the append, prepend, before or after operations are executed.

Repeat some of the experiments you conducted earlier with cloning enabled and note how the original sources are unaffected by the operations.

We can insert, we can remove, and we can copy. Using these operations in combination it'd be easy to concoct higher-level operations such as *replace*. But guess what? We don't need to!

REPLACING ELEMENTS

For those times when we want to replace existing elements with new ones, or to move an existing element to replace another, jQuery provides the `replaceWith()` method.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Method syntax: replaceWith

`replaceWith(context)`

Replaces each matched element with the specific content.

Parameters

`content`	(String|Element) A string containing an HTML fragment to become the replaced content, or an element reference to be moved to replace the existing elements.

Returns

A jQuery wrapped set containing the replaced elements.

Let's say that, under particular circumstances, we want to replace all images on the page that have `alt` attributes with `<span>` elements that contain the `alt` values of the images. Employing `each()` and `replaceWith()` we could do it with:

```
$('img[alt]').each(function(){
  $(this).replaceWith('<span>'+ $(this).attr('alt') +'</span>')
});
```

The `each()` method lets us iterate over each matching element, and `replaceWith()` is used to replace the images with generated `<span>` elements.

The `replaceWith()` method returns a jQuery wrapped set containing the elements that were removed from the DOM in case we want to do something other than just discard them. How would you augment the example code to reattach these elements elsewhere in the DOM after their removal?

When an existing element is passed to `replaceWith()`, it is detached from its original location in the DOM and reattached to replace the target elements. If there are multiple such targets, the original element is cloned as many times as needed.

At times, it may be convenient to reverse the order of the elements as specified by `replaceWith()` so that the *replacing* element can be specified using the matching selector. We've already seen such complementary methods, such as `append()` and `appendTo()`, that let us specify the elements in the order that makes the most sense for our code.

Similarly, the `replaceAll()` method mirrors `replaceWith()`, allowing us to perform a similar operation, but with the order of specification reversed.

Method syntax: replaceAll

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`replaceAll(selector)`

Replaces each element matched by the passed selector with the content of the matched set to which this method is applied.

Parameters

`selector`	(String) A selector expression identifying the elements to be replaced..

Returns

A jQuery wrapped set containing the replaced elements.

As with `replaceWith()`, `replaceAll()` returns a jQuery wrapped set. But this set contains not the replaced elements, but the *replacing* elements. The replaced elements are lost and cannot be further operated upon. Keep this in mind when deciding which replace method to employ.

Now that we've discussed handling general DOM elements, let's take a brief look at handling a special type of element: the form elements.

3.4 Dealing with form element values

Because form elements have special properties, jQuery's core contains a number of convenience functions for activities such as: getting and setting their values, serializing them, and selecting elements based on form-specific properties. They will serve us well in most cases, but the Form Plugin—an officially sanctioned plugin developed by members of the jQuery Core Team—provides even more form-related functionality. We'll discuss the Form Plugin in chapter 9.

So what's a form element?

When we use the term *form element*, we are referring to the elements that can appear within a form, possess `name` and `value` attributes, and whose values are sent to the server as HTTP request parameters when the form is submitted. Dealing with such elements by hand in script can be tricky because, not only can elements be disabled, but the W3C defines an *unsuccessful* state for controls. This state determines which elements should be ignored during a submission, and it's a tad on the complicated side.

That said, let's take a look at one of the most common operations we'll want to perform on a form element: getting access to its value. jQuery's `val()` method takes care of the most common cases, returning the `value` attribute of a form element for the first element in the wrapped set. Its syntax is as follows:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Method syntax: val

`val()`

Returns the `value` property of the first element in the matched set. When the element is a multi-select element, the returned value is an array of all selections.

Parameters

none

Returns

The fetched value or values.

This method, although quite useful, has a number of limitations of which we need to be wary. If the first element in the wrapped set isn't a form element, an empty string is returned, which isn't the most intuitive value that could have been chosen (`undefined` would probably be clearer). This method also doesn't distinguish between the checked or unchecked states of check boxes and radio buttons, and will simply return the value of check boxes or radio buttons as defined by their `value` attribute, regardless of whether they are checked or not.

For radio buttons, the power of jQuery selectors combined with the `val()` method saves the day as we've already seen in the example with which we opened this book. Consider a form with a radio group (a set of radio buttons with the same name) named `radioGroup` and the following expression:

```
$('[name=radioGroup]:checked').val()
```

This expression returns the value of the single checked radio button (or `undefined` if none is checked). That's a lot easier than looping through the buttons looking for the checked element, isn't it?

Since `val()` only considers the first element in a wrapped set, it's not as useful for check box groups where more than one control might be checked. But jQuery rarely leaves us without recourse. Consider the following:

```
var checkboxValues = $('[name=checkboxGroup]:checked').map(
  function(){ $(this).val(); }
).get();
```

Even though we haven't formally covered extending jQuery (that's still 4 chapters away), you've probably seen enough examples to give it a go. See if you can refactor the above code into a jQuery wrapper method that returns an array of any checked checkbox in the wrapped set.

While the `val()` method is great for obtaining the value of any single form control element, if we want to obtain the complete set of values that would be submitted through a form submission, we'll be much better off using the `serialize()` or `serializeArray()` methods (which we'll see in chapter 8) or the official Form Plugin (chapter 9).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Another common operation we'll perform is to set the value of a form element. The `val()`method is also used bilaterally for this purpose by supplying a value. Its syntax is as follows:

Method syntax: val

`val(value)`

Sets the passed value as the `value` of all matched form elements

Parameters

`value`	(String) A string that is set as the `value` property of each form element in the wrapped set

Returns

The wrapped set

Another way that the `val()` method can be used is to cause check box or radio elements to become checked, or to select options within a `<select>` element. The syntax of this variant of `val()` is as follows:

Method syntax: val

`val(values)`

Causes any check boxes, radio buttons, or options of `<select>` elements in the wrapped set to become checked or selected if their values match any of the values passed in the `values` array.

Parameters

`values`	(Array) An array of values that will be used to determine which elements are to be checked or selected.

Returns

The wrapped set.

Consider the following statement:

```
$('input,select').val(['one','two','three']);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

This statement will search all the `<input>` and `<select>` elements on the page for values that match any of the input strings: *one*, *two* or *three*. Any check boxes or radio buttons that are found to match will become checked, and any options that match will become selected.

This makes `val()` useful for much more than just the text-based form elements.

3.5 Summary

In this chapter, we've gone beyond the art of selecting elements and started manipulating them. With the techniques we've learned so far, we can select elements using powerful criteria, and then move them surgically to any part of the page.

We can choose to copy elements, or to move them, replace them, or even create brand new elements from scratch. We can append, prepend, or wrap any element or set of elements on the page. And we've learned how to manage the values of form elements, all leading to powerful yet succinct logic.

With that behind us, we're ready to start looking into more advanced concepts, starting with the typically messy job of handling events in our pages.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

4

Events are where it happens!

This chapter covers

- The event models as implemented by the browsers
- The jQuery event model
- Binding event handlers to DOM elements
- The `Event` object instance
- Triggering event handlers under script control
- Registering proactive event handlers

Anyone familiar with the Broadway show *Cabaret*, or its subsequent Hollywood film, probably remembers the song "Money Makes the World Go Around." Although this cynical view might be applicable to the physical world, in the virtual realm of the World Wide Web, it's events that make it all happen!

Like many other GUI management systems, the interfaces presented by HTML web pages are *asynchronous* and *event-driven* (even if the HTTP protocol used to deliver them to the browser is wholly synchronous in nature). Whether a GUI is implemented as a desktop program using Java Swing, X11, the .NET framework, or a page in a web application using HTML and JavaScript, the process is pretty much the same:

1. Set up the user interface.
2. Wait for something interesting to happen.
3. React accordingly.
4. Repeat.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

The first step sets up the *display* of the user interface; the others define its *behavior*. In web pages, the browser handles the setup of the display in response to the markup (HTML and CSS) that we send to it. The script we include in the page defines the behavior of the interface.

This script takes the form of *event handlers*, also known as *listeners*, that react to the various events that occur while the page is displayed. These events could be generated by the system (such as timers or the completion of asynchronous requests) but are most often the result of some user activity (such as moving or clicking the mouse, entering text via the keyboard, or even through iPhone gestures). Without the ability to react to these events, the World Wide Web's greatest use might be limited to showing pictures of kittens.

Although HTML itself *does* define a small set of built-in semantic actions that require no script on our part (such as reloading pages as the result of clicking an anchor tag or submitting a form via a submit button), any other behaviors that we wish our pages to exhibit require us to handle the various events that occur as our users interact with those pages.

In this chapter, we examine the various ways that the browsers expose these events, how they allow us to establish handlers to control what happens when these events occur, and the challenges that we face due to the multitude of differences between the browser event models. Then we'll see how jQuery cuts through the browser-induced fog to relieve us of these burdens.

JavaScript you need to know!

One of the great benefits that jQuery brings to web applications is the ability to implement a great deal of scripting-enabled behavior without having to write a whole lot of script ourselves. jQuery handles the nuts-and-bolts details so that we can concentrate on the job of making our applications do what it is that our applications need to do!

Up to this point, the ride has been pretty painless. You only needed rudimentary JavaScript skills to code and understand the jQuery examples we introduced in the previous chapters. In this chapter and the chapters that follow, you *must* understand a handful of important fundamental JavaScript concepts to make effective use of the jQuery library.

Depending on your background, you may already be familiar with these concepts, but some page authors may have been able to get pretty far without a firm grasp of these concepts—the very flexibility of JavaScript makes such a situation possible. Before we proceed, it's time to make sure that you've wrapped your head around these core concepts.

If you're already comfortable with the workings of the JavaScript `Object` and `Function` classes, and have a good handle on concepts like *function contexts* and *closures*, you may want to continue reading this and the upcoming chapters. If these concepts are unfamiliar or hazy, we strongly urge you to turn to the appendix to help you get up to speed on these necessary concepts.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Let's start off by examining how browsers expose their events models.

4.1 Understanding the browser event models

Long before anyone considered standardizing how browsers would handle events, Netscape Communications Corporation introduced an event-handling model in its Netscape Navigator browser; all modern browsers still support this model, which is still probably the best understood and most employed by the majority of page authors.

This model is known by a few names. You may have heard it termed the Netscape Event Model, the Basic Event Model, or even the rather vague Browser Event Model; but most people have come to call it the *DOM Level 0 Event Model*.

> **NOTE**
>
> The term *DOM Level* is used to indicate what level of requirements an implementation of the W3C DOM Specification meets. There isn't a DOM Level 0, but that term is used to informally describe what was implemented *prior* to the DOM Level 1.

The W3C didn't create a standardized model for event handling until DOM Level 2, introduced in November 2000. This model enjoys support from all modern standards-compliant browsers such as Firefox, Camino (as well as other Mozilla browsers), Safari, and Opera. Internet Explorer continues to go its own way and supports a subset of the DOM Level 2 Event Model functionality, albeit using a proprietary interface.

Before we see how jQuery makes that irritating fact a non-issue, let's spend time getting to know how the event models operate.

4.1.1 The DOM Level 0 Event Model

The DOM Level 0 Event Model is probably the event model that most web developers employ on their pages. In addition to being somewhat browser-independent, it's fairly easy to use.

Under this event model, event handlers are declared by assigning a reference to a function instance to properties of the DOM elements. These properties are defined to handle a specific event type; for example, a click event is handled by assigning a function to the `onclick` property, and a mouseover event by assigning a function to the `onmouseover` property of elements that support these event types.

The browsers allow us to specify the body of an event handler function as attribute values in the DOM elements' HTML, providing a shorthand for creating event handlers. An example of defining such handlers is shown in listing 4.1. This page can be found in the downloadable code for this book in the file `chapter4/dom.0.events.html`.

Listing 4.1 Declaring DOM Level 0 event handlers

```
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Level 0 Events Example</title>
    <script type="text/javascript" src="../scripts/jquery-1.3.2.min.js"></script>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
    <script type="text/javascript">
      $(function(){
        $('#vstar')[0].onmouseover = function(event) {                          #1
          say('Whee!');
        };
      });

      function say(text) {                                                      #2
        $('#console').append('<div>'+new Date()+' '+text+'</div>');
      }
    </script>
  </head>

  <body>
    <img id="vstar" src="vstar.jpg"
         onclick="say('Vroom vroom!')"                                        #3
         alt="Bear's ride"/>
    <div id="console"></div>                                                  #4
  </body>
</html>
```

#1 Defines mouseover handler
#2 Emits text to "console"
#3 Instruments <img> element
#4 Serves as "console"

Cueballs in code and text

In this example, we employ both styles of event handler declaration: declaring under script control and declaring in a markup attribute.

The page first declares a ready handler (#1) in which a reference to the image element with the `id` of `vstar` is obtained (using jQuery), and its `onmouseover` property is set to a function instance that we declare inline. This function becomes the event handler for the element when a mouseover event is triggered on it. Note that this function expects a single parameter to be passed to it. We'll learn more about this parameter shortly.

We also declare a small utility function, `say()` (#2), that we use to emit text messages to a `<div>` element on the page that we'll call the "console"(#4). This will save us the trouble of using annoying and disruptive alerts to indicate when things happen on our page.

In the body of the page (along with the console element), we define an image element (#3) on which we've defined the event handlers. We've already seen how to define one under script control in the ready handler (#1), but here we declare a handler for a click event using the `onclick` attribute of the `<img>` element.

NOTE

Obviously we've thrown out the concept of Unobtrusive JavaScript for this example. Long before we reach the end of this chapter, we'll see why we won't need to do this anymore!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Loading this page into a browser (found in the file `chapter4/dom.0.events.html`), waving the mouse pointer over the image a few times, and then clicking the image, results in a display similar to that shown in figure 4.1.

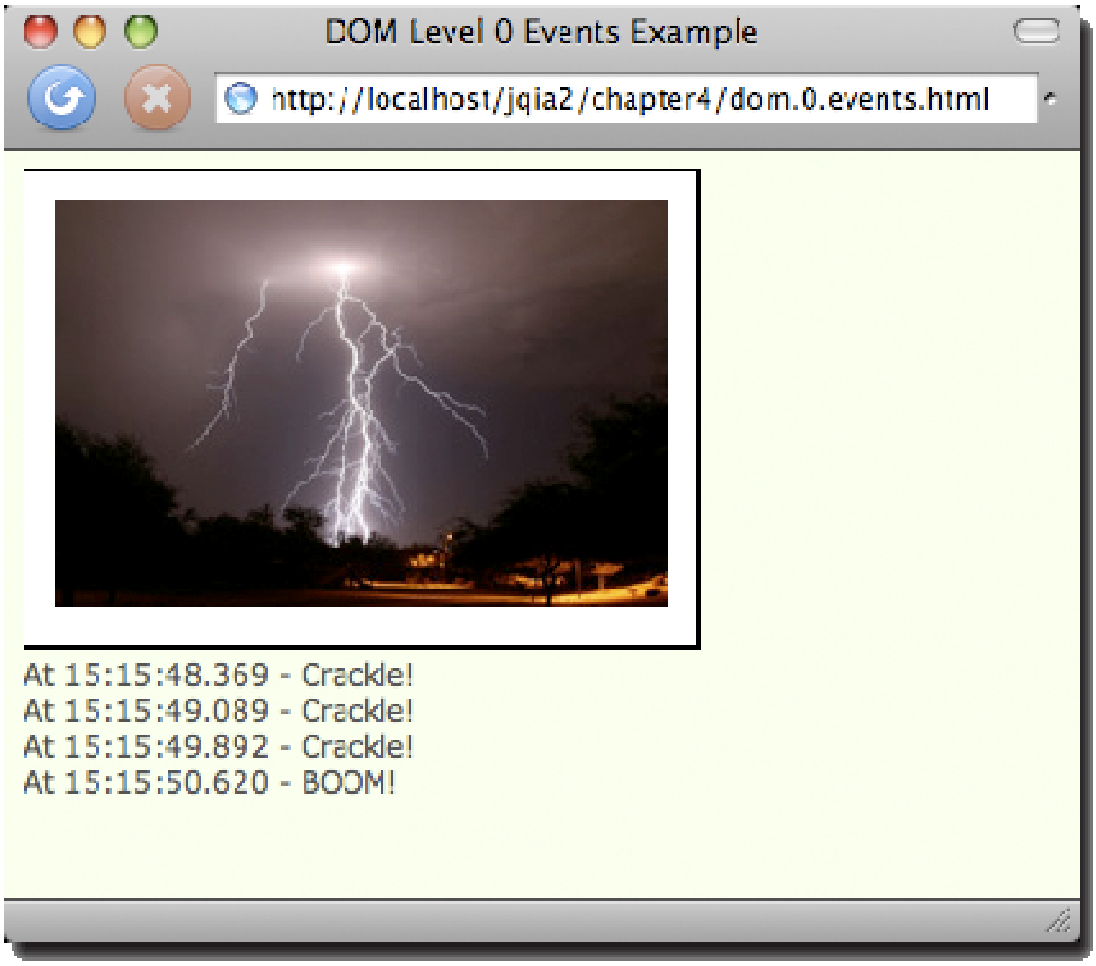


Figure 4.1 Waving the mouse over the image and clicking it result in the event handlers firing and emitting their messages to the console

We declared the click event handler in the `<img>` element markup using the following attribute:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
onclick="say('Vroom vroom!');"
```

This might lead us to believe that the `say()` function becomes the click event handler for the element, but that's not really the case. When handlers are declared via HTML markup attributes, an anonymous function is automatically created using the value of the attribute as the function *body*. Assuming that `imageElement` is a reference to the image element, the construct created as a result of the attribute declaration is equivalent to the following:

```
imageElement.onclick = function(event) {
 say('Vroom vroom!');
}
```

Note how the value of the attribute is used as the *body* of the generated function, and note that the function is created so that the `event` parameter is available within the generated function.

Before we move on to examining what that `event` parameter is all about, we should note that using the attribute mechanism of declaring DOM Level 0 event handlers violates the precepts of Unobtrusive JavaScript that we explored in section 1.2. When using jQuery in our pages, we should adhere to the principles of Unobtrusive JavaScript and avoid mixing behavior defined by JavaScript with display markup. We'll shortly see that jQuery provides a much better way to declare event handlers than either of these means.

But first, let's examine what that `event` parameter is all about.

THE EVENT INSTANCE

When an event handler is fired, an instance of a class named `Event` is passed to the handler as its first parameter in most browsers. Internet Explorer, always the life of the party, does things in its own proprietary way by tacking the `Event` instance onto a global property (in other words, a property on `window`) named `event`.

In order to deal with this discrepancy we'll often see the following used as the first statement in an event handler:

```
if (!event) event = window.event;
```

This levels the playing field by using feature detection (a concept we'll explore in greater depth in chapter 6) to check if the `event` parameter is undefined (or null) and assigning the value of the window's `event` property to it if so. After this statement, the `event` parameter can be referenced regardless of how it was made available to the handler.

The properties of the `Event` instance provide a great deal of information regarding the event that has been fired and is currently being handled. This includes details such as which element the event was triggered on, the coordinates of mouse events, and which key was clicked for keyboard events.

But not so fast. Not only does Internet Explorer use a proprietary means to get the `Event` instance to the handler, but it also uses a proprietary definition of the `Event` class in place of the W3C-defined standard—we're not out of the object-detection woods yet.

For example, to get a reference to the target element—the element on which the event was triggered—we access the `target` property in standards-compliant browsers and the `srcElement` property in Internet Explorer. We deal with this inconsistency by employing object detection with a statement such as the following:

```
var target = (event.target) ? event.target : event.srcElement;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

This statement tests if `event.target` is defined and, if so, assigns its value to the local `target` variable; otherwise, it assigns `event.srcElement`. We will be required to take similar steps for other `Event` properties of interest.

Up until this point, we've acted as if event handlers are only pertinent to the elements that serve as the trigger to an event—the image element of listing 4.1, for example--but events propagate throughout the DOM tree. Let's find out about that.

EVENT BUBBLING

When an event is triggered on an element in the DOM tree, the event-handling mechanism of the browser checks to see if a handler has been established for that particular event on that element and, if so, invokes it. But that's hardly the end of the story.

After the target element has had its chance to handle the event, the event model checks with the parent of that element to see if *it* has established a handler for the event type, and if so, it's also invoked—after which *its* parent is checked, then its parent, then its parent, and on and on, all the way up to the top of the DOM tree. Because the event handling propagates upward like the bubbles in a champagne flute (assuming we view the DOM tree with its root at the top), this process is termed *event bubbling*.

Let's modify the example of listing 4.1 so that we can see this process in action. Consider the code in listing 4.2.

Listing 4.2 Events propogate from the point of origin to the top of the DOM

```
<!DOCTYPE html>
<html id="greatgreatgrandpa">
  <head>
    <title>DOM Level 0 Bubbling Example</title>
    <script type="text/javascript" src="../scripts/jquery-1.3.2.min.js">
    </script>
    <script type="text/javascript">
      $(function(){
        $('*').each(function(){                                          #1
          var current = this;
          this.onclick = function(event) {                               #2
            if (!event) event = window.event;
            var target = (event.target) ?
                         event.target : event.srcElement;
            say('For ' + current.tagName + '#'+ current.id +
                ' target is ' + target.id);
          };
        });
      });

      function say(text) {
        $('#console').append('<div>'+text+'</div>');
      }
    </script>
  </head>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
  <body id="greatgrandpa">
    <div id="grandpa">
      <div id="pops">
        <img id="vstar" src="vstar.jpg" alt="Bear's ride"/>
      </div>
    </div>
    <div id="console"></div>
  </body>
</html>
```

#1 Selects every element on the page
#2 Applies onclick handler to every selected element

Cueballs in code and text

We do a lot of interesting things in the changes to this example. First, we removed the previous handling of the mouseover event so that we can concentrate on the click event. We also embed the image element that will serve as the target for our event experiment in a couple of nested `<div>` elements merely to place the image element artificially deeper within the DOM hierarchy. We also give almost every element in the page a specific and unique `id`—even the `<body>` and `<html>` tags!

We retain the console and its `say()` utility function for the same reporting purposes used in the previous example.

Now let's look at even more interesting changes.

In the ready handler for the page, we use jQuery to select all elements on the page and to iterate over each one with the `each()` method (#1). For each matched element, we record its instance in the local variable `current` and establish an `onclick` handler (#2). This handler first employs the browser-dependent tricks that we discussed in the previous section to locate the `Event` instance and identify the event target, and then emits a console message. This message is the most interesting part of this example.

It displays the tag name and `id` of the current element, putting *closures* to work (please read section A.2.4 in the appendix if closures are a subject that gives you heartburn), followed by the `id` of the target. By doing so, each message that's logged to the console displays the information about the current element of the bubble process, as well as the target element that started the whole shebang.

Loading the page (located in the file `chapter4/dom.0.propagation.html`) and clicking the image results in the display of figure 4.2.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

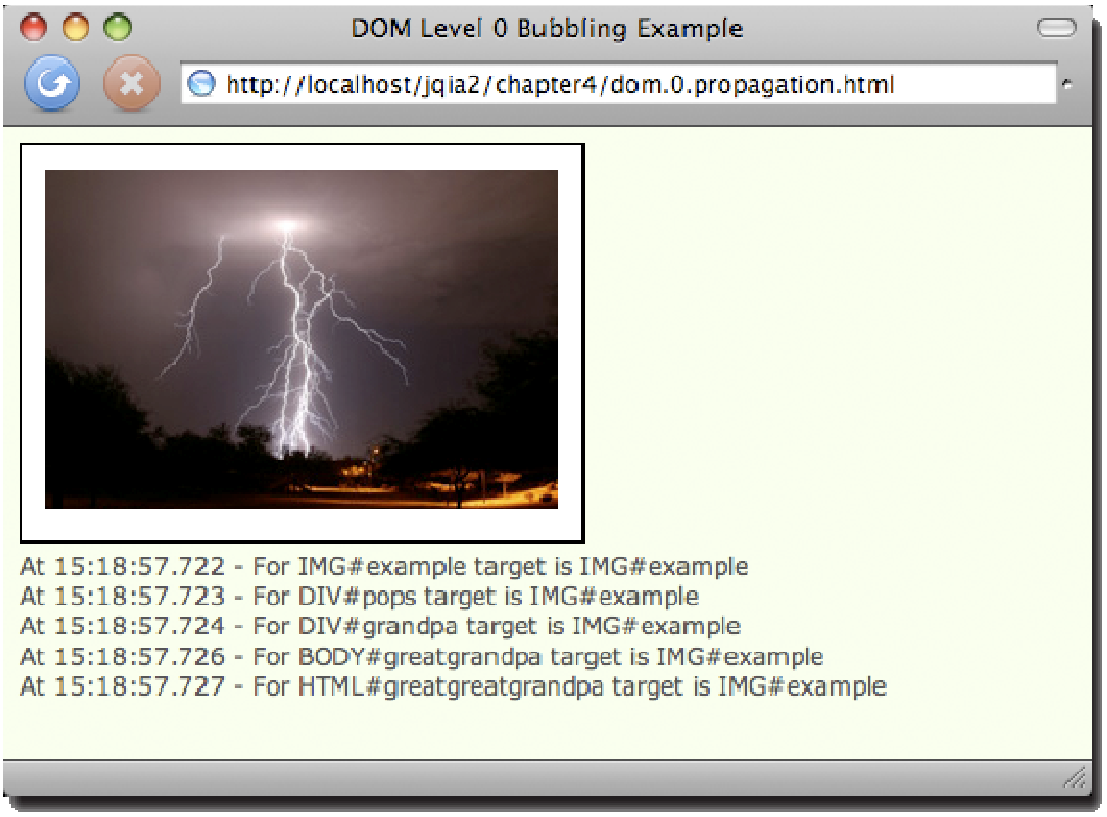


Figure 4.2 The console messages clearly show the propagation of the event as it bubbles up the DOM tree from the target element to the tree root.

This clearly illustrates that, when the event is fired, it's delivered first to the target element and then to each of its ancestors in turn, all the way up to the root `<html>` element.

This is a powerful ability because it allows us to establish handlers on elements at any level to handle events occurring on its descendents. Consider a handler on a `<form>` element that reacts to any change event on its child elements to effect dynamic changes to the display based upon the elements' new values.

But what if we don't *want* the event to propagate? Can we stop it?

AFFECTING EVENT PROPAGATION AND SEMANTIC ACTIONS

There may be occasions where we want to prevent an event from bubbling any further up the DOM tree. This might be because we're fastidious and we know that we've already accomplished any processing

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

necessary to handle the event, or we may want to forestall unwanted handling that might occur higher up in the chain.

Regardless of the reason, we can prevent an event from propagating any higher via mechanisms provided on the `Event` instance. For standards-compliant browsers, we call the `stopPropagation()` method of the `Event` instance to halt the propagation of the event further up the ancestor hierarchy. In Internet Explorer, we set a property named `cancelBubble` to `true` in the `Event` instance. Interestingly, many modern standards-compliant browsers support the `cancelBubble` mechanism even though it's not part of any W3C standard.

Some events have default semantics associated with them. As examples, a click event on an anchor element will cause the browser to navigate to the element's `href`, and a submit event on a `<form>` element will cause the form to be submitted. Should we wish to cancel these semantic actions—sometimes termed the *default actions*—of the event, we simply return the value `false` from the event handler.

A frequent use for such an action is in the realm of form validation. In the handler for the form's submit event, we can make validation checks on the form's controls, and return `false` if any problems with the data entry are detected.

We may also have seen the following on `<form>` elements:

```
<form name="myForm" onsubmit="return false;" ...
```

This effectively prevents the form from being submitted under any circumstances except under script control (via `form.submit()`, which doesn't trigger a submit event)—a common trick used in many Ajax applications where asynchronous requests will be made in lieu of form submissions.

Under the DOM Level 0 Event Model, almost every step we take in an event handler involves using browser-specific detection in order to figure out what action to take. What a headache! But don't put away the aspirin yet—it doesn't get any easier when we consider the more advanced event model.

4.1.2 The DOM Level 2 Event Model

One severe shortcoming of the DOM Level 0 Event Model is that, because a property is used to store a reference to a function that's to serve as an event handler, only one event handler per element can be registered for any specific event type at a time. If we have two things that we want to do when an element is clicked, the following statements aren't going to let that happen:

```
someElement.onclick = doFirstThing;
someElement.onclick = doSecondThing;
```

Because the second assignment replaces the previous value of the `onclick` property, only `doSecondThing` is invoked when the event is triggered. Sure, we could wrap both functions in another single function that calls both; but as pages get more complicated, as is highly likely in DOM-scripted Applications, it becomes increasingly difficult to keep track of such things. Moreover, if we use multiple reusable components or libraries in a page, they may have no idea of the event-handling needs of the other components.

We could employ other solutions: implementing the Observable pattern that establishes a publish/subscribe scheme for the handlers, or even tricks using closures. But all of these add complexity to pages that are likely to already be complex enough.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Besides the establishment of a *standard* event model, the DOM Level 2 Event Model was designed to address these types of problems. Let's see how event handlers, even multiple handlers, are established on DOM elements under this more advanced model.

ESTABLISHING EVENT HANDLERS

Rather than assigning a function reference to an element property, DOM Level 2 event handlers—also termed *listeners*—are established via an element *method*. Each DOM element defines a method named `addEventListener()` that's used to attach event handlers (listeners) to the element. The format of this method is as follows:

```
addEventListener(eventType,listener,useCapture)
```

The `eventType` parameter is a string that identifies the type of event to be handled. This string is, generally, the same event names we used in the DOM Level 0 Event Model without the *on* prefix: for example: "click", "mouseover", "keydown", and so on.

The `listener` parameter is a reference to the function (or an inline function) that's to be established as the handler for the named event type on the element. As in the basic event model, the `Event` instance is passed to this function as its first parameter.

The final parameter, `useCapture`, is a Boolean whose operation we'll explore in a few moments when we discuss event propagation in the Level 2 Model. For now, we'll leave it set to `false`.

Let's once again change the example of listing 4.1 to use the more advanced event model. We'll concentrate only on the click event type; this time, we'll establish *three* click event handlers on the image element. The new example code can be found in the file `chapter4/dom.2.events.html` and is shown in listing 4.3.

Listing 4.3 Establishing event handlers with the DOM Level 2 Model

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>DOM Level 2 Events Example</title>
    <link rel="stylesheet" type="text/css" href="../styles/examples.css"/>
    <script type="text/javascript" src="../scripts/jquery-1.3.2.min.js"></script>
    <script type="text/javascript">
      $(function(){                                                    #1
        var element = $('#vstar')[0];
        element.addEventListener('click',function(event) {
          say('Whee once!');
        },false);
        element.addEventListener('click',function(event) {
          say('Whee twice!');
        },false);
        element.addEventListener('click',function(event) {
          say('Whee three times!');
        },false);
      });

      function say(text) {
        $('#console').append('<div>'+text+'</div>');
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
      }
    </script>
  </head>

  <body>
    <img id="vstar" src="vstar.jpg"/>
    <div id="console"></div>
  </body>
</html>
```

#1 Establishes three event handlers!

Cueballs in code and text

This code is simple but clearly shows how we have the ability to establish multiple event handlers on the same element for the same event type—something we were not able to do easily with the Basic Event Model. In the ready handler #1 for the page, we grab a reference to the image element and then establish *three* event handlers for the click event.

Loading this page into a standards-compliant browser (*not* Internet Explorer) and clicking the image results in the display shown in figure 4.3.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

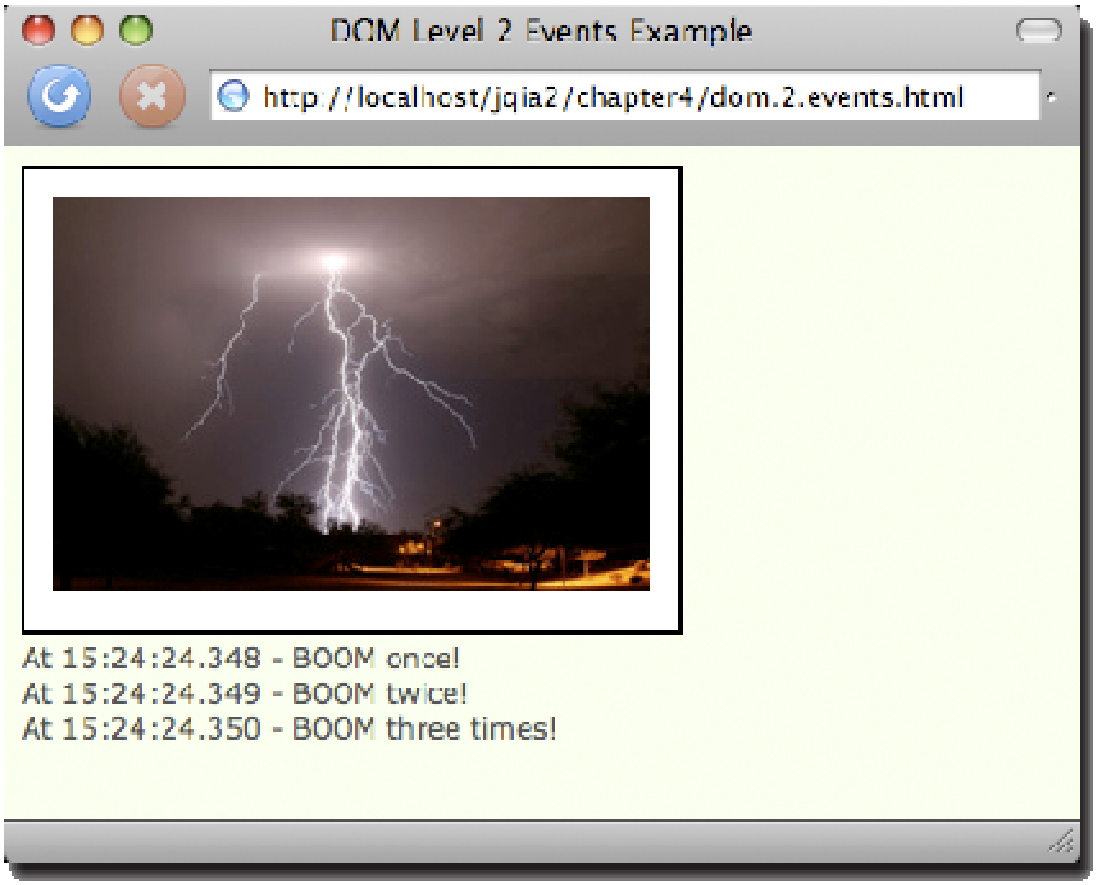


Figure 4.3 Clicking the image once demonstrates that all three handlers established for the click event are triggered.

Note that even though the handlers fire in the order in which they were established, *this order isn't guaranteed by the standard*! Testers of this code never observed an order other than the order of establishment, but it would be foolish to write code that relies on this order. Always be aware that multiple handlers established on an element may fire in random order.

Now, let's find out what's up with that `useCapture` parameter?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

EVENT PROPAGATION

We saw earlier that, with the Basic Event Model, once an event was triggered on an element the event propagated from the target element upwards in the DOM tree to all the target's ancestors. The advanced Level 2 Model also provides this bubbling phase but ups the ante with an additional phase: *capture phase*.

Under the DOM Level 2 Event Model, when an event is triggered, the event first propagates from the root of the DOM tree down to the target element and then propagates again from the target element up to the DOM root. The former phase (root to target) is called *capture phase*, and the latter (target to root) is called *bubble phase*.

When a function is established as an event handler, it can be flagged as a capture handler in which case it will be triggered during capture phase, or as a bubble handler to be triggered during bubble phase. As you might have guessed by this time, the `useCapture` parameter to `addEventListener()` identifies which type of handler is established. A value of `false` for this parameter establishes a bubble handler, whereas a value of `true` registers a capture handler.

Think back a moment to the example of listing 4.2 where we explored the propagation of the Basic Model events through a DOM hierarchy. In that example we embedded an image element within two layers of `<div>` elements. Within such a hierarchy, the propagation of a click event with the `<img>` element as its target would move through the DOM tree as shown in figure 4.4.

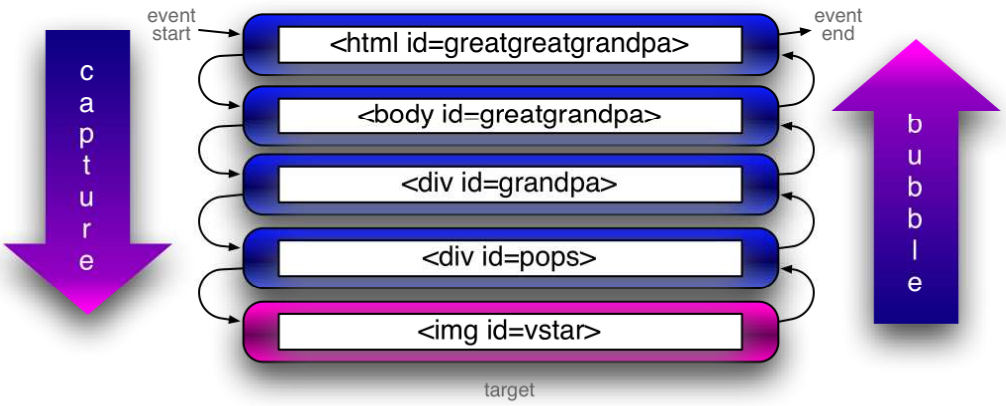


Figure 4.4 Propagation in the DOM Level 2 Event Model traverses the DOM hierarchy twice: once from top to target during capture phase and once from target to top during bubble phase.

Let's put that to the test, shall we? Listing 4.4 shows the code for a page containing the element hierarchy of figure 4.4 (`chapter4/dom.2.propagation.html`).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Listing 4.4 Tracking event propagation with bubble and capture handlers

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
id="greatgreatgrandpa">
  <head>
    <title>DOM Level 2 Propagation Example</title>
    <link rel="stylesheet" type="text/css" href="../styles/examples.css"/>
    <script type="text/javascript" src="../scripts/jquery-1.3.2.min.js"></script>
    <script type="text/javascript">
      $(function(){                                                    #1
        $('*').each(function(){
          var current = this;
          this.addEventListener('click',function(event) {
            say('Capture for ' + current.tagName + '#'+ current.id +
                ' target is ' + event.target.id);
            },true);
          this.addEventListener('click',function(event) {
            say('Bubble for ' + current.tagName + '#'+ current.id +
                ' target is ' + event.target.id);
            },false);
        });
      });

      function say(text) {
        $('#console').append('<div>'+text+'</div>');
      }
    </script>
  </head>

  <body id="greatgrandpa">
    <div id="grandpa">
      <div id="pops">
        <img id="vstar" src="vstar.jpg"/>
      </div>
    </div>
    <div id="console"></div>
  </body>
</html>
```

#1 Establishes listeners on all elements

Cueballs in code and text

This code changes the example of listing 4.2 to use the DOM Level 2 Event Model API to establish the event handlers. In the ready handler #1, we use jQuery's powerful abilities to run through every element of the DOM tree. On each, we establish two handlers: one capture handler and one bubble handler. Each handler emits a message to the console identifying which type of handler it is, the current element, and the `id` of the target element.

With the page loaded into a standards-compliant browser, clicking the image results in the display in figure 4.5, showing the progression of the event through the handling phases and the DOM tree.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

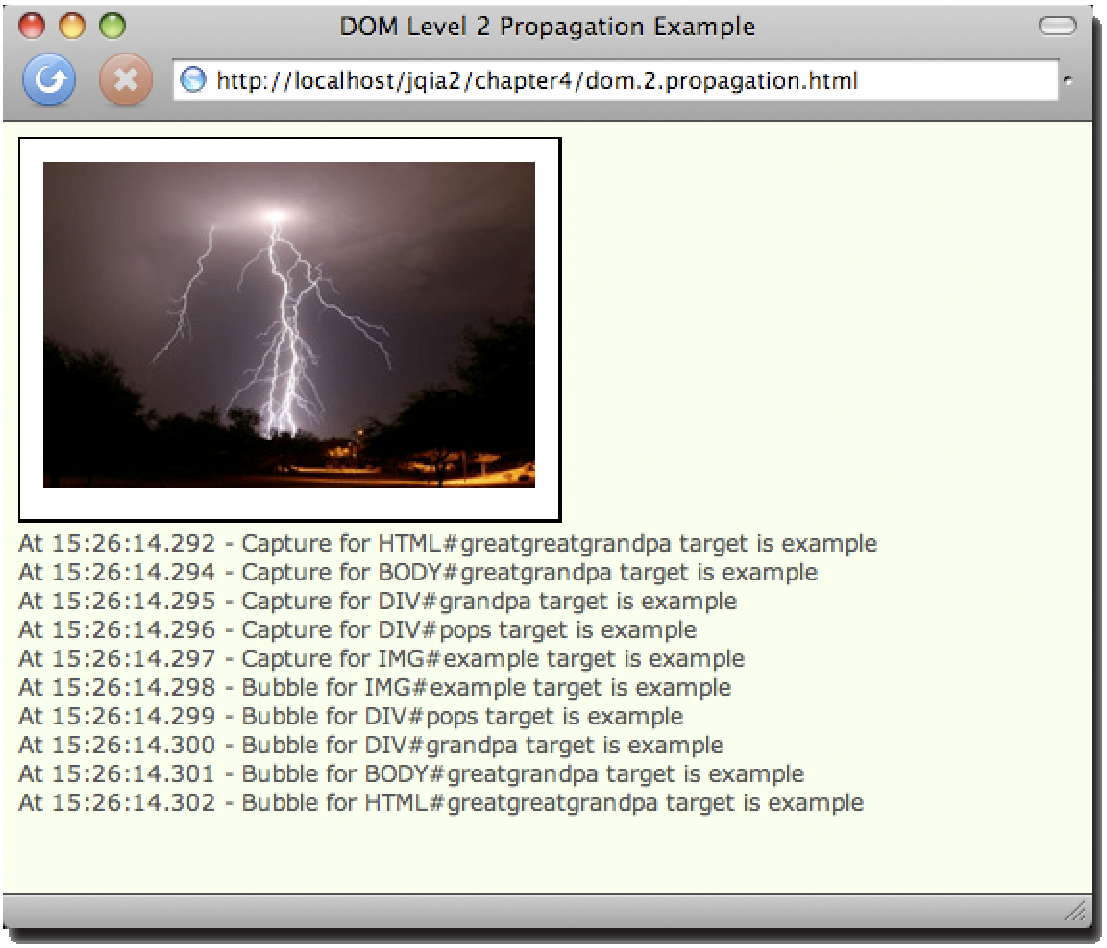


Figure 4.5 Clicking the image results in each handler emitting a console message that identifies the path of the event during both capture and bubble phases

Note that, because we defined both capture and bubble handlers for the target, two handlers were executed for the target and all its ancestor nodes.

Well, now that we've gone through all the trouble to understand that, we should know that capture handlers are hardly ever used in web pages. The simple reason for that is that Internet Explorer doesn't

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

support the DOM Level 2 Event Model. Although it *does* have a proprietary model corresponding to the bubble phase of the Level 2 standard, it doesn't support any semblance of a capture phase.

Before we look at how jQuery is going to help sort all this mess out, let's briefly examine the Internet Explorer Model.

4.1.3 The Internet Explorer Event Model

Internet Explorer (IE6, IE7 and, most disappointingly, even IE8) doesn't provide support for the DOM Level 2 Event Model. Both these versions of Microsoft's browser provide a proprietary interface that closely resembles the bubble phase of the standard model.

Rather than `addEventListener()`, the Internet Explorer Model defines a method named `attachEvent()` for each DOM element. This method, as follows, accepts two parameters similar to those of the standard model:

```
attachEvent(eventName,handler)
```

The first parameter is a string that names the event type to be attached. The standard event names aren't used; the name of the corresponding element property from the DOM Level 0 Model is used—"onclick", "onmouseover", "onkeydown", and so on.

The second parameter is the function to be established as the handler, and as in the Basic Model, the `Event` instance must be fetched from the `window .event` property.

What a mess! Even when using the relatively browser-independent DOM Level 0 Model, we're faced with a tangle of browser-dependent choices to make at each stage of event handling. And when using the more capable DOM Level 2 or Internet Explorer Model, we even have to diverge our code when establishing the handlers in the first place.

Well, jQuery is going to make our lives simpler by hiding these browser disparities from us as much as it possibly can. Let's see how!

4.2 The jQuery Event Model

Although it's true that the creation of Rich Internet Applications requires a hefty reliance on event handling, the thought of writing event-handling code on a large scale while dealing with the browser differences is enough to daunt even the most intrepid of page authors.

We could hide the differences behind an API that abstracts the differences away from our page code, but why bother when jQuery has already done it for us?

jQuery's event implementation, which we'll refer to informally as the jQuery Event Model, exhibits the following features:

- Provides a unified method for establishing event handlers
- Allows multiple handlers for each event type on each element
- Uses standard event-type names: for example, click or mouseover
- Makes the `Event` instance available as a parameter to the handlers
- Normalizes the `Event` instance for the most often used properties

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

- Provides unified methods for event canceling and default action blocking

With the notable exception of support for a capture phase, the feature set of the jQuery Event Model closely resembles that of the Level 2 Model while supporting both standards-compliant browsers and Internet Explorer with a single API. The omission of capture phase should not be an issue for the vast majority of page authors who never use it (or even know it exists) due to its lack of support in IE.

Is it really that simple? Let's find out.

4.2.1 Binding event handlers with jQuery

Using the jQuery Event Model, we can establish event handlers on DOM elements with the `bind()` method. Consider the following simple example:

```
$('img').bind('click',function(event){alert('Hi there!');});
```

This statement binds the supplied inline function as the click event handler for every image on a page. The full syntax of the `bind()` methodis as follows:

Method syntax: bind

`bind(eventType,data,listener)`

Establishes a function as the event handler for the specified event type on all elements in the matched set.

Parameters

`eventType`	(String) Specifies the name of the event type or types for which the handler is to be established. Multiple event types can be specified as a space-separated list. These event types can be name-spaced with a suffix affixed to the event name with a period character. See the remainder of this section for details.
`data`	(Object) Caller-supplied data that's attached to the `Event` instance as a property named `data` for availability to the handler functions. If omitted, the handler function can be specified as the second parameter.
`listener`	(Function) The function that's to be established as the event handler. When invoked, it will be passed the Event instance, and its function context (`this`) is set to the current element of the bubble phase.

Returns

The wrapped set.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Let's put `bind()` into action. Taking the example of listing 4.3 and converting it from the DOM Level 2 Model to the jQuery Model, we end up with the code shown in listing 4.5 and found in the file `chapter4/jquery.events.html`.

Listing 4.5 Estqblishing advanced event handlers without the need for browser-specific code

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>jQuery Events Example</title>
    <link rel="stylesheet" type="text/css" href="../styles/examples.css"/>
    <script type="text/javascript" src="../scripts/jquery-1.3.2.min.js"></script>
    <script type="text/javascript">
      $(function(){
        $('#vstar')                                    #1
          .bind('click',function(event) {
            say('Whee once!');
          })
          .bind('click',function(event) {
            say('Whee twice!');
          })
          .bind('click',function(event) {
            say('Whee three times!');
          });
      });

      function say(text) {
        $('#console').append('<div>'+text+'</div>');
      }
    </script>
  </head>

  <body>
    <img id="vstar" src="vstar.jpg"/>
    <div id="console"></div>
  </body>
</html>
```

#1 Binds three event handlers to the image

Cueballs in code and text

The changes to this code, limited to the body of the ready handler, are minor but significant #1. We create a wrapped set consisting of the target `<img>` element and apply three `bind()`methods to it—remember, jQuery chaining lets us apply multiple methods in a single statement—each of which establishes a click event handler on the element.

Loading this page into a standards-compliant browser and clicking the image results in the display of figure 4.6, which not surprisingly, is the exact same result we saw in figure 4.3 (except for the URL and window caption).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

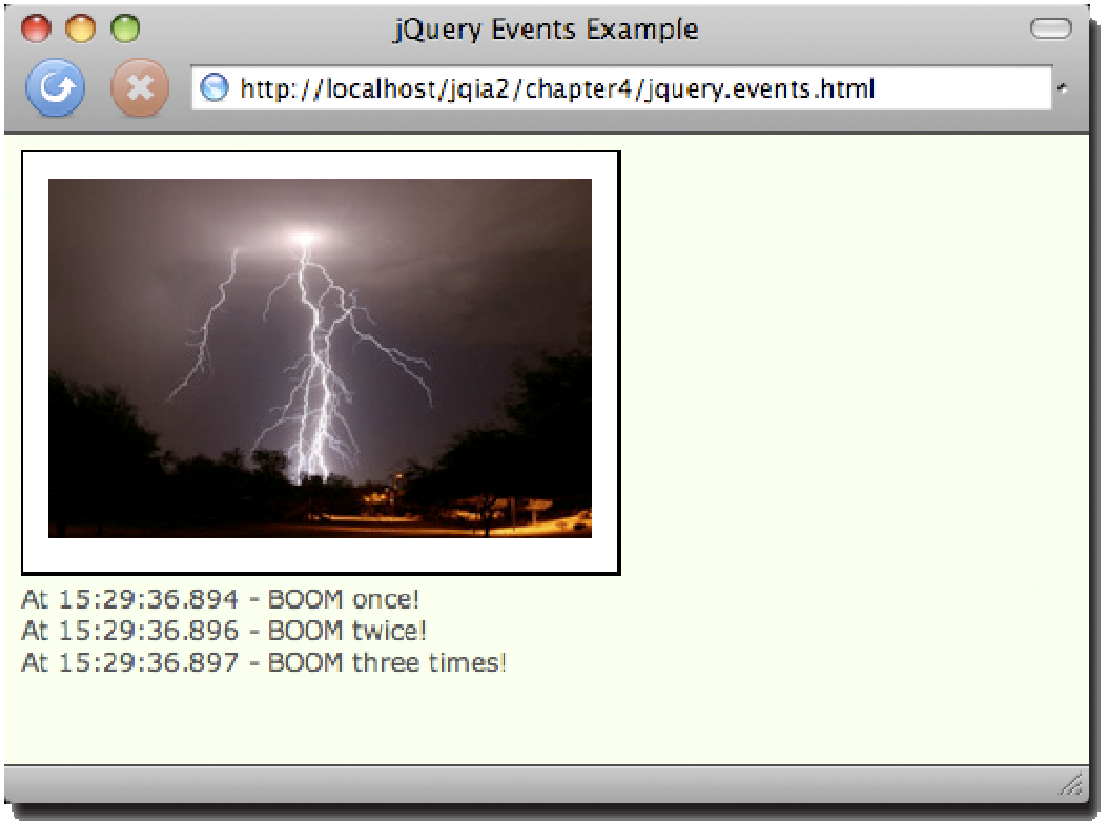


Figure 4.6 Using the jQuery Event Model allows us to specify multiple event handlers just like the DOM Level 2 Model.

But perhaps more importantly, when loaded into Internet Explorer, it also works as shown in figure 4.7. This was not possible using the code from listing 4.3 without adding any browser-specific testing and branching code to use the correct event model for the current browser.

At this point, page authors who have wrestled with mountains of browser-specific event-handling code in their pages are no doubt singing "Happy Days Are Here Again" and spinning in their office chairs. Who could blame them?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

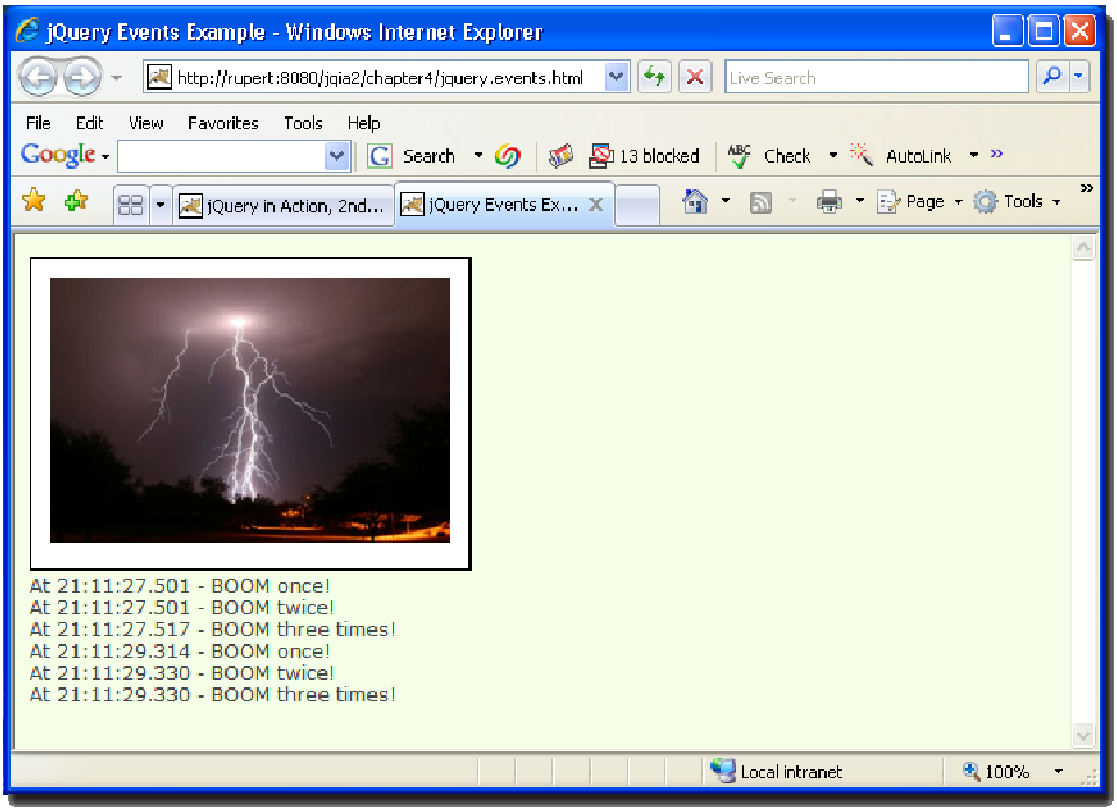


Figure 4.7 The jQuery Event Model allows us to use a unified events API to support events across the standards-compliant browsers as well as Internet Explorer

Another nifty little event handling extra that jQuery provides for us is the ability to group event handlers by assigning them to a namespace. Unlike conventional name-spacing (which assigns namespaces via a prefix), the event names are name-spaced by adding a *suffix* to the event name separated by a period character. In fact, if you'd like, you can use multiple suffixes to place the event into multiple namespaces.

By grouping event bindings in this way, we can easily act upon them later as a unit.

Take, for example, a page that has two modes: a display mode and an edit mode. When in edit mode, event listeners are placed on many of the page elements, but these listeners are not appropriate for display mode and need to be removed when the page transitions out of edit mode. We could namespace the edit mode events with code such as

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
$('#thing1').bind('click.editMode',someListener);
$('#thing2').bind('click.editMode',someOtherListener);
  ...
$('#thingN').bind('click.editMode',stillAnotherListener);
```

By grouping all these bindings into a namespace named `editMode`, we can later operate upon them as a whole. For example, when the page leaves edit mode and it comes time to remove all the bindings we could do this easily with

```
$('*').unbind('click.editMode');
```

This will remove all `click` bindings (the explanation of the `unbind()` method is coming up in the next section) in the namespace `editMode` for all elements on the page.

In addition to the `bind()`method, jQuery provides a handful of shortcut methodsto establish specific event handlers. Because the syntax of each of these methodsis identical except for the method name of the method, we'll save some space and present them all in the following single syntax descriptor:

Method syntax: *specific event binding*

***eventTypeName*(listener)**

Establishes the specified function as the event handler for the event type named by the method's name. The supported methods are as follows:

- `blur`
- `change`
- `click`
- `dblclick`
- `error`
- `focus`
- `keydown`
- `keypress`
- `keyup`
- `load`
- `mousedown`
- `mouseenter`
- `mouseleave`
- `mousemove`
- `mouseout`
- `mouseover`
- `mouseup`
- `resize`
- `scroll`
- `select`
- `submit`
- `unload`

Note that when using these shortcut methods, we cannot specify a data value to be placed in the `event.data` property.

Parameters

`listener` (Function) The function that's to be established as the event handler.

Returns

The wrapped set.

jQuery also provides a specialized version of the `bind()`method, named `one()`, that establishes an event handler as a one-shot deal. Once the event handler executes the first time, it's automatically removed as an event handler. Its syntax is similar to the `bind()`method and is as follows:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Method syntax: one

`one(eventType,data,listener)`

Establishes a function as the event handler for the specified event type on all elements in the matched set. Once executed, the handler is automatically removed.

Parameters

`eventType`	(String) Specifies the name of the event type for which the handler is to be established.
`data`	(Object) Caller-supplied data that's attached to the `Event` instance for availability to the handler functions. If omitted, the handler function can be specified as the second parameter.
`listener`	(Function) The function that's to be established as the event handler.

Returns

The wrapped set.

These methods give us many choices to bind an event handler to matched elements. And once a handler is bound, we may eventually need to remove it.

4.2.2 Removing event handlers

Typically, once an event handler is established, it remains in effect for the remainder of the life of the page. But particular interactions may dictate that handlers be removed based on certain criteria. Consider, for example, a page where multiple steps are presented, and once a step has been completed, its controls revert to read-only.

For such cases, it would be advantageous to remove event handlers under script control. We've seen that the `one()`method can automatically remove a handler after it has completed its first (and only) execution, but for the more general case where we'd like to remove event handlers under our own control, jQuery provides the `unbind()`method.

The syntax of `unbind()` is as follows:

Method syntax: unbind

`unbind(eventType,listener)`

`unbind(event)`

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Removes events handlers from all elements of the wrapped set as specified by the optional passed parameters. If no parameters are provided, all listeners are removed from the elements.

Parameters

`eventType`	(String) If provided, specifies that only listeners established for the specified event type are to be removed.
`listener`	(Function) If provided, identifies the specific listener that's to be removed.
`event`	(Event) Removes the listener that triggered the event described by this Event instance.

Returns

The wrapped set.

This method can be used to remove event handlers from the elements of the matched set at various levels of granularity. All listeners can be removed by omitting any parameters, or listeners of a specific type can be removed by providing just that event type.

Specific handlers can be removed by providing a reference to the function originally established as the listener. For this to be possible, a reference to the function must be retained when binding the function as an event listener in the first place. For this reason, when a function that's eventually to be removed as a handler is originally established as a listener, it's either defined as a top-level function (so that it can be referred to by its top-level variable name) or a reference to it is retained by some other means. Supplying the function as an anonymous inline reference would make it impossible to later reference the function in a call to `unbind()`.

So far, we've seen that the jQuery Event Model makes it easy to establish (as well as remove) event handlers without worries about browser differences, but what about writing the event handlers themselves?

4.2.3 Inspecting the Event instance

When an event handler established with the `bind()`method (or any of its related convenience methods) is invoked, an `Event` instance is passed to it as the first parameter to the function regardless of browser, eliminating the need to worry about the `window.event` property under Internet Explorer. But that still leaves us with dealing with the divergent properties of the `Event` instance, doesn't it?

Thankfully, no, because truth be told, jQuery doesn't really pass the Event instance to the handlers.

Screech! (sound of needle being dragged across record).

Yes, we've been glossing over this little detail because, up until now, it hasn't really mattered. But now that we've advanced to the point where we're going to examine the instance within handlers, the truth must be told!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

In reality, jQuery defines an object of type `jQuery.Event` that it passes to the handlers. But we can be forgiven our simplification, because jQuery copies most of the original `Event` properties to this object. As such, if you only look for the properties that you expected to find on Event, the object is almost indistinguishable from the original Event instance.

But that's not the important aspect of this object – what's really valuable, and the reason that this object exists, is to hold a set of normalized values and methods that we can use independently of the containing browser, ignoring the differences in the Event instance.

Table 4.1 lists the jQuery.`Event` properties and methods that are safe to access in a platform-independent manner.

Table 4,1 Browser-indeoendent jQuery.Event properties

Name	Description
Properties	
`altKey`	Set to `true` if the Alt key was pressed when the event was triggered, `false` if not. The Alt key is labeled Option on most Mac keyboards.
`ctrlKey`	Set to `true` if the Ctrl key was pressed when the event was triggered, `false` if not.
`currentTarget`	The current element during the bubble phase. This is the same object that is set as the function context of the event handler.
`data`	The value, if any, passed as the second parameter to the `bind()`method when the handler was established.
`metaKey`	Set to `true` if the Meta key was pressed when the event was triggered, `false` if not. The Meta key is the Ctrl key on PCs and the Command key on Macs.
`pageX`	For mouse events, specifies the horizontal coordinate of the event relative from the page origin.
`pageY`	For mouse events, specifies the vertical coordinate of the event relative from the page origin.
`relatedTarget`	For mouse movement events, identifies the element that the cursor left or entered when the event was triggered.
`screenX`	For mouse events, specifies the horizontal coordinate of the event relative from the screen origin.
`screenY`	For mouse events, specifies the vertical coordinate of the event relative from the screen origin.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`shiftKey`	Set to `true` if the Shift key was pressed when the event was triggered, `false` if not.
`result`	The most recent non-undefined value returned from a previous event handler.
`target`	Identifies the element for which the event was triggered.
`timestamp`	The timestamp, in milliseconds, when the `jQuery.Event` instance was created.
`type`	For all events, specifies the type of event that was triggered (for example, click). This can be useful if you're using one event handler function for multiple events.
`which`	For keyboard events, specifies the numeric code for the key that caused the event, and for mouse events, specifies which button was pressed (1 for left, 2 for middle, 3 for right). This should be used instead of `button`, which can't be relied on to function consistently across browsers.

Methods

`preventDefault()`	Prevents any default semantic action (such as form submission, link redirection, checkbox state change, etc) from occurring.
`stopPropagation()`	Stops any further propagation of the event up the DOM tree. Additional events on the current target are not affected. Works with browser-defined events as well as custom events.
`stopImmediatePropagation()`	Stops all further event propagation including additional events on the current target.
`isDefaultPrevented()`	Returns `true` if the `preventDefault()` method has been called on this instance.
`isPropagationStopped()`	Returns `true` if the `stopPropagation()` method has been called on this instance.
`isImmediatePropagationStopped()`	Returns `true` if the `stopImmediatePropagation()` method has been called on this instance.

It's important to not that the `keypress` property isn't reliable cross-browser for non-alphabetic characters. For instance, the left arrow key has a code of 37, which works reliably on keyup and keydown events, but Safari returns nonstandard results for these keys on a keypress event.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

We can get a reliable, case-sensitive character code in the `which` property of keypress events. During keyup and keydown events, we can only get a case-insensitive key code (so *a* and *A* both return 65), but we can determine case by checking `shiftKey`.

Also, if we want to stop the propagation of the event (but not *immediate* propagation), as well as cancel its default behavior, we can simply return `false` as the return value of the listener function.

In addition to allowing us to set up event handling in a browser-independent manner, jQuery provides a set of methods that gives us the ability to trigger event handlers under script control. Let's look at those.

4.2.5 Triggering event handlers

Event handlers are designed to be invoked when browser or user activity triggers the propagation of their associated events through the DOM hierarchy. But there may be times when we want to trigger the execution of a handler under script control. We could define such event handlers as top-level functions so that we can invoke them by name, but as we've seen, defining event handlers as inline anonymous functions is much more common and so darned convenient! Moreover, simply calling an event handler as a function doesn't cause semantic actions or bubbling to occur.

jQuery has provided means to assist us by defining methods that will automatically trigger event handlers on our behalf under script control. The most general of these methods is `trigger()`, whose syntax is as follows:

Method syntax: trigger

`trigger(eventType,data)`

Invokes any event handlers established for the passed event type for all matched elements.

Parameters

`eventType`	(String) Specifies the name of the event type for which handlers are to be invoked. This includes name-spaced, events. You can append the exclamation point (`!`) to the event type to prevent name-spaced events from triggering.
`data`	(Any) Data to be passed to the handlers as the second parameter (after the event instance).

Returns

The wrapped set

The `trigger()` method, as well as the convenience methods that we'll introduce in a moment, does its best to simulate the event to be triggered, including propagation through the DOM hierarchy and the execution of semantic actions.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Each handler called is passed a populated instance of `jQuery.Event`. Because there's no real event, properties that report event-specific values, such as the location of a mouse event, or the key of a keyboard event, have no value. The `target` property is set to reference the element of the matched set to which the handler was bound.

Just as with actual events, triggered event propagation can be halted via a call to the `jQuery.Event` instance's `stopPropagation()` method, or a `false` value an be returned from ay of the invoked handlers.

NOTE

The `data` parameter passed to the `trigger()` method is *not* the same as the one passed when a handler is established. The latter is placed into the `jQuery.Event` instance as the `data` property, the value passed to `trigger()` (and, as we're about to see, `triggerHandler()`) is passed as a parameter to the listeners. This allows both data values to be used without conflicting with each other.

For cases where we want to trigger a handler, but not cause propagation of the event and execution of semantic actions, jQuery provides the `triggerHandler()` method, which looks and acts just like `trigger()` except that no bubbling or semantic actions will occur, and no events bound by `live()` will be triggered.

Method syntax: triggerHandler

`triggerHandler(eventType,data)`

Invokes any event handlers established for the passed event type for all matched elements without bubbling, semantic actions, or live events.

Parameters

`eventType`	(String) Specifies the name of the event type for which handlers are to be invoked.
`data`	(Any) Data to be passed to the handlers as the second parameter (right after the event instance).

Returns

The wrapped set

In addition to the `trigger()` and `triggerHandler()` methods, jQuery provides convenience methods for triggering most of the event types. The syntax for all these methods is exactly the same except for the method name, and that syntax is described as follows:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Method syntax: *eventName*

***eventName*()**

Invokes any event handlers established for the named event type for all matched elements. The supported methods are as follows:

- blur
- change
- click
- dblclick
- error
- focus
- keydown
- keypress
- keyup
- select
- submit

Parameters

none

Returns

The wrapped set.

In addition to binding, unbinding, and triggering event handlers, jQuery offers higher-level functions that further make dealing with events on our pages as easy as possible.

4.2.6 Other event-related methods

There are often interaction styles that are commonly applied to pages in Rich Internet Applications and are implemented using combinations of behaviors. jQuery provides a few event-related convenience methods that make it easier to use these interaction behaviors on our pages. Let's look at them.

TOGGLING LISTENERS

The first of these is the `toggle()` method, which establishes a circular progression of click event handlers that are applied on each subsequent click event. In other words, on the first click event, the first registered handler is called, on the second click, the second is called, on the third click the third is called and so on. When the end of the list of established handlers is reached, the first handler become the next in line. Its syntax is as follows:

Method syntax: toggle

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`toggle(listener1,listener2, …)`

Establishes the passed functions as a circular list of click event handlers on all elements of the wrapped set. The handlers are called in order on each subsequent click event.

Parameters

`listenerN`	(Function) Ffunctions that serves as the click event handlers for subsequent clicks. An arbitrary number of functions of two or greater can be established.

Returns

The wrapped set

A common use for this convenience method is to toggle the enabled state of an element back and forth on each odd or even click. For this, we'd supply two handlers; on for the odd clicks, and one for the even clicks.

But this method can also be used to create a progression through and arbitrary number of clicks of two or greater. Let's consider an example.

We'll imagine that we have a site in which we want users to be able to view images in one of three sizes: small, medium or large. The interaction will take place through a simple series of clicks. Clicking on the image bumps it up to its next bigger size, until we reach the largest size where will revert back to the smallest.

Examine the progression shown in the time-lapse screen shots of figure 4.8. Each time the image is clicked, it grows to the next bigger size. If one more click were to be made, the image would revert to the smallest size.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Figure 4.8 jQuery's toggle() method lets us pre-define a progression of behaviors for click events

The code for this page is shown in Listing 4.6, and can be found in file `chapter4/toggle.html`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>jQuery .toggle() Example</title>
    <link rel="stylesheet" type="text/css" href="../styles/examples.css"/>
    <style type="text/css">
      img {
        cursor: pointer;
      }
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
    </style>
    <script type="text/javascript" src="../scripts/jquery-1.3.2.min.js"></script>
    <script type="text/javascript">
      $(function(){

        $('img[src*=small]').toggle(                                    #A
            function() {
              $(this).attr('src',$(this).attr('src').replace(/small/,'medium'));
            },
            function() {
              $(this).attr('src',$(this).attr('src').replace(/medium/,'large'));
            },
            function() {
              $(this).attr('src',$(this).attr('src').replace(/large/,'small'));
            }
        );

      });
    </script>
  </head>

  <body>

    <div>Click on the image to change its size.</div>
    <div>
      <img src="hibiscus.small.jpg" alt="Hibiscus"/>
    </div>

  </body>
</html>
```

#A Establishes a progression of handlers

Cueball in code

If you're like your authors and pay attention to the names of things, you might wonder why this method is named `toggle()` when that really only makes sense for the case when only two handlers are established. The reason is that in earlier versions of jQuery, this method was limited to only two handlers and was later expanded to accept an arbitrary number of handlers.

Another common multi-event scenario that's frequently employed in Rich Internet Applications involves mousing into and out of elements.

HOVERING OVER ELEMENTS

Events that inform us when the mouse pointer has entered an area, as well as when it has left that area, are essential to building many of the user interface elements that are commonly presented to users on our pages. Among these element types, cascading menus used as navigation systems for some web applications are a common example.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

A vexing behavior of the mouseover and mouseout event types often hinders the easy creation of such elements when a mouseout event fires as the mouse is moved over an area and its children. Consider the display in figure 4.9 (available in the file `chapter4/hover.html`).

This page displays two identical (except for naming) sets of areas: an outer area and an inner area. Load this page into your browser as you follow the rest of this section.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

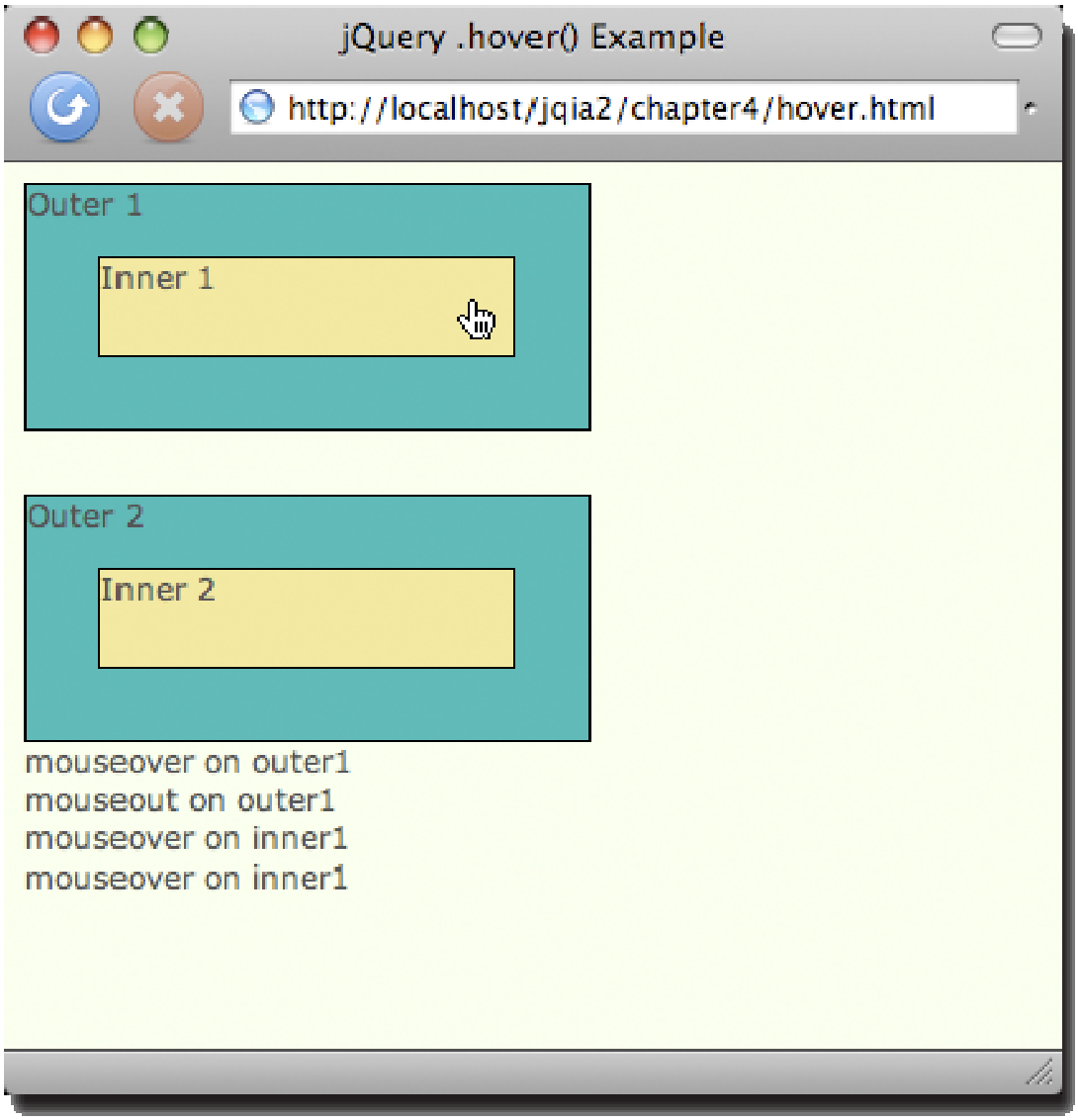


Figure: 4.9 This page helps demonstrate when mouse events fire as the mouse pointer is moved over an area and its children

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

For the top set, the following script in the ready handler establishes handlers for the mouseover and mouseout events:

```
$('#outer1')
 .bind('mouseover mouseout',report);
$('#inner1')
 .bind('mouseover mouseout',report);
```

This statement establishes a function named `report` as the event handler for both the mouseover and mouseout events. The `report()` function is defined as follows:

```
function report(event) {
  $('#console').append('<div>'+event.type+'</div>');
}
```

This listener merely adds a `<div>` element containing the name of the event that fired to a `<div>` named `console` that's defined at the bottom of the page, allowing us to see when each event fires.

Now, let's move the mouse pointer into the area labeled Outer 1 (being careful not to enter Inner 1). We'll see (from looking at the bottom of the page) that a mouseover event has fired. Move the pointer back out of the area. As expected, we'll see that a mouseout event has fired.

Let's refresh the page to start over, clearing the console.

Now, move the mouse pointer into Outer 1 (noting the event), but this time continue inward until the pointer enters Inner 1. As the mouse enters Inner 1, a mouseout event fires for Outer 1. If we wave our pointer over the inner area, we'll see a flurry of mouseout and mouseover events. This *is* the defined behavior, even if it's rather unintuitive. Even though the pointer is still within the bounds of Outer 1, when the pointer enters a contained element, the event model considers the transition from the area of Outer 1 for its contained element to be leaving the outer area.

Expected or not, we don't always want that behavior. Often, we want to be informed when the pointer leaves the bounds of the outer area and don't care whether the pointer is over a contained area or not.

Luckily, some of the major browsers support a non-standard pair of mouse events, mouseenter and mouseleave, first introduced by Microsoft in Internet Explorer. This event pair acts slightly more intuitively, not firing a mouseleave event when moving from an element to a descendant of that element. For browsers not supporting these events, jQuery emulates them so that they work the same across all browsers.

Using jQuery we could establish handlers for this set of events using:

```
$(element).mousenter(function1).mouseleave(function2);
```

But jQuery also provides a single method that makes it even easier: `hover()`.The syntax of this method is as follows:

Method syntax: hover

```
hover(enterListener,leaveListener)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Establishes handlers for the mouseenter and mouseleave events for matched elements. These handlers only fire when the area covered by the elements is entered and exited, ignoring transitions to child elements.

Parameters

`enterListener`	(Function) The function to become the mouseenter handler.
`leaveListener`	(Function) The function to become the mouseleave handler.

Returns

The wrapped set.

We use the following script to establish mouse event handlers for the second set of areas (Outer 2 and its Inner 2 child) on the hover.html example page:

```
$('#outer2').hover(report,report);
```

As with the first set of areas, the `report()` function is established as the mouseenter and mouseleave handlers for Outer 2. But unlike the first set of areas, when we pass the mouse pointer over the boundaries between Outer 2 and Inner 2, neither of these handlers (for Outer 2) is invoked. This is useful for those situations where we have no need for parent handlers to react when the mouse pointer passes over child elements.

With all these event-handling tools under our belts, let's use what we've learned so far and look at an example page that makes use of them, as well as some of the other jQuery techniques that we've learned from previous chapters!

4.3 Putting events (and more) to work

Now that we've covered how jQuery brings order to the chaos of dealing with disparate event models across browsers, let's work an example page that puts the knowledge that we've gained so far to use. This example uses not only events but also some jQuery techniques that we've explored in earlier chapters, including some heavy-weight jQuery method chains. For this comprehensive example, let's pretend that we're videophiles whose DVD collection, numbering in the thousands, has become a huge problem. Not only has organization become an issue, making it hard to find a DVD quickly, but all those DVDs in their cases have become a storage problem that has taken over way too much space and threatened to get us thrown out of the house if the problem is not solved.

We'll posit that we solved the storage side of the problem by buying DVD binders that hold one hundred DVDs each in much less space than the comparable number of DVDs in their cases. But while that saved us from having to sleep on a park bench, organization of the DVD discs is still an issue. How will we find a DVD that we are looking for without having to manually flip through each binder until we find the one we're seeking?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

We can't do something like sort the DVDs in alphabetic order to help quickly locate a specific disc. That would mean that every time we buy a new DVD, we'd need to shift all the discs in perhaps dozens of binders to keep the collection sorted. Imagine the job ahead of us after we buy *Abbott and Costello Go to Mars*!

Well, we've got computers, we've got the know-how to write web applications, and we've got jQuery! So we'll solve the problem by writing a DVD database program to help keep track of what DVDs we have, and where they are.

Let's get to work!

n4.3.1 Filtering large data sets

Our DVD database program is faced with the same problem facing many other applications, web-delivered or otherwise. How do we allow our users (in this case ourselves) to quickly find the information that they seek?

We could be all low-tech about it and just display a sorted list of all the titles; but that would still be painful to scroll through with anything more than a handful of entries, and besides, we want to learn how to do it *right* so that we can apply what we learn to real, customer-facing applications.

So no short cuts!

Obviously, designing the entire application would be well beyond the scope of this chapter, so what we'll concentrate on is developing a control panel that will allow us to specify *filters* with which we can tune the titles that will be returned when we perform a database search.

We'll want the ability to filter on the DVD title, of course. But we'll also add the ability to filter the search based upon the year that the movie was released, its category, the binder in which we placed the disc, and even whether we've viewed the movie yet or not. (Which will help answer the commonly asked question "What should I watch tonight?")

Your initial reaction may be to wonder what the big deal is. After all, we can just put up a number of fields and be done with it, right?

Well, not so fast. A single field for something like the title is fine if, for example, we wanted to find all movies with the term "creature" in their title. But what if we want to search for either of "creature" or "monster"? Or only movies released in 1957 or 1972?

In order to provide a robust interface for specifying filters, we're going to need to be able to allow the user to specify multiple filters for either the same or different properties of the DVD. And rather than try to guess how many will be needed, we'll be all swank about it and create them on-demand.

For our interface we're going to steal a page from Apple's user interface playbook, and model our interface on the way that filters are specified in many Apple applications. (If you're an iTunes user, check out how Smart Playlists are created for an example.)

Each filter, one per "line", is identified by a dropdown that specifies the field that is to be filtered. Based upon the type of that field (strings, dates, numbers, and even Booleans) the rest of the controls on the line are displayed as appropriate to capture information about the filter.

The user is given the ability to add as many of these filters as they like, or to remove previously specified filters.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

A picture being worth a thousand words, study the time-progression display of figures 4.10a through 4.10c. They show the filter panel that we'll be building: (a) when initially displayed, (b) after a filter has been specified, and (c) after a number of filters have been specified.

Figure 4.10a The display initial shows a single, unconfigured filter

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Figure 4.10b After a filter type is selected, its qualifier controls are added

Figure 4.10c The user can add as many filters as required

As we can see by inspecting the interactions shown in figures 4.10a through 4.10c, there's going to be a lot of element creation on-the fly. Let's take a few moments to discuss how we're going to go about that.

4.3.2 Element creation by template replication

We can readily see that to implement this filtering control panel, we're going to need to create a fair number of elements in response to various events. For example, we'll new to create a new filter entry whenever the user clicks the Add Filter button, and new controls that qualify that filter whenever a specific field is selected.

No problem! In the previous chapter we saw how easy jQuery makes it to dynamically create elements using the $( ) function. And while we'll do some of that in our example, we're also going to explore some alternatives.

When we're dynamically creating lots of elements, all the code necessary to create those elements and stitch together their relationships can get a bit unwieldy and a bit difficult to maintain, even with jQuery's assistance. (Without jQuery assist, it can be a complete nightmare!) What'd be great would be if we could

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

create a "blueprint" of the complex markup using HTML, and then replicate it whenever we needed an instance of the blueprint.

Yearn, no more! The jQuery `clone()` method gives us just that ability.

The approach that we are going to take is to create sets of "template" markup that represent the HTML fragments that we'd like to replicate, and use the `clone()` method whenever we need to create an instance of that template. We don't want these templates to be visible to the end user, so we'll sequester them in a `<div>` element at the end of the page that's hidden from view using CSS.

Let's consider the combination of the "X" button and drop-down that identifies the filterable fields as an example. We'll need to create an instance of this combination every time that the user clicks the Add Filter button. The jQuery code to create such a button and `<select>` element, along with its child `<option>` elements, could be considered a tad long, but would not be too onerous to write or maintain. But it'd be easy to envision that anything more complex would get unwieldy quickly.

Using our template technique, and placing the template markup for that button and drop-down in a parent `<div>` used to hide all the templates, we create markup as follows:

```
<!-- hidden templates -->
<div id="templates">                                                    #1

  <div class="template filterChooser">                                  #2
    <button type="button" class="filterRemover" title="Remove this filter">X</button>

    <select name="filter" class="filterChooser" title="Select a property to filter">
      <option value="" data-filter-type="" selected="selected">-- choose a filter --
</option>
      <option value="title" data-filter-type="stringMatch">DVD Title</option>
      <option value="category" data-filter-type="stringMatch">Category</option>
      <option value="binder" data-filter-type="numberRange">Binder</option>
      <option value="release" data-filter-type="dateRange">Release Date</option>
      <option value="viewed" data-filter-type="boolean">Viewed?</option>
    </select>
  </div>

  <!—- more templates go here -->

</div>
```

#1 Encloses and hides all templates
#2 Defines the filterChooser template

Cueballs in code and text

The outer `<div>` with `id` of "templates" serves as a container for all our templates and will be given a CSS display rule of `none` to prevent it from being displayed in the browser (`#1`).

Within this container, we define another `<div>` which we give the classes "template" and "filterChooser" (`#2`). We'll use the "template" class to identify templates in general, and the "filterChooser" class identifies this particular template type. We'll see how these classes are used in the code shortly – remember, classes aren't just for CSS anymore!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Also note that each `<option>` in the `<select>` has been given a custom attribute: `data-filter-type`. We'll use this value to determine what type of filter controls need to be used for the selected filter field.

Based upon which filter type is identified, we'll populate the remainder of the filter entry "line" with qualifying controls that are appropriate for the filter type.

For example, if the filter type is "stringMatch", we'll want to display a text field into which the user can type a text search term, and a drop-down giving them options on how that term is to be applied.

We've set up the template for this set of controls as follows:

```
<div class="template stringMatch">
  <select name="stringMatchType">
    <option value="*">contains</option>
    <option value="^">starts with</option>
    <option value="$">ends with</option>
    <option value="=">is exactly</option>
  </select>
  <input type="text" name="term"/>
</div>
```

Again, we've used the "template" class to identify the element as a template, and we've flagged the element with the class "stringMatch". We've purposely made it such that this class matches the `data-filter-type` value on the field chooser drop-down.

Replicating these templates whenever, and wherever, we want is easy using the jQuery knowledge under our belts. Let's say that we want to append a template instance to the end of an element that we have a reference to in a variable named `whatever`. We could write:

```
$('div.template.filterChooser').children().clone().appendTo(whatever);
```

In this statement, we select the template container to be replicated (using those convenient classes we placed on the template markup), select the child elements of the template container (we don't want to replicate the `<div>`, just its contents), make clones of those children, and then attach them to the end of the contents of the element identified by `whatever`.

See why we keep emphasizing the power of jQuery method chains?

Inspecting the options of the `filterChooser` dropdown, we see that we have a number of other filter types defined: "numberRange", "dateRange", and "boolean". So we define qualifying control templates for those filter types as well with:

```
<div class="template numberRange">
  <input type="text" name="numberRange1" class="numeric"/> <span>through</span>
  <input type="text" name="numberRange2" class="numeric"/>
</div>

<div class="template dateRange">
  <input type="text" name="dateRange1" class="dateValue"/> <span>through</span>
  <input type="text" name="dateRange2" class="dateValue"/>
</div>

<div class="template boolean">
  <input type="radio" name="booleanFilter" value="true" checked="checked"/>
<span>Yes</span>
  <input type="radio" name="booleanFilter" value="false"/> <span>No</span>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
</div>
```

OK. Now that we've got our replication strategy defined, let's take a look at the primary markup.

4.3.3 Setting up the mainline markup

If we refer back to figure 4.10a, we see that the initial display of our DVD search page is pretty simple: a few headers, a first filter instance, and a couple of buttons. Let's take a look at the HTML markup to achive that:

```
<div id="pageContent">

  <h1>DVD Ambassador</h1>
  <h2>Disc Locator</h2>

  <form id="filtersForm" action="/fetchFilteredResults" method="post">

    <fieldset id="filtersPane">
      <legend>Filters</legend>
      <div id="filterPane"></div>                                    #1
      <div class="buttonBar">
        <button type="button" id="addFilterButton">Add Filter</button>
        <button type="submit" id="applyFilterButton">Apply Filters</button>
      </div>
    </fieldset>

    <div id="resultsPane">
      <span class="none">No results displayed</span>                 #2
    </div>

  </form>

</div>
```

#1 Container for filter instances
#2 Container for search results

Cueballs in code and text

There's nothing too surprising in that markup – or is there? Where, for example, is the markup for the initial filter dropdown? We've set up a container in which the filters will be placed (#1), but it's initially empty. Why?

Well, we're going to need to be able to populate new filters dynamically – which we'll be getting to in just a moment – so why do the work in two places? As we shall see, we'll be able to leverage the dynamic code to initially populate the first filter, so we don't need to explicitly create it in the static markup.

One other thing that should be pointed out is that we've set aside a container to receive the results (#2) (the fetching of which is beyond the scope of this chapter), and that we've placed these results *inside* the form so that the results themselves can contain form controls (for sorting, paging, and so on).

OK. So we have our very simple, mainline HTML laid out, and we have a handful of hidden templates that we can use to quickly generate new elements via replication. Let's finally get to writing the code that will apply the behavior to our page!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

4.3.4 Adding new filters

Upon a click of the Add Filter button, we need to add a new filter to the element that we've set up to receive them, which we've identified with the id of filterPane.

Recalling how easy it is to establish event handlers using jQuery, it should be an easy matter to add a click handler to the Add Filter button. But wait! There's something we forgot to consider!

We've already seen how we're going to replicate form controls as users add filters, and we've got a good strategy for easily creating multiple instances of these controls. But eventually, we're going to have to submit these values to the server so it can look up the filtered results in the database. And if we just keep copying the same name attributes over and over again for our controls, the server is just going to get a jumbled mess without knowing what qualifiers belong to which filters!

To help the server-side code out (there's a good chance we'll be writing it ourselves in any case) we're going to append a unique suffix to the name attributes for each filter entry. We'll keep it simple and use a simple counter, so that the first filter's controls will all have ".1" appended to their names, the second set ".2", and so on. That way, the server-side code can group them together by suffix when they arrive as part of the request.

To keep track of how many filters we've added (so we can use the count as the suffix for subsequent filter names), we'll create a global variable, initialized to zero, as follows:

```
var filterCount = 0;
```

"Global variable? I thought those were evil?", I hear you ask.

Global variables can be a problem *when used incorrectly*. But in this case, this is truly a global value that represents a page-wide concept, and will never create any conflicts as all aspects of the page will want to access this *single* value in a consistent fashion.

With that set up, we're ready to establish the click handler for the Add Filter button by adding the following code to a ready handler (remember, we don't want to start referencing DOM elements until after we *know* that they've been created):

```
$('#addFilterButton').click(function(){                    #1
  var filterItem = $('<div>')                              #2
    .addClass('filterItem')
    .appendTo('#filterPane');
    .data('suffix','.' + (filterCount++));
  $('div.template.filterChooser')                          #3
      .children().clone().appendTo(filterItem)
      .trigger('adjustName');                              #4
});
```

#1 Establishes the click handler
#2 Creates the filter entry block
#3 Replicates the filter dropdown template
#4 Triggers custom event

Cueballs in code and text

While this compound statement may look complicated at first glance, it actually accomplishes a great deal without a whole lot of code. Let's break it down one step at a time.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

The first thing that we do in this code is to establish a click handler on the Add Filter button (#1) by using the jQuery `click()` method. It's within the function passed to this method, which will get invoked when the button is clicked, that all the interesting stuff happens.

Since a click of the Add Filter button is going to, well, add a filter, we create a new container for the filter to reside within (#2). We give it the class `filterItem` not only for CSS styling, but to be able to locate these elements later in code. After the element is created, it is appended to the master filter container that we created with the `id` value of `filterPane`.

We also need to record the suffix that we want to add to the control names that will be placed within this container. This value will be used when it comes time to adjust the names of the controls, and this is a good example of a value that is *not* suited for a global variable. Each filter container (class `filterItem`) will have its own suffix, and so trying to record this value globally will require some sort of complex array or map construct so that the various values don't step all over each other.

Rather, we'll simply avoid the whole mess by recording the suffix value on the elements themselves using the very handy jQuery `data()` method. Later, when we need to know what suffix to use for a control, we'll simply look at this data value on its container and won't have to worry about getting it confused with the values recorded on other containers.

The code fragment

```
.data('suffix','.' + (filterCount++))
```

formats the suffix value using the current value of the `filterCount` variable, then increments that value. The value is attached to the `filterItem` container using the name `suffix`, and we'll be able to later retrieve it by that name whenever we need it.

The final statement of the click handler (#3) replicates the template that we set up containing the filter dropdown using the approach that we discussed in the previous section. You might think that the job is over at this point, but after the cloning and appending, we execute (#4) the following fragment:

```
.trigger('adjustName')
```

Recalling the `trigger()` method, we recollect that it is used to trigger an event handler. In this case, the event is "adjustName".

"adjustName"?

If you thumb through your specifications, you'll find this event listed nowhere! It's not a standard event defined anywhere *but in this page*. What we've done with this code is to trigger a *custom event*.

A custom event is a very useful concept where we can attach code to an element as a handler for these custom events, and cause them to execute by triggering the event. The beauty of this approach, as opposed to directly calling code, is that we can register the custom handlers in advance, and by simply triggering the event, cause any registered handlers to be executed, without having to know where they've been established.

PATTERN ALERT!

The custom event capability in jQuery is an instance of the Observer pattern; sometimes referred to as the Publish/Subscribe pattern. We *subscribe* an element to a particular event by establishing a handler for that event on that element, and then when the event is *published* (triggered), any subscribed

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

elements in the event hierarchy automatically have their handlers invoked. This can greatly reduce the complexity of code by reducing the *coupling* necessary.

OK, so that will *trigger* the custom event, but we need to define the handler for that event, so also within the ready handler, we establish the code to adjust the control names of the filters:

```
$('.filterItem [name]').live('adjustName',function(){
  var suffix = $(this).closest('.filterItem').data('suffix');
  if (/(\w)+\.(\d)+$/.test($(this).attr('name'))) return;
  $(this).attr('name',$(this).attr('name')+suffix);
});
```

Here we see a use of the `live()` method to pro-actively establish event handlers. The input elements with `name` attributes will be popping into and out of existence whenever a filter is added or removed, so we employ `live()` to automatically establish and remove the handlers as necessary. That way, we set it up *once*, and jQuery will handle the details whenever an item that matches the ".filterItem [name]" selector is created or destroyed. How easy is that?

We specify our custom event name of "adjustName" and supply a handler to be applied whenever we trigger the custom event (Because it's a custom event, there's never a possibility for it to be triggered by any user activity the way that, say, a click handler can be.)

Within the handler we obtain the suffix that we recorded on the `filterItem` container – remember, within a handler, `this` refers to the element upon which the event was trigged, in this case, the element with the name attribute. The `closest()` method quickly locates the parent container upon which we find the suffix value.

We don't want to adjust element names that we've already adjusted once, so we use a regular expression test to see if the name already has the suffix attached, and if so, simply return from the handler.

If the name hasn't been adjusted, we use the `attr()` method to both fetch the original name and set the adjusted name back onto the element.

It's worth reflecting at this point how implementing this as a custom event, and using `live()`, creates very loose coupling in our page code. This frees us from having to worry about calling the adjustment code explicitly, or establishing the custom handlers explicitly at the various points in the code when they need to be applied.

This not only keeps the code cleaner, it increases the flexibility of the code.

Load this page into your browser and test the action of the Add Filter button. Note how every time that you click on the Add Filter button, a new filter is added to the page. If you inspect the DOM with a JavaScript debugger (Firebug in Firefox is great for this), you will see that the `name` of each `<select>` element has been suffixed with a per-filter suffix as expected.

But our job isn't over yet. The dropdowns don't yet specify which field is to be filtered. When a selection is made by the user, we need to populate the filter container with the appropriate qualifiers for the filter type for that field.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

4.3.5 Adding the qualifying controls

Whenever a selection is made from a filter dropdown, we need to populate the filter with the qualifying controls that are appropriate for that filter. We've made it easy for ourselves by creating markup templates that we just need to copy once we determine which one is appropriate. But there are also a few other housekeeping tasks that we need to do whenever the value of the dropdown is changed.

Let's take a look at what needs to db done when establishing the change handler for the dropdown:

```
$('select.filterChooser').live('change',function(){                    #1
  var filterType = $(':selected',this).attr('data-filter-type');
  var filterItem = $(this).closest('.filterItem');
  $('.qualifier',filterItem).remove();                                 #2
  $('div.template.'+filterType)                                        #3
      .children().clone().addClass('qualifier')
      .appendTo(filterItem)
      .trigger('adjustName');
  $('option[value=""]',this).remove();                                 #4
});
```

#1 Establishes the change handler
#2 Removes any old controls
#3 Replicates the appropriate template
#4 Removes the obsolete option

Cueballs in code and text

Once again we've taken advantage of jQuery's `live()` method to establish a handler up-front that will automatically be established at the appropriate points without further action on our part. This time, we've pro-actively established a change handler for any filter dropdown that comes into being (#1).

When the change handler fires, we first collect a few pieces of information: the filter type recorded in the custom `data-filter-type` attribute, and the parent filter container.

Once we've got those values in hand we need to remove any filter qualifier controls that might already be in the container (#2). After all, the user can change the value of the selected field many times and we don't want to just keep adding more and more controls as we go along! We'll add the `qualifier` class to all the appropriate elements as they are created (in the next statement), so it's easy to select and remove them.

Once we're sure we have a clean slate, we replicate the template for the correct set of qualifiers (#3) by using the value we obtained from the `data-filter-type` attribute. The `qualifier` class name is added to each created element for easy selection (as we saw in the previous statement). Also note how once again we trigger the "adjustName" custom event to automatically trigger the hander that will adjust the `name` attributes of the newly created controls.

Finally, we want to remove the "choose a filter" `<option>` element from the filter dropdown (#4), as once the user has selected a specific field, it doesn't make any sense to choose that entry again. We *could* just ignore the change event that triggers when the user selects this option, but the best way to prevent a user from doing something that doesn't make sense is to prevent them from being able to do it at all!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Once again refer to the example page in your browser. Try adding multiple filters, and change their selections. Note how the qualifiers always match the field selection. And if you can view the DOM in a debugger, observe how the `name` attributes are augmented.

Now, for those remove buttons.

4.3.6 Removing unwanted filters and other tasks

We've given the user the ability to change the field that any filter will be applied to, but we've also given them a remove button (labeled 'X') that they can use to remove a filter completely.

By this time, you should already have realized that this task will be almost trivial with the tools at our disposal. When the button is clicked, all we need to do is find the closest parent filter container, and blow it away!

```
$('button.filterRemover').live('click',function(){
  $(this).closest('div.filterItem').remove();
});
```

And yes, it turns out to be that simple.

Onto a few other matters…

You may recall that when the page is first loaded, an initial filter is displayed, even though we did not include it in the markup. Now we realize that we can easily realize this by simulating a click of the Add Filter button upon page load.

So in the ready handler we simply add:

```
$('#addFilterButton').click();
```

This causes the Add Button handler to be invoked just as if the user had clicked it.

And one final matter. While it's beyond the scope of this example to actually deal with submitting this form to the server, we thought we'd give you a tantalizing glimpse of what's coming up in future chapters.

Go ahead and click the Apply Filters button, which you may have noted is a "submit" button for the form. But rather than the page reloading as you might have expected, the results appear in the `<div>` element that we assigned the `id` of `resultsPane`.

That's because we subverted the submit action of the form with a handler of our own that cancels the form submission, and instead, makes an Ajax call with the form contents, loading the results into the `resultsPane` container. We'll see lots more about how easy jQuery makes Ajax in chapter 8, but you might be surprised (especially if you've already done some cross-browser Ajax programming) to see that we can make the Ajax call with just one line of code:

```
$('#filtersForm').submit(function(){
  $('#resultsPane').load('applyFilters',$('#filtersForm').serializeArray());
  return false;
});
```

If you pay attention to those displayed results, you might have noted that the results are the same no matter what filters are specified. Obviously, there's no real database powering this example, so its really just returning a hard-coded HTML page. But it's easy to imagine that the URL passed to jQuery's `load()` Ajax method could reference a dynamic PHP, Java Servlet, or Rails resource that would return actual results.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

That completes the page, at least as far as we've wanted to take it for the purposes of this chapter, but as we know…

4.3.7 There's always room for improvement

For our filter form to be considered production-quality, there's still lots of room for improvement.

Below we're going to list some additional functionality that this form either requires before being deemed "complete", or that would be just plain nice to have. Can you implement these additional features with the knowledge you've gained up to this point?

- Data validation is non-existent in our form. For example, the qualifying fields for the binder range should be numeric values, but we do nothing to prevent the user from entering invalid values. The same problem exists for the date fields.

  We could just punt and let the server-side code handle it – after all, it has to validate the data regardless. But that makes for a less-than-pleasant user experience, and as we've already pointed out, the best way to deal with errors is to prevent them from happening in the first place.

  Because the solution involves inspecting the Event instance – something that wasn't included in the example up to this point – we're going to give you the code to disallow the entry of any character but decimal digits into the numeric fields. The operation of the code should be evident to you with the knowledge you've gained in this chapter, but if not, now would be a good time to go back and review the key points.

```
$('input.numeric').live('keypress',function(event){
  if (event.which < 48 || event.which > 57) return false;
});
```

- As mentioned, the date fields aren't validated. How would you go about ensuring that only valid dates (by whatever format you choose) are entered? It can't be done on a character-by-character basis as we did with the numeric fields.

  Later in the book, we'll see how the jQuery UI plug-in solves this problem for us in a handy fashion, but for now put your event-handling knowledge to the test!

- When qualifying fields are added to a filter, the user must click into one of the fields to give it focus. Not all that friendly! Add code to the example to give focus to the new controls as they added.

- The use of a global variable to hold the filter count violates our sensibilities, and limits us to one instance of the "widget" per page. Replace it with the use of the `data()` method applied to an appropriate element keeping in mind that we may want to use this multiple times on a page.

- Our form allows the user to specify more than one filter, but we haven't defined how these filters are applied on the server. Do they form a disjunction (in which any one filter must match), or a conjunction (in which all filters must match)? Usually, if unspecified, a disjunction is assumed. But

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

how would you change the form to allow the user to specify which?

- What other improvements, either to the robustness of the code, or the usability of the interface would you make? How does jQuery help?

If you come up with ideas that you're proud of, be sure to visit the Manning web page for this book at http://www.manning.com/bibeault, which contains a link to the discussion forum. You're encouraged to post your solutions for all to see and discuss!

4.4 Summary

Building upon the jQuery knowledge that we've gained so far, this chapter introduced us to the world of event handling.

We learned that there are vexing challenges to implementing event handling in web pages, but such handling is essential for creating pages in DOM-scripted Applications. Not insignificant among those challenges is the fact the there are three event models that each operate in different ways across the set of modern popularly used browsers.

The legacy Basic Event Model, also informally termed the DOM Level 0 Event Model, enjoys somewhat browser-independent operation to declare event listeners, but the implementation of the listener functions requires divergent browser-dependent code in order to deal with differences in the Event instance. This event model is probably the most familiar to page authors, and assigns event listeners to DOM elements by assigning references to the listener functions to properties of the elements; the `onclick` property, for example.

Although simple, this model limits us to only one listener for any event type on a particular DOM element.

We can avoid this deficiency by using the DOM Level 2 Event Model, a more advanced and standardized model in which an API binds handlers to their event types and DOM elements. Versatile though this model is, it enjoys support only by standards-compliant browsers such as Firefox, Safari, Camino, and Opera.

For Internet Explorer, even up to IE 8,an API-based proprietary event model that provides a subset of the functionality of the DOM Level 2 Model is available.

Coding all event handling in a series of `if` statements—one clause for the standard browsers and one for Internet Explorer—is a good way to drive ourselves to early dementia. Luckily jQuery comes to the rescue and saves us from that fate.

jQuery provides a general `bind()`method to establish event listeners of any type on any element, as well as event-specific convenience methods such as `change()` and `click()`. These methods operate in a browser-independent fashion and normalize the `Event` instance passed to the handlers with the standard properties and methods most commonly used in event listeners.

jQuery also provides the means to remove event handlers, cause them to be triggered under script control, and even defines some higher-level methods that make implementing common event-handling tasks as easy as possible.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

As if all that were not enough, jQuery provides the `live()` method to assign handlers pro-actively to elements that may not even exist yet, and allows us to specify custom methods to easily register handlers to be invoked when those custom events are published.

We explored a few examples of using events in our pages, and explored a comprehensive example that demonstrated many of the concepts that we've learn up to this point. In the next chapter, we'll look at how jQuery builds upon these capabilities to put animation and animated effects to work for us.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

5

Energizing pages with animations and effects

This chapter covers

- Showing and hiding elements without animation
- Showing and hiding elements using core animation effects
- Writing our own custom animations
- Controlling animation and function queuing

Browsers have come a long way since LiveScript – subsequently renamed to JavaScript -- was introduced to Netscape Navigator in 1995 to allow scripting of web pages.

In those early days, the capabilities afforded to page authors was severely limited; not only by the minimal APIs, but by the sluggishness of scripting engines and low-powered systems.

The idea of using these limited abilities for animation and effects was laughable, and for years the only animation afforded was in the guise of animated GIF images (which were usually used poorly, making pages more annoying than usable).

My, how times have changed. Today's browser scripting engines are lightning fast, running on hardware that was unimaginable 10 years ago, and offer a rich variety of capabilities to us as page authors.

But even though the capabilities exist in low-level operations, JavaScript has no easy-to-use animation engine, so we're on our own. Except of course, that jQuery comes to our rescue, providing a trivially simple interface for doing all sorts of neat effects.

But before we dive into adding whiz-bang effects to our pages, we need to contemplate the question: *should we*? Like a Hollywood blockbuster that's all special effects and no plot, a page that overuses effects

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

can elicit a very different, and negative, reaction than what we intended. Be mindful that effects should be used to *enhance* the usability of a page, not hinder it by just "showing off".

With that caution in mind, let's see what jQuery has to offer.

5.1 Showing and hiding elements

Perhaps the most common type of dynamic effect we'll want to perform on an element, or any group of elements, is the simple act of showing or hiding them. We'll get to more fancy animations (like fading an element in or out) in a bit, but sometimes we want to keep it simple and pop elements into existence or make them instantly vanish!

The methods for showing and hiding elements are pretty much what we'd expect: `show()` to show the elements in a wrapped set and `hide()` to hide them. We're going to delay presenting their formal syntax for reasons that will become clear in a bit; for now, let's concentrate on using these methods with no parameters.

As simple as these methods may seem, we should keep a few things in mind. First, jQuery hides elements by changing the `display` value of the `style` property to `none`. If an element in the wrapped set is already hidden, it will remain hidden but still be returned for chaining. For example, suppose we have the following HTML fragment:

```
<div style="display:none;">This will start hidden</div>
<div>This will start shown</div>
```

If we apply `$("div").hide().addClass("fun")`, we'll end up with the following:

```
<div style="display:none;" class="fun">This will start hidden</div>
<div style="display:none;" class="fun">This will start shown</div>
```

Note that even though the first element was already hidden, it remains part of the matched set and takes part in the remainders of the method chain.

Second, jQuery shows objects by changing the display property from `none` to one of either `block` or `inline`. Which of these values is chosen is based either upon a whether a previously specified explicit value was set for the element or not. If the value was explicit, it is remembered and reverted. Otherwise it is based upon the default state of the `display` property for the target element type. For example, `<div>` elements will have their `display` property set to `block`, while a `<span>` element's `display` property will be set to `inline`.

Let's see about putting these methods to good use.

5.1.1 Implementing a collapsible "module"

You are no doubt familiar with sites, some that aggregate data from other sites, that present you with various pieces of information in configurable "modules" on some sort of "dashboard" page. The *iGoogle* site is a good example, as shown in figure 5.1.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

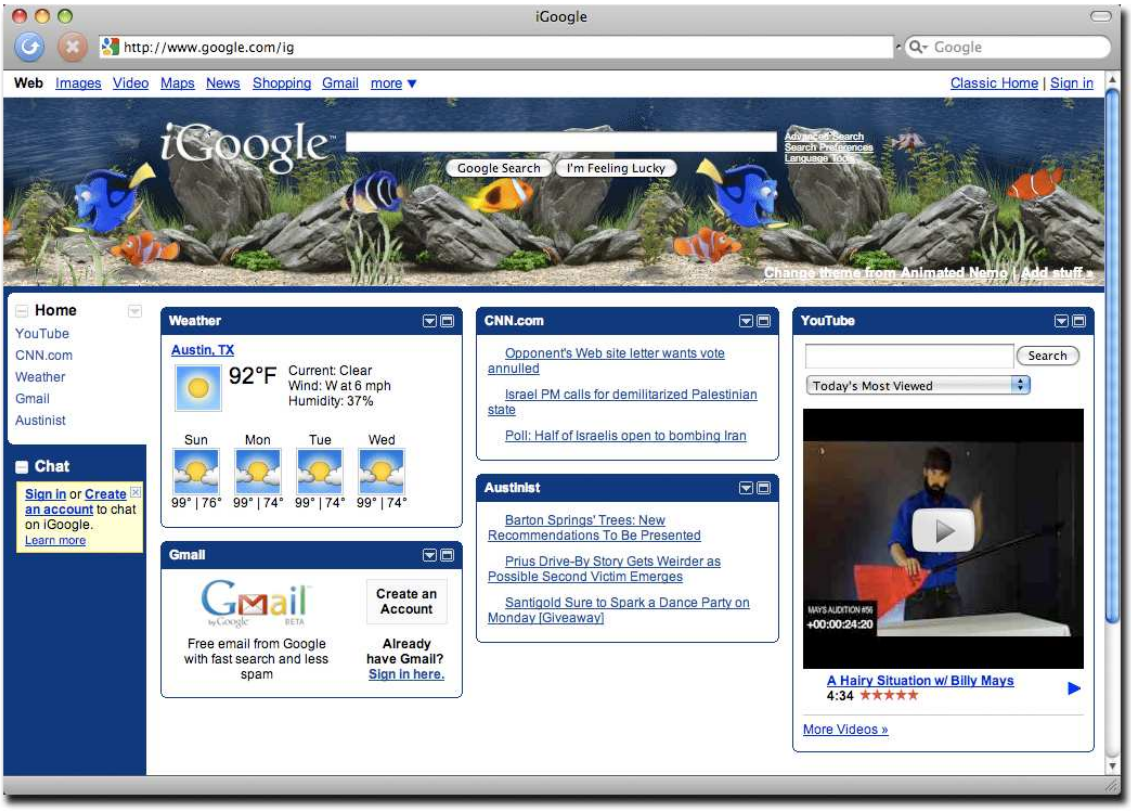


Figure 5.1 iGoogle is an example of a site that presents aggregated information in a series of "dashboard modules"

This site lets us configure much about how the page is presented, to include moving the modules around, expanding them to full-page, specifying configuration information, and even removing them completely.

But one thing it does not let us do (at least at the time of this writing) is to "roll up" the module into its caption bar so that it takes up less room, without having to remove it from the page.

Let's define our own "dashboard modules" and one-up Google by allowing users to roll up a module into its caption bar.

First, let's take a look at what we want the module to look like in its normal and rolled-up states, shown in figures 5.2a and 5.2b respectively.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

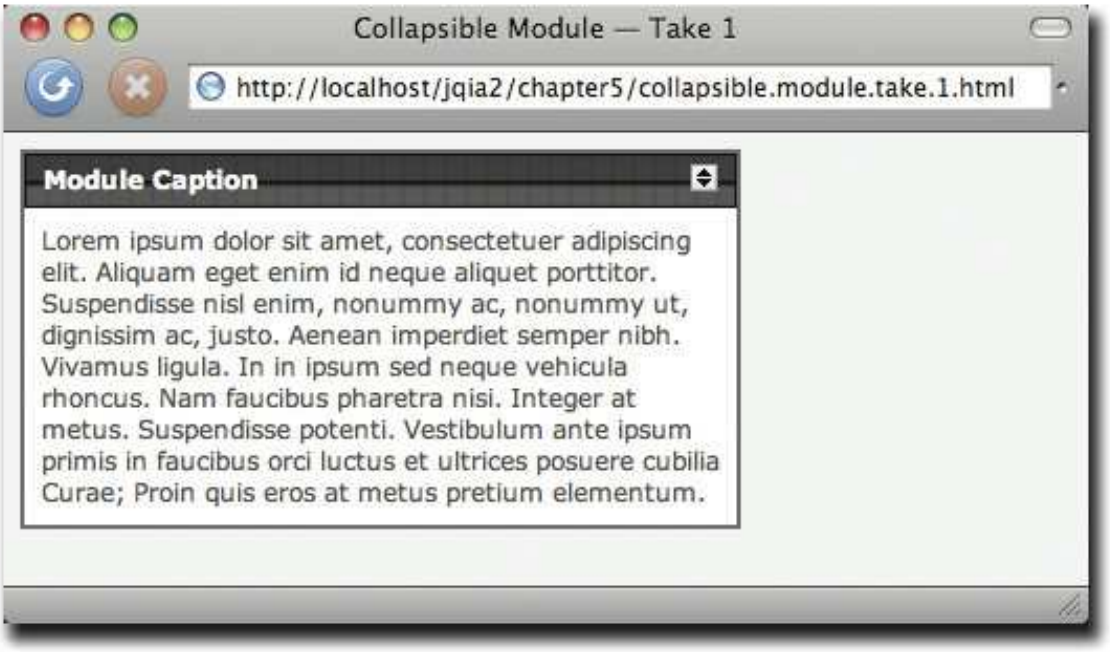


Figure 5.2a We'll create our own dashboard modules which consist of two parts: a banner with a caption and 'roll up" button, and a body in which data can be displayed

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

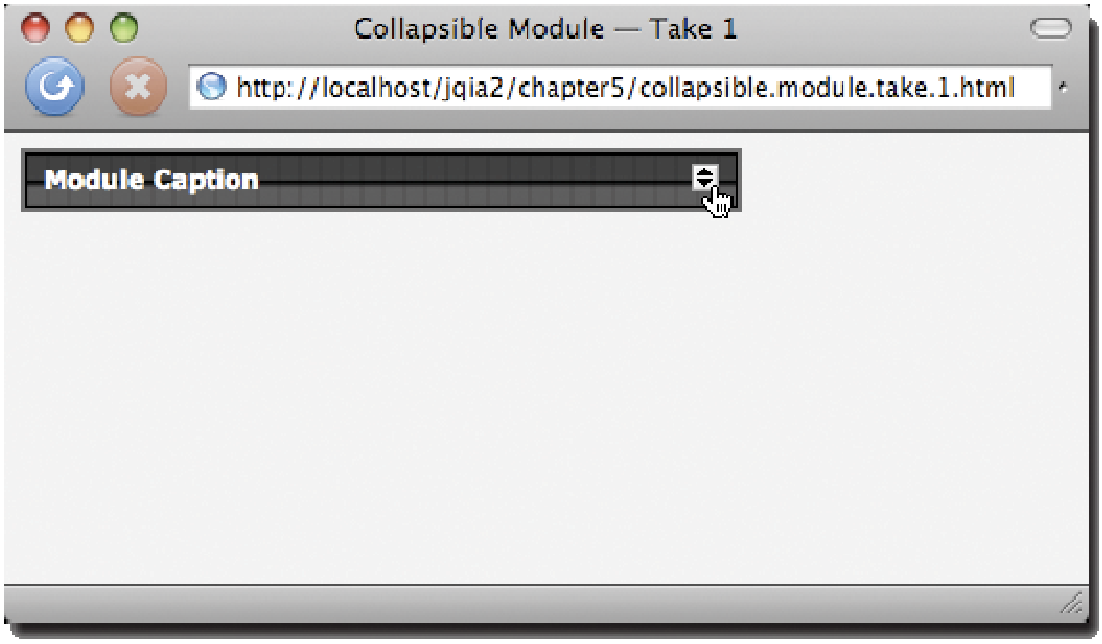


Figure 5.2b When the roll up button is clicked, the module body disappears as if it had been "rolled up" into the banner

In figure 5.2a, we see that we've created a module with two major sections: a banner, and a body. The body contains the data of the module, in this case random *Lorem ipsum* text. The more interesting banner contains a caption for the module, and small button that we will instrument to invoke the roll up (and roll down) functionality.

Once clicked, the body of the module will disappear, as if it had been "rolled up" into the banner. A subsequent click will "roll down" the body, restoring its original appearance.

The HTML markup we've used to create the structure of our module is fairly straightforward. We've applied numerous class names to the elements both for identification as well as for CSS styling.

```
<div class="module">
  <div class="banner">
    <span>Module Caption</span>
    <img src="rollup.gif" alt="rollup" title="rolls up this module"/>
  </div>
  <div class="body">
    Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
    Aliquam eget enim id neque aliquet porttitor. Suspendisse
    nisl enim, nonummy ac, nonummy ut, dignissim ac, justo.
    Aenean imperdiet semper nibh. Vivamus ligula. In in ipsum
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
      sed neque vehicula rhoncus. Nam faucibus pharetra nisi.
      Integer at metus. Suspendisse potenti. Vestibulum ante
      ipsum primis in faucibus orci luctus et ultrices posuere
      cubilia Curae; Proin quis eros at metus pretium elementum.
    </div>
  </div>
```

The entire construct is enclosed in a `<div>` element tagged with the `module` class, and the banner and body constructs are each represented as `<div>` children with the classes `banner` and `body` respectively.

In order to give this module the roll-up behavior, we'll instrument the image in the banner with a click handler that does all the magic. And with the `hide()` and `show()` methods up our sleeves, giving the module this behavior is considerably easier than pulling a quarter out from behind someone's ear.

Let's examine the code placed into the ready handler to take care of the roll-up behavior:

```
$('div.banner img').click(function(){                                    #1
  var body = $(this).closest('div.module').find('div.body');             #2
  if (body.is(':hidden')) {
    body.show();                                                         #3
  }
  else {
    body.hide();                                                         #4
  }
});
```

#1 Instruments the button
#2 Finds the related body
#3 Shows the body
#4 Hides the body

Cueballs in code and text

As would be expected, this code first establishes a click handler on the image in the banner (#1).

Within the click handler we first locate the body associated with the module. We need to find the specific instance of the module body because, remember, we may have many modules on our "dashboard" page, so we can't just select all elements that have the `body` class. We quickly locate the correct body element by finding the closest `module` container, and using it as the jQuery context, find the `body` within it using the jQuery expression:

```
$(this).closest('div.module').find('div.body') (#2)
```

(If how this expression finds the correct element isn't clear to you, now would be a good time to review the information in chapter 2 regarding finding and selecting elements.)

Once the body is located, it becomes a simple matter of determining if the body is hidden or shown (the jQuery `is()` method comes in mighty handy here), and either showing or hiding it as appropriate using the `show()` (#3) and `hide()` (#4) methods.

The full code for this page can be found in file `chapter5/collapsible.module.take.1.html` and is shown in listing 5.1. (If you surmise that the "take 1" part of this file name indicates that we'll be revisiting this example, you're right!)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Listing 5.1 The first implementation of our collapsible module

```
<!DOCTYPE html>
<html>
  <head>
    <title>Collapsible Module &mdash; Take 1</title>
    <link rel="stylesheet" type="text/css" href="../styles/core.css">
    <link rel="stylesheet" type="text/css" href="module.css">
    <script type="text/javascript" src="../scripts/jquery-1.3.2.min.js"></script>
    <script type="text/javascript">
      $(function() {

        $('div.banner img').click(function(){
          var body = $(this).closest('div.module').find('div.body');
          if (body.is(':hidden')) {
            body.show();
          }
          else {
            body.hide();
          }
        });

      });
    </script>
  </head>

  <body class="plain">

    <div class="module">
      <div class="banner">
        <span>Module Caption</span>
        <img src="rollup.gif" alt="rollup" title="rolls up this module"/>
      </div>
      <div class="body">
        Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
        Aliquam eget enim id neque aliquet porttitor. Suspendisse
        nisl enim, nonummy ac, nonummy ut, dignissim ac, justo.
        Aenean imperdiet semper nibh. Vivamus ligula. In in ipsum
        sed neque vehicula rhoncus. Nam faucibus pharetra nisi.
        Integer at metus. Suspendisse potenti. Vestibulum ante
        ipsum primis in faucibus orci luctus et ultrices posuere
        cubilia Curae; Proin quis eros at metus pretium elementum.
      </div>
    </div>

  </body>

</html>
```

That wasn't difficult at all, was it? But as it turns out, it can be even easier!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

5.1.2 Toggling the display state of elements

Toggling the display state of elements between revealed or hidden—as we did for the collapsible module example—is such a common occurrence that jQuery defines a method named `toggle()` that makes it even easier.

Let's apply this method to the collapsible module and see how it helps to simplify the code of listing 5.1. Listing 5.2 shows only the ready handler for the refactored page (no other changes are necessary) with the new changes highlighted in bold. The complete page code can be found in file `chapter5/collapsible.module.take.2.html`.

Listing 5.2 The collapsible module code, simplified with `toggle()`

```
$(function() {

  $('div.banner img').click(function(){
    $(this).closest('div.module').find('div.body').toggle();
  });

});
```

Note that we no longer need the conditional statement to determine whether to hide or show the module body; `toggle()` takes care of swapping the displayed state on our behalf. This allowed us to simplify the code quite a bit and the need to store the body reference in a variable simply vanishes.

Instantaneously making elements appear and disappear is handy, but sometimes we want the transition to be less abrupt. Let's see what's available for that.

5.2 Animating the display state of elements

Human cognitive ability being what it is, making items pop into and out of existence instantaneously can be jarring to us. If we blink at the wrong moment, we could miss the transition, leaving us to wonder, "What just happened?"

Gradual transitions of a *short* duration help us know what's changing and *how* we got from one state to the other – and that's where the jQuery core effects come in. There are three sets of effect types:

- Show and hide (there's a bit more to these methods than we let on in section 5.1)
- Fade in and fade out
- Slide down and slide up

Let's look more closely at each of these effect sets.

5.2.1 Showing and hiding elements gradually

The `show()`, `hide()`, and `toggle()` methods are a tad more complex than we led you to believe in the previous section. When called with no parameters, these methods effect a simple manipulation of the display state of the wrapped elements, causing them to instantaneously be revealed or hidden from the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

display. But when passed parameters, these effects can be animated so that the changes in display status of the affected elements take place over a period of time.

With that, we're now ready to look at the full syntaxes of these methods.

Method syntax: hide

`hide(speed,callback)`

Causes the elements in the wrapped set to become hidden. If called with no parameters, the operation takes place instantaneously by setting the `display` style property value of the elements to `none`. If a `speed` parameter is provided, the elements are hidden over a period of time by adjusting their size and opacity downward to zero, at which time their `display` style property value is set to `none` to remove them from the display.
An optional callback can be specified that's invoked when the animation is complete.

Parameters

`speed`	(Number|String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: "slow", "normal", or "fast". If omitted, no animation takes place, and the elements are immediately removed from the display.
`callback`	(Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated. The callback is fired for each element that undergoes animation.

Returns

The wrapped set.

Method syntax: show

`show(speed,callback)`

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Causes any hidden elements in the wrapped set to be revealed. If called with no parameters, the operation takes place instantaneously by setting the `display` style property value of the elements to an appropriate setting (one of `block` or `inline`)
If a `speed` parameter is provided, the elements are revealed over a specified duration by adjusting their size and opacity upward.
An optional callback can be specified that's invoked when the animation is complete.

Parameters

`speed`	(Number|String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: "slow", "normal", or "fast". If omitted, no animation takes place and the elements are immediately revealed in the display.
`callback`	(Function) An optional function invoked when the animation is complete. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated. The callback is fired for each element that undergoes animation.

Returns

The wrapped set.

Method syntax: toggle

`toggle(speed,callback)`

Performs `show()` on any hidden wrapped elements and `hide()` on any non-hidden wrapped elements. See the syntax description of these methods for their respective semantics.

Parameters

`speed`	(Number|String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: "slow", "normal", or "fast". If omitted, no animation takes place.
`callback`	(Function) An optional function invoked when the animation is complete. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated. The callback is fired for each element that undergoes animation.

Returns

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

The wrapped set.

Method syntax: toggle

`toggle(condition)`

Shows or hides the matched elements based upon the evaluation of the passed condition. If `true`, the elements are shown; otherwise, they are hidden.

Parameters

`condition`	(Boolean) Determines if elements are shown (if `true`), or hidden (if `false`)..

Returns

The wrapped set.

Let's do a third take on the collapsible module, animating the opening and closing of the sections.

Given the previous information, you'd think that the only change we need to make to the code of *take 2* of this collapsible module implementation would be to change the call to the `toggle()` method to

```
toggle('slow')
```

And you'd be right.

But not so fast! Because that was just *too* easy, let's take the opportunity to also add some whizz-bang to the module.

Let's say that, for whatever reason, we decide that we want the module's banner to display a different background when it's in its rolled up state. We could just make the change before firing off the animation, but it'd be much more suave to wait until the animation is finished.

We can't just make the call right after the animation method call as animations *do not block.* The statements following the animated method call execute immediately, probably even before the animation has a chance to commence.

Rather, that's where the callback that we can register with the `toggle()` method comes in.

The approach that we will take is that, after the animation is complete, we'll simply add a class name to the module to indicate that it is rolled up, and remove the class name when it is not rolled up. CSS rules will take care of the rest.

Your initial thoughts might have led you to think of using the `css()` to add a `background-image` style property directly to the banner, but why use a sledgehammer when we have a scalpel?

The "normal" CSS rule for the module banner (found in file `chapter5/module.css`) is as follows:

```
div.module div.banner {
  background: black url('module.banner.backg.png');
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
  …
}
```

We've also added another rule:

```
div.module.rolledup div.banner {
  background: black url('module.banner.backg.rolledup.png');
}
```

This second rule causes the background image of the banner to change whenever its parent module possesses the `rolledup` class. So in order to effect the change, all we have to do is to add or remove the `rolledup` class to the module at the appropriate points.

Listing 5.3 shows the new ready-handler code that makes that happen.

Listing 5.3 Animated version of the collapsible module, now with magically changing banner!

```
$(function() {

  $('div.banner img').click(function(){
    $(this).closest('div.module').find('div.body')
      .toggle('slow',function(){
        $(this).closest('div.module')
          .toggleClass('rolledup',$(this).is(':hidden'));
      });
  });

});
```

The page with these changes can be found in file `chapter5/collapsible.list.take.3.html`.

Knowing how much people like us love to tinker, we've set up a handy tool that we'll use to further examine the operation of these and the remaining effects methods.

INTRODUCING THE JQUERY EFFECTS LAB PAGE

Back in chapter 2, we introduced the concept of Lab Pages to help us experiment with using jQuery selectors. For this chapter, we've set up a lab page for exploring the operation of the jQuery effects in file `chapter5/lab.effects.html`.

Loading this page into your browser results in the display of figure 5.3.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

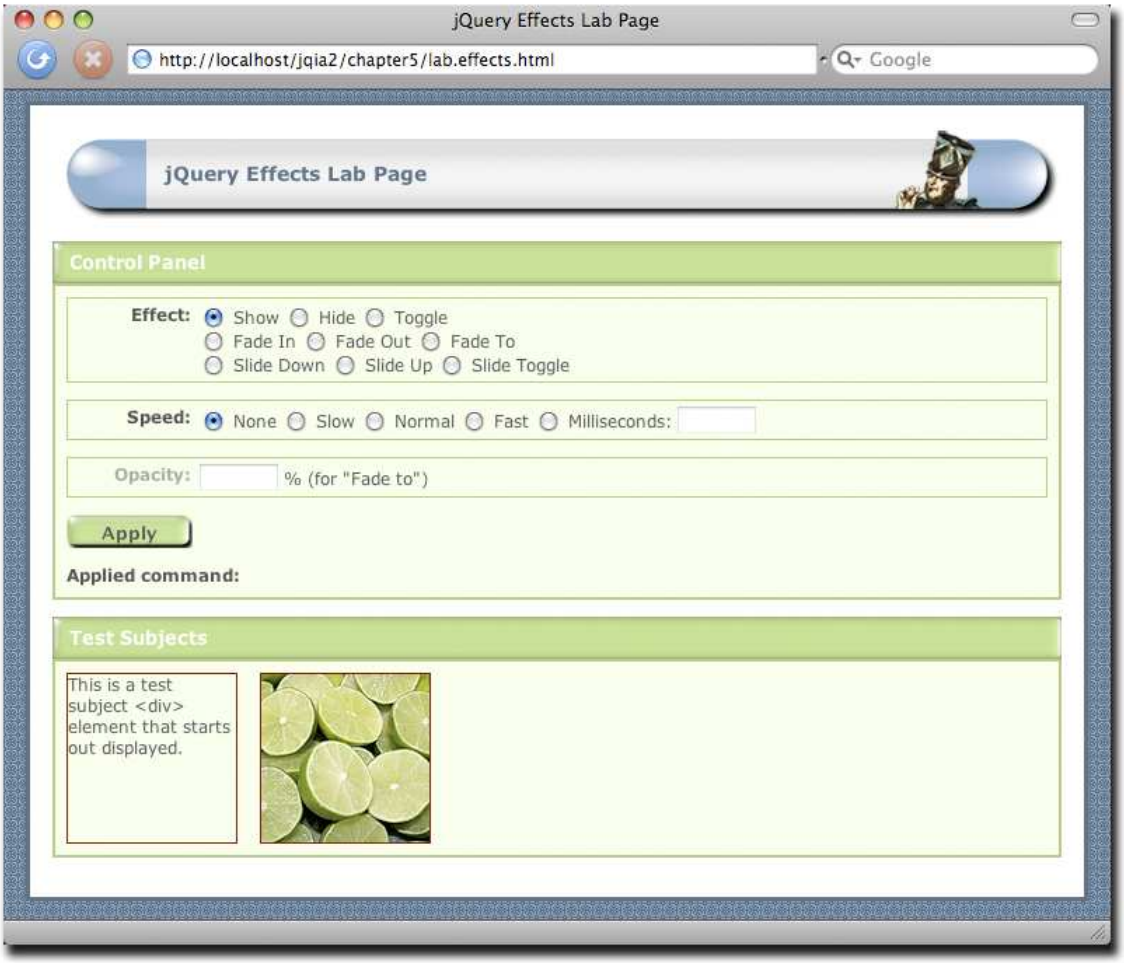


Figure 5.3 The initial state of the jQuery Effects Lab Page, which will help us examine the operation of the jQuery effects methods

This lab page consists of two main panels: a control panel in which we'll specify which effect will be applied, and one that contains four test subject elements upon which the effects will act.

"Are they daft?" you might be thinking. "There are only two test subjects."
No, your authors haven't lost it yet. There *are* four elements, but two of them (another `<div>` with text and another image) are initially hidden.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Let's use this page to demonstrate the operations of the methods we've discussed to this point. Display the page in your browser, and follow along with the ensuing exercises:

- *Exercise 1*—With the controls left as is after the initial page load, click the Apply button. This will execute a `show()` method with no parameters. The expression that was applied is displayed below the Apply button for your information. Note how the two initially hidden test subject elements appear instantly. If you're wondering why the belt image on the far right appears a bit faded, its opacity has been purposefully set to 50 percent.

- *Exercise 2*—Select the Hide radio button, and click Apply to execute a parameter-less `hide()` method. All of the test subjects immediately vanish. Take special notice that the pane in which they resided has tightened up. This indicates that the elements have been completely removed from the display rather than merely made invisible.

NOTE

When we say that an element has been *removed from the display* (here, and in the remainder of our discussion about effects), we mean that the element is no longer being taken into account by the browser's layout manager by setting its CSS `display` style property to `none`. It does *not* mean that the element has been removed from the DOM tree; none of the effects will ever cause an element to be removed from the DOM.

- *Exercise 3*—Next, select the Toggle radio button, and click Apply. Click Apply again. And again. You'll note that each subsequent execution of `toggle()` flips the presence of the test subjects.

- *Exercise 4*—Reload the page to reset everything to the initial conditions (in Firefox and other Gecko browsers, set focus to the address bar and hit the Enter key – simply hitting the reload button won't reset the form elements). Select Toggle, and click Apply. Note how the two initially visible subjects vanish and the two that were hidden appear. This demonstrates that the `toggle()` method applies individually to each wrapped element, revealing the ones that are hidden and hiding those that aren't.

- *Exercise 5*—In this exercise, let's move into the realm of animation. Refresh the page, leave Show selected, and for the Speed setting, select Slow. Click Apply, and carefully watch the test subjects. The two hidden elements, rather than popping into existence, gradually grow from their upper left corners. If you want to really see what's going on, refresh the page, select Milliseconds for the speed and enter `10000` for the speed value. This will extend the duration of the effect to 10 (excruciating) seconds and give you plenty of time to observe the behavior of the effect.

- *Exercise 6*—Choosing various combinations of Show, Hide, and Toggle, as well as various speeds, experiment with these effects until you feel you have a good handle on how they operate.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Armed with the jQuery Effects Lab Page, and the knowledge of how this first set of effects operates, let's take a look at the next set of effects.

5.2.2 Fading elements into and out of existence

If you watched the operation of the `show()` and `hide()` effects carefully, you noted that they scaled the size of the elements (either up or down as appropriate) *and* adjusted the opacity of the elements as they grew or shrank. The next set of effects, `fadeIn()` and `fadeOut()`, only affect the opacity of the elements.

Other than the lack of scaling, these methods work in a fashion similar to the animated forms of `show()` and `hide()` respectively. The syntaxes of these methods are as follow:

Method syntax: fadeIn

`fadeIn(speed,callback)`

Causes any matched elements that are hidden to be shown by gradually changing their opacity to their natural value. This value is either the opacity originally applied to the element, or 100%. The duration of the change in opacity is determined by the `speed` parameter. Any elements that aren't hidden aren't affected.

Parameters

`speed`	(Number|String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: "slow", "normal", or "fast". If omitted, the default is "normal".
`callback`	(Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated. This callback is fired individually for each animated element.

Returns

The wrapped set.

Method syntax: fadeOut

`fadeOut(speed,callback)`

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Causes any matched elements that aren't hidden to be removed from the display by gradually changing their opacity to 0% and then removing the element from the display. The duration of the change in opacity is determined by the `speed` parameter. Any elements that are already hidden aren't affected.

Parameters

`speed`	(Number|String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: "slow", "normal", or "fast". If omitted, the default is "normal".
`callback`	(Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated. This callback is fired individually for each animated element.

Returns

The wrapped set.

Let's have some more fun with the jQuery Effects Lab Page. Display the lab, and run through a set of exercises similar to those we followed in the previous section using the Fade In and Fade Out selections (don't worry about Fade To for now, we'll attend to that soon enough).

It's important to note that when the opacity of an element is adjusted, the jQuery `hide()`, `show()`, `fadeIn()`, and `fadeOut()` effects remember the original opacity of an element and honor its value. In the lab page, we purposefully set the initial opacity of the belt image at the far right to 50 percent before hiding it. Throughout all the opacity changes that take place when applying the jQuery effects, this original value is never stomped on.

Run though additional exercises in the lab until you're convinced that this is so and are comfortable with the operation of the fade effects.

Another effect that jQuery provides is via the `fadeTo()` method. This effect adjusts the opacity of the elements like the previously examined fade effects, but never removes the elements from the display. Before we start playing with `fadeTo()` in the lab, here's its syntax.

Method syntax: fadeTo

`fadeTo(speed,opacity,callback)`

Gradually adjusts the opacity of the wrapped elements from their current setting to the new setting specified by `opacity`.

Parameters

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`speed`	(Number|String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: "slow", "normal", or "fast". If omitted, the default is "normal".
`opacity`	(Number) The target opacity to which the elements will be adjusted specified as a value from 0.0 to 1.0.
`callback`	(Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated. This callback is fired individually for each animated element.

Returns

The wrapped set.

Unlike the other effects that adjust opacity while hiding or revealing elements, `fadeTo()` doesn't remember the original opacity of an element. This makes sense because the whole purpose of this effect is to explicitly change the opacity to a specific value.

Bring up the lab page, and cause all elements to be revealed (you should know how by now). Then work through the following exercises:

- *Exercise 1*—Select Fade To and a speed value slow enough for you to observe the behavior; 4000 milliseconds is a good choice. Now set the Fade to Opacity field (which expects a percentage value between 0 and 100, converted to 0.0 through 1.0 when passed to the method) to `10`, and click Apply. The test subjects will fade to 10 percent opacity over the course of four seconds.
- *Exercise 2*—Set the opacity to `100`, and click Apply. All elements, including the initially semi-transparent belt image, are adjusted to full opaqueness.
- *Exercise 3*—Set the opacity to `0`, and click Apply. All elements fade away to invisibility, but note that once they've vanished, the enclosing module does *not* tighten up. Unlike the `fadeOut()` effect, `fadeTo()` never removes the elements from the display, even when they're fully invisible.

Continue experimenting with the Fade To effect until you've mastered its workings. Then we'll be ready to move on to the next set of effects.

5.2.3 Sliding elements up and down

Another set of effects that hide or show elements—`slideDown()` and `slideUp()`—also works in a similar manner to the `hide()` and `show()` effects, except that the elements appear to slide down from their tops when being revealed and to slide up into their tops when being hidden.

As with `hide()` and `show()`, the slide effects have a related method that will toggle the elements between hidden and revealed: `slideToggle()`. The by-now-familiar syntaxes for these methods follow.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Method syntax: slideDown

`slideDown(speed,callback)`

Causes any matched elements that are hidden to be shown by gradually increasing their vertical size. Any elements that aren't hidden aren't affected.

Parameters

`speed`	(Number|String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: "slow", "normal", or "fast". If omitted, the default is "normal".
`callback`	(Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated. This callback is fired individually for each animated element.

Returns

The wrapped set.

Method syntax: slideUp

`slideUp(speed,callback)`

Causes any matched elements that aren't hidden to be removed from the display by gradually decreasing their vertical size.

Parameters

`speed`	(Number|String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: "slow", "normal", or "fast". If omitted, the default is "normal".
`callback`	(Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated. This callback is fired individually for each animated element.

Returns

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

The wrapped set.

Method syntax: slideToggle

`slideToggle(speed,callback)`

Performs `slideDown()` on any hidden wrapped elements and `slideUp()` on any non-hidden wrapped elements. See the syntax description of these methods for their respective semantics.

Parameters

`speed`	(Number|String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: "slow", "normal", or "fast". If omitted, the default is "normal".
`callback`	(Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated. This callback is fired individually for each animated element.

Returns

The wrapped set.

Except for the manner in which the elements are revealed and hidden, these effects act similarly to the other show/hide effects. Convince yourself of this by displaying the jQuery Effects Lab Page and running through sets of exercises similar to those we applied to the other effects.

5.2.4 Stopping animations

We may have a reason now and again to stop an animation once it has started. This could be because a user event dictates that something else should occur or because we want to start a completely new animation. The `stop()` command will achieve this for us:

Command syntax: stop

`stop(clearQueue,gotoEnd)`

Halts all animations that may be currently in progress for the elements in the matched set.

Parameters

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`clearQueue`	(Boolean) If specified and set to `true`, not only stops the current animation, but any other animations waiting in the animation queue. (Animation queue? We'll get to that shortly...)
`gotoEnd`	(Boolean) If specified and set to `true`, advances the current animation to its logical end (as opposed to merely stopping it).

Returns

The wrapped set

Note that any changes that have already taken place for any animated elements will remain in effect. If we want to restore the elements to their original state, it's our responsibility to put the CSS values back to their starting values using the `css()` method or similar commands.

By the way, there's also a global flag that we can use to completely disable all animations. Setting the flag `jquery.fx.off` to `true`, will cause all effects to take place immediately without animation. We'll cover this flag more formally in chapter 6 with the other jQuery flags.

Now that we've seen the effects built into core jQuery, let's investigate writing our own!

5.3 Creating custom animations

The number of core effects supplied with jQuery is purposefully kept small, in order to keep jQuery's core footprint to a minimum, with the expectation that plugins are available to add more animations at the page author's discretion. It's also a surprisingly simple matter to write our own animations.

jQuery publishes the `animate()` wrapper method that allows us to apply our own custom animated effects to the elements of the wrapped set. Let's take a look at its syntax.

Method syntax: animate

`animate(properties,duration,easing,callback)`
`animate(properties,options)`

Applies an animation, as specified by the `properties` and `easing` parameters, to all members of the wrapped set. An optional callback function can be specified that's invoked when the animation is complete. An alternate format specifies a set of `options` in addition to the `properties`.

Parameters

`properties`	(Object) An object hash that specifies the end values that supported CSS styles should reach at the end of the animation. The animation takes place by adjusting the values of the style properties from the current value for an element to the value specified in this object hash.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`duration`	(Number|String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: "slow", "normal", or "fast". If omitted, no animation takes place.
`easing`	(String) The optional name of a function to perform easing of the animation. Easing functions must be registered by name and are often provided by plugins. Core jQuery supplies two easing functions registered as "linear" and "swing".
`callback`	(Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated. This callback is fired individually for each animated element.
`options`	(Object) Specifies the animation parameter values using an object hash. The supported properties are as follow: `duration`—See previous description of `duration` parameter. `easing`—See previous description of `easing` parameter. `complete`—Function invoked when the animation completes. `queue`—If `false`, the animation isn't queued and begins running immediately. `step`—A callback function called at each step of the animation. This callback is passed: the step index and an internal effects object (that doesn't contain much of interest to us as page authors). The function context is set to the element under animation.

Returns

The wrapped set.

We can create custom animations by supplying a set of CSS style properties and target values that those properties will converge towards as the animation progresses. Animations start with an element's original style value and proceed by adjusting that style value in the direction of the target value. The intermediate values that the style achieves during the effect (automatically handled by the animation engine) are determined by the duration of the animation and the easing function.

The specified target values can be absolute values, or we can specify relative values from the starting point. To specify relative values, prefix the value with `+=` or `-=` to indicate relative target values in the positive or negative direction, respectively.

The term *easing* is used to describe the manner in which the processing and pace of the frames of the animation are handled. By using some fancy math on the duration of the animation and current time position, some interesting variations to the effects are possible. The subject of writing easing functions is a complex, niche topic that's usually only of interest to the most hard-core of plugin authors; we're not going to delve into the subject of custom easing functions in this book. If you'd like to sample more easing functions than "linear" (which provides a linear progression) or "swing" (which speeds up slightly near the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

end of an animation), check out the Easing Plugin at `http://gsgd.co.uk/sandbox/jquery.easing.php`.

By default, animations are added to a queue for execution (much more on that coming up); applying multiple animations to an object will cause them to run serially. If you'd like to run animations in parallel, set the `queue` option to `false`.

The list of CSS style properties that can be animated is limited to those that accept numeric values for which there is a logical progression from a start value to a target value. This numeric restriction is completely understandable—how would we envision the logical progress from a source value to an end value for a non-numeric property such as `image-background`? For values that represent dimensions, jQuery assumes the default unit of pixels, but we can also specify em units or percentages by including the *em* or *%* suffixes.

Frequently animated style properties include `top`, `left`, `width`, `height`, and `opacity`. But if it makes sense for the effect we want to achieve, numeric style properties such as font size, margin, padding and border dimensions can also be animated.

NOTE

If you want to animate a CSS value that specifies a color, you may be interested in the official jQuery Color Animation Plugin at `http://jquery.com/plugins/project/color`.

In addition to specific values for the target properties, we can also specify one of the strings "hide", "show", or "toggle"; jQuery will compute the end value as appropriate to the specification of the string. Using "hide" for the `opacity` property, for example, will result in the opacity of an element being reduced to 0. Using any of these special strings has the added effect of automatically revealing or removing the element from the display (like the `hide()` and `show()` methods), and it should be noted that "toggle" remembers the initial state so that it can be restored on a subsequent "toggle".

Did you notice that when we introduced the core animations that there was no toggling method for the fade effects? That's easily solved using `animate()` and "toggle" to create a simple custom animation as follows:

```
$('.animateMe').animate({opacity:'toggle'},'slow');
```

Taking this to the next logical step – creating a wrapper function – could be coded as follows:

```
$.fn.fadeToggle = function(speed){
  return this.animate({opacity:'toggle'},speed);
};
```

Now, let's try our hand at writing a few more custom animations.

5.3.1 A custom scale animation

Consider a simple *scale* animation in which we want to adjust the size of the elements to twice their original dimensions. We write such an animation as shown in listing 5.

Listing 5.4 A custom Scale Animation

```
$('.animateMe').each(function(){                                    #1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
  $(this).animate({
      width: $(this).width() * 2,                                    #2
      height: $(this).height() * 2
    },
    2000                                                             #3
  );
});
```

#1 Iterates over each matched element
#2 Specifies individual target values
#3 Sets duration

Cueballs in code and text

To implement this animation, we iterate over all the elements in the wrapped set via `each()` to apply the animation individually to each matched element (#1). This is important because the property values that we need to specify for each element are based upon the individual dimensions for that element (#2). If we always knew that we'd be animating a single element (such as if we were using an `id` selector) or applying the exact same set of values to each element, we could dispense with `each()` and animate the wrapped set directly.

Within the iterator function, the `animate()` method is applied to the element (identified via `this`) with style property values for `width` and `height` set to double the element's original dimensions. The result is that over the course of two seconds (as specified by the `duration` parameter of `2000` (#3)), the wrapped elements (or element) will grow from their original size to twice that original size.

Now let's try something a bit more extravagant.

5.3.2 A custom drop animation

Let's say that we want to conspicuously animate the removal of an element from the display, perhaps because it's vitally important to convey to users that the item being removed is *gone* and that they should make no mistake about it. The animation we'll use to accomplish this will make it appear as if the element *drops* off the page, disappearing from the display as it does so.

If we think about it for a moment, we can figure out that, by adjusting the `top` position of the element, we can make it move down the page to simulate the drop; adjusting the `opacity` will make it seem to vanish as it does so. And finally, when all that's done, we want to remove the element from the display (similar to the animated `hide()`).

We accomplish this drop effect with the code of listing 5.5.

Listing 5.5 A custom Drop Animation

```
$('.animateMe').each(function(){
  $(this)
    .css('position','relative')                                      #1
    .animate(
      {
        opacity: 0,
        top: $(window).height() - $(this).height() -                 #2
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
            $(this).position().top                          #2
        },
        'slow',
        function(){ $(this).hide(); }                       #3
      );
});
```

#1 Dislodges element from static layout
#2 Computes drop distance
#3 Removes element from display

Cueballs in code and text

There's a bit more going on here than in our previous custom effect. We once again iterate over the element set, this time adjusting the position *and* opacity of the elements. But to adjust the `top` value of an element relative to its original position, we first need to change its CSS `position` style property value to `relative` (#1).

Then for the animation, we specify a target `opacity` of `0` and a computed `top` value. We don't want to move an element so far down the page that it moves below the window's bottom; this could cause scroll bars to be displayed where none may have been before, possibly distracting users. We don't want to draw their attention away from the animation—grabbing their attention is why we're animating in the first place! So we use the height and vertical position of the element, as well as the height of the window, to compute how far down the page the element should drop (#2).

When the animation is complete, we want to remove the element from the display, so we specify a callback routine (#3) that applies the non-animated `hide()` method to the element (which is available to the function as its function context).

NOTE

We did a little more work than we needed to in this animation, just so we could demonstrate doing something that needs to wait until the animation is complete in the callback function. If we were to specify the value of the opacity property as "hide" rather than `0`, the removal of the element(s) at the end of the animation would be automatic, and we could dispense with the callback.

Now let's try one more type of "make it go away" effect for good measure.

5.3.3 A custom puff animation

Rather than dropping elements off the page, let's say that we want an effect that makes it appear as if the element dissipates away into thin air like a puff of smoke. To animate such an effect, we combine a scale effect with an opacity effect, growing the element while fading it away. One issue we need to deal with for this effect is that this dissipation would not fool the eye if we let the element grow in place with its upper-left corner anchored. We want the *center* of the element to stay in the same place as it grows, so in addition to its size we also need to adjust the *position* of the element as part of the animation.

The code for our puff effect is shown in listing 5.6.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Listing 5.6 A custom Puff Animation

```
$('.animateMe').each(function(){
  var position = $(this).position();
  $(this)
    .css({position: 'absolute',                                    #1
          top: position.top,
          left: position.left})
    .animate(                                                      #2
      {
        opacity: 'hide',
        width: $(this).width() * 5,
        height: $(this).height() * 5,
        top: position.top - ($(this).height() * 5 / 2),
        left: position.left - ($(this).width() * 5 / 2)
      },
      'normal');
});
```

#1 Dislodges element from static flow
#2 Adjusts element size, position and opacity

Cueballs in code and text

In this animation, we decrease the opacity to 0 while growing the element to five times its original size while adjusting its position by half that new size, resulting in the location of the center of the element remaining constant (#2). We don't want the elements surrounding the animated element to be pushed out while the target element is growing, so we take it out of the layout flow completely by changing its position to `absolute` and explicitly setting its position coordinates (#1).

Because we specified "hide" for the `opacity` value, the elements are automatically hidden (removed from the display) once the animation is complete.

Each of these three custom effects can be observed by loading the page at `chapter5/custom.effects.html` whose display is shown in figure 5.4.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Figure 5.4 The Custom Effects we developed: Scale, Drop, and Puff, can be observed in action using the buttons provided on this example page

We purposefully kept the browser window to a minimum for the screen shot, but you'll want to make the window bigger when running this page to properly observe the behavior of the effects. And although we'd love to show you how these effects behave, screenshots have obvious limitations. Nevertheless, figure 5.5 shows the puff effect in progress.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

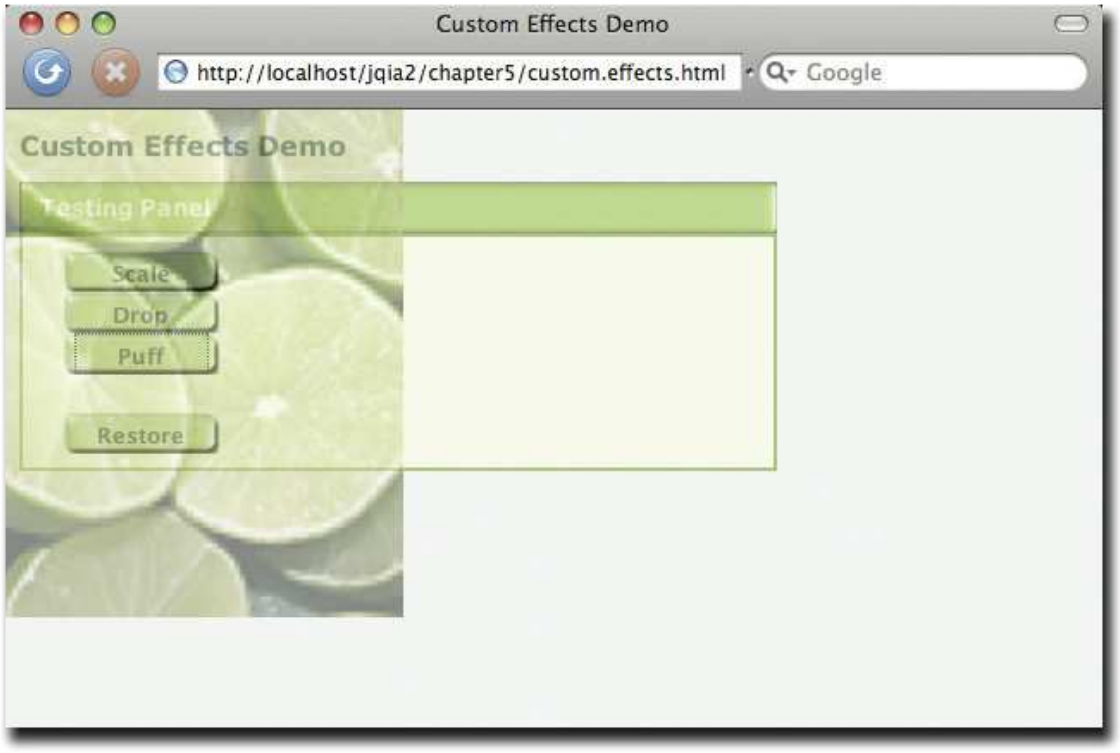


Figure 5.5 The Puff Effect expands and moves the image while simultaneously reducing its opacity.

We'll leave it to you to try out the various effects on this page and observe their behavior.

Up until this point, all of the examples we've examined have used a single animation method. Let's discuss how things work when we use more than one.

5.4 Animations and Queuing

We've seen how multiple properties of elements can be animated using a single animation method, but we haven't really examined how animations behave when we call simultaneous animations methods.

In this section we'll examine just how animations behave in concert with each other.

5.4.1 Simultaneous animations

What would you expect to happen if we were to execute the following code?

```
$('#testSubject').animate({left:'+=256'},'slow');
$('#testSubject').animate({top:'+=256'},'slow');
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

We know that the `animate()` method, as well as all the other animation methods, do not block while their animations are running on the page. And we can prove that to ourselves by experimenting with this code block:

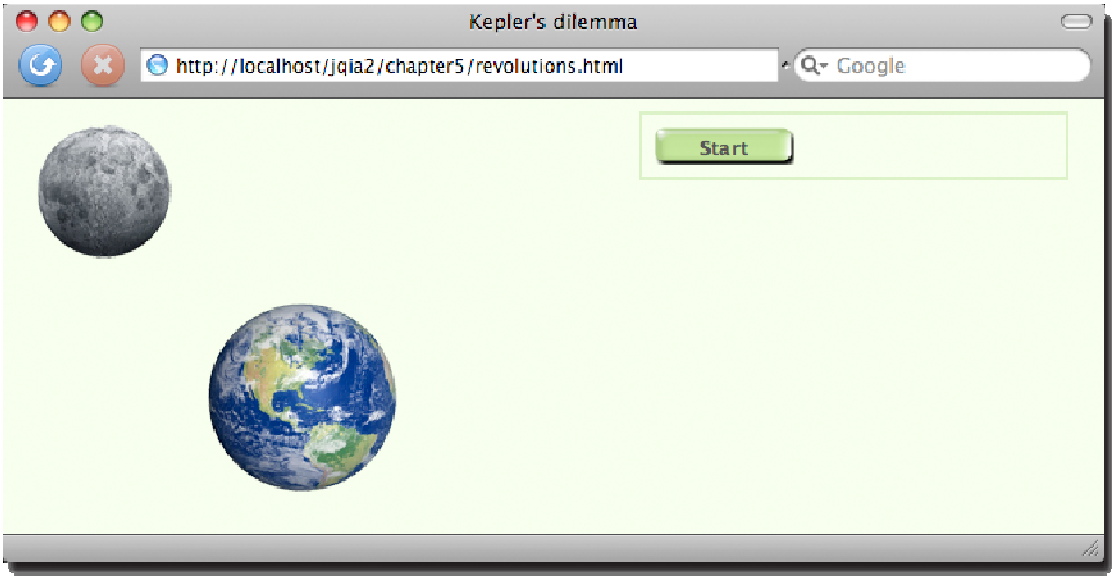
```
say(1);
$('#testSubject').animate({left:'+=256'},'slow');
say(2);
```

Recall that we introduced the `say()` function in chapter 4 as a way to spit messages onto an on-page "console" in order to avoid alerts (which would most definitely screw up our observation of the experiment).

If we were to execute this code, we'd see that the messages "1" and "2" are emitted immediately, one after the other, without waiting for the animation to complete.

So, considering the code with two animation method calls, what would we expect to happen? Because the second method isn't blocked by the first, it stands to reason that both animations fire off simultaneously (or within a few milliseconds of each other), and that the effect on the test subject would be the combination of the two effects. In this case, because one effect is adjusting the left property, and the other the top property, we might expect that the result would be a meandering diagonal movement of the test subject.

Let's put that to the test. In file `chapter5/revolutions.html`, we've put together an experiment that sets up two images (one of which is to be animated), a button to start the experiment, and a "console" in which the `say()` function will write its output. Figure 5.6 shows its initial state:



©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Figure 5.6 Initial state of the page where we will observe the behavior of multiple, simultaneous animations

The Start button is instrumented as shown in listing 5.7:

Listing 5.7 Multiple simultaneous animations instrumentation

```
$('#startButton').click(function(){
  say(1);
  $("img[alt='moon']").animate({left:'+=256'},2500);
  say(2);
  $("img[alt='moon']").animate({top:'+=256'},2500);
  say(3);
  $("img[alt='moon']").animate({left:'-=256'},2500);
  say(4);
  $("img[alt='moon']").animate({top:'-=256'},2500);
  say(5);
});
```

In the click handler for the button, we fire off four animations, one after the other, interspersed with calls to `say()` that show us when the animation calls were fired off.

Bring up the page, and click the Start button.

As expected, the console messages, "1" through "5", immediately appear on the "console" as shown in figure 5.7, each firing off a few milliseconds after the previous one.

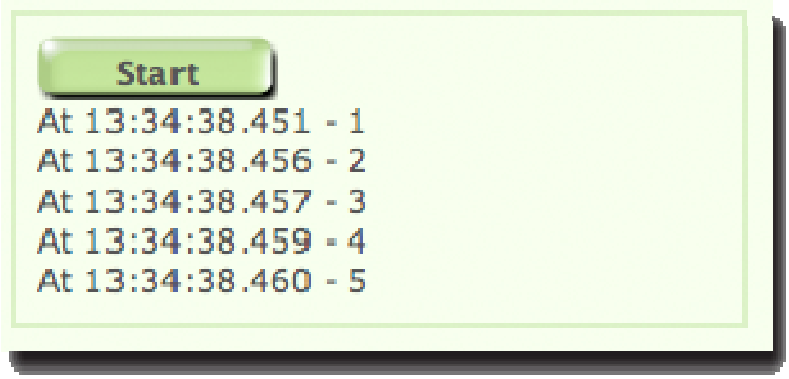


Figure 5.7 The console messages appear in rapid succession, proving that the animation methods aren't blocking until completion

But what of the animations? If we examine the code of listing 5.7, we see that we have two animations changing the `top` property, and two animations changing the `left` property. In fact, the animations for

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

each property are doing the exact opposite of each other. So what should we expect? Might they just cancel each other out and the moon (our test subject) remain completely still?

But no. Upon clicking Start and observing the behavior of the animations, we see that each animation happens in a serial fashion, one after the other, such that the Moon makes a complete and orderly revolution around the Earth (albeit in a very unnatural square orbit that would have made Kepler's head explode).

What's going on? We have proven via the console messages that the animations aren't blocking, yet they execute serially just as if they were (at least with respect to each other).

What's happening is that, internally, jQuery is queuing up the animations and executing them serially on our behalf.

Refresh the Earth and Moon page to clear the console, and click the Start button three times in succession. (Pause between clicks just long enough to avoid double-clicks.)

You'll note how 15 messages get immediately sent to the console, indicating that our click handler has executed three times, and then sit back as the Moon makes three orbits around the Earth.

Each of the 12 animations is queued up by jQuery and executed in order. jQuery maintains a queue on each animated element named "fx" just for this purpose. (The significance of the queue having a name will become clear in the next section.)

The queuing of animations in this manner means that we can have our cake and eat it too! We can affect multiple properties simultaneously by using a single `animate()` method that specifies all the animated properties, and we can serially execute any animations we want by simple calling them in order.

Doesn't that butter-cream icing taste good?

What's even better is that JQuery makes it possible for us to create our own execution queues for whatever purposes we want. Let's learn about that.

5.4.2 Queuing functions for execution

Queuing up animations for serial execution is an obvious use for function queues. But is there a real benefit? After all, the animation methods allow for completion callbacks, why not just fire off the next animation in the callback of the previous animation?

Let's review the code fragment of listing 5.7 (minus the `say()` invocations for clarity):

```
$("img[alt='moon']").animate({left:'+=256'},2500);
$("img[alt='moon']").animate({top:'+=256'},2500);
$("img[alt='moon']").animate({left:'-=256'},2500);
$("img[alt='moon']").animate({top:'-=256'},2500);
```

Compare that to the equivalent code, that would be necessary without function queuing, using the completion callbacks:

```
$('#startButton').click(function(){
  $("img[alt='moon']").animate({left:'+=256'},2500,function(){
    $("img[alt='moon']").animate({top:'+=256'},2500,function(){
      $("img[alt='moon']").animate({left:'-=256'},2500,function(){
        $("img[alt='moon']").animate({top:'-=256'},2500);
      });
    });
  });
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
});
```

It's not that the callback variant of the code is that much more complicated, but it'd be hard to argue that the original code isn't a lot easier to read (and to write in the first place). And if the bodies of the callback functions were to get substantially more complicated… well, it's easy to see how being able to queue up the animations makes the code a lot less complex.

So what if we wanted to do the same thing with our own functions? Well, jQuery isn't greedy about its queues; we can create our own to queue up any functions that we'd like to have executed in serial order.

Queues can be created on any element, and distinct queues can be created by using unique names for them (except for "fx" which is reserved for the effects queue). The method to add a function instance to a queue is, unsurprisingly, `queue()`, and it has three variants, as described in the following syntax:

Method syntax: queue

`queue(name)`
`queue(name,function)`
`queue(name,queue)`

The first form returns any queue of the passed name already established on the first element in the matched set as an array of functions.
The second form adds the passed function to the end of the named queue for all elements in the matched set. If such a named queue does not exist on an element, it is created.
The last form replaces any existing queue on the matched elements with the passed queue.

Parameters

`name`	(String) The name of the queue to be fetched, added to, or replaced. If omitted, the default effects queue of "fx" is assumed.
`function`	(Function) The function to be added to the end of the queue. When invoked, the function context (`this`) will be set to the DOM element upon which the queue has been established.
`queue`	(Array) An array of functions that *replaces* the existing functions in the named queue.

Returns

An array of functions for the first form, the wrapped set for the remaining forms.

The `queue()` method is most often used to add functions to the end of the named queue, but can also be used to fetch any existing functions in a queue, or to replace the list of functions in a queue. Note that the array form, in which an array of functions is passed to `queue()`, cannot be used to add multiple functions to the *end* of a queue as any existing queued functions are removed.

OK, so now we can queue functions up for execution. That's not all that useful unless we can somehow cause the execution of the functions to actually occur. Enter the `dequeue()` method:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Method syntax: dequeue

dequeue(name)

Removes the foremost function in the named queue for each element in the matched set and executes it for each element.

Parameters

name	(String) The name of the queue from which the foremost function is to be removed and executed. If omitted, the default effects queue of "fx" is assumed.

Returns

The wrapped set

When `dequeue()` is invoked, the foremost function in the named queue for each element in the wrapped set is executed with the function context for the invocation (`this`) being set to the element.

Let's consider the code of listing 5.8:

Listing 5.8 Queuing and dequeuing functions on multiple elements

```
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="../styles/core.css">
    <script type="text/javascript" src="../scripts/jquery-1.3.2.min.js"></script>
    <script type="text/javascript" src="console.js"></script>
    <script type="text/javascript">
      $(function() {

        $('img').queue('chain',                                          #1
         function(){ say('First: ' + $(this).attr('alt')); });           #1
        $('img').queue('chain',                                          #1
          function(){ say('Second: ' + $(this).attr('alt')); });         #1
        $('img').queue('chain',                                          #1
          function(){ say('Third: ' + $(this).attr('alt')); });          #1
        $('img').queue('chain',                                          #1
          function(){ say('Fourth: ' + $(this).attr('alt')); });         #1

        $('button').click(function(){                                    #2
          $('img').dequeue('chain');
        });

      });
    </script>
  </head>

  <body>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
    <div>
      <img src="earth.png" alt="Earth"/>
      <img src="moon.png" alt="Moon"/>
    </div>

    <button type="button" class="green90x24">Dequeue</button>

    <div id="console"></div>

  </body>
</html>
```

#1 Establishes four queued functions
#2 Dequeues one function on each click

Cueballs in code and text

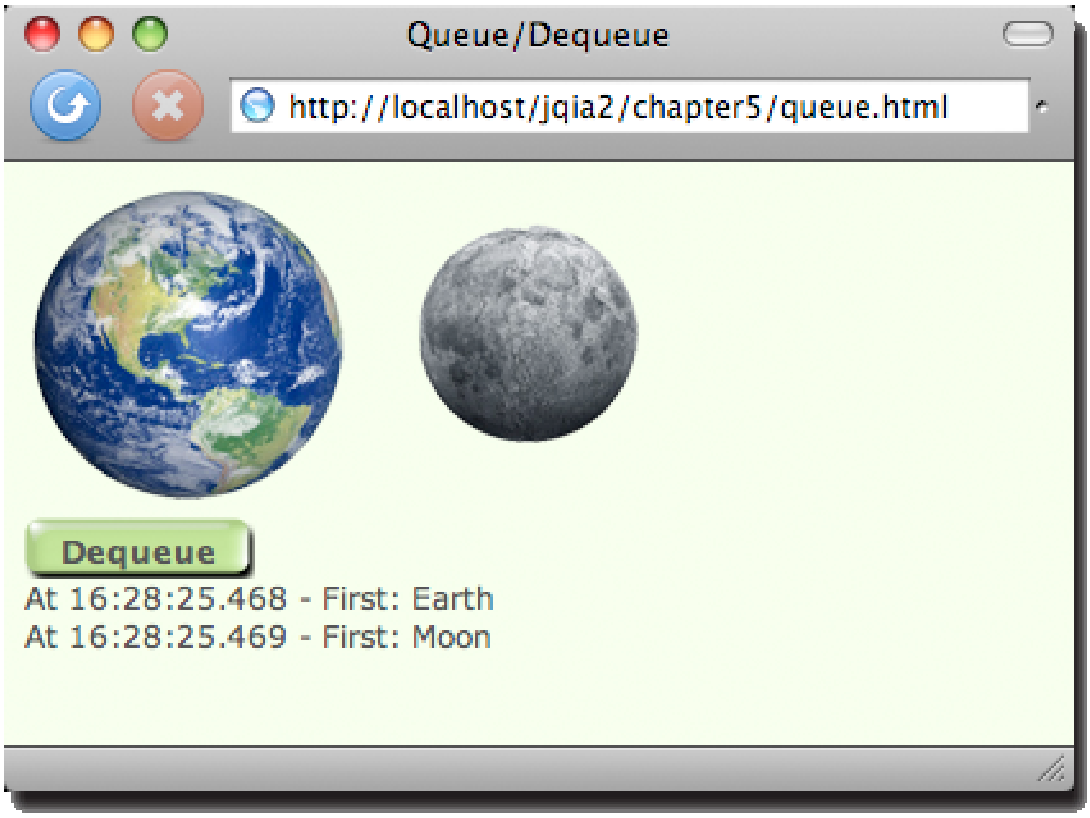
In this example (found in file `chapter5/queue.html`), we have two images upon which we establish queues named "chain". In each queue, we place four functions (#1) that identify themselves in order and emit the `alt` attribute of whatever DOM element is serving as the function context. This way, we can tell which function is being executed, and from which element's queue.

Upon clicking the Dequeue button, the button's click handler (#2) causes a single execution of the `dequeue()` method.

Go ahead and click the button once, and observe the messages in the console as shown in figure 5.8:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>



We can see that the first function we added to the "chain" queue for the images has been fired twice: once for the Earth image, and once for the Moon image.

Clicking the button subsequent times removes the next function from the queues and executes them until the queues have been emptied; after which, calling `dequeue()` has no effect.

In this example, the dequeuing of the functions was under manual control – we needed to click the button four times (resulting in four calls to `dequeue()`) to get all four functions executed. Frequently we may want to trigger the execution of the entire set of queued functions. For such times, a commonly used idiom is to call the `dequeue()` method *within* the queued function in order to trigger the execution of (in other words, "chain to") the next queued function.

Consider the following changes to our above example:

```
$('img').queue('chain',
 function(){
   say('First: ' + $(this).attr('alt'));
   $(this).dequeue('chain');
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
 });
$('img').queue('chain',
  function(){
    say('Second: ' + $(this).attr('alt'));
    $(this).dequeue('chain');
  });
$('img').queue('chain',
  function(){
    say('Third: ' + $(this).attr('alt'));
    $(this).dequeue('chain');
  });
$('img').queue('chain',
  function(){
    say('Fourth: ' + $(this).attr('alt'));
    $(this).dequeue('chain');
  });
```

We've made just such a change in the example of file `chapter5/queue.2.html`. Bring up that page in your browser and click the Dequeue button. Note how the single click now triggers the execution of the entire chain of queued functions.

There's one more thing to discuss regarding queuing functions before moving along…

5.4.3 Inserting functions into the effects queue

We previously mentioned that internally, jQuery uses a queue named "fx" to queue up the functions necessary to implement the animations. What if we'd like to add our own functions to this queue in order to trigger actions interspersed within a queued series of effects? Now that we know about the queuing methods, we can!

Think back to our previous example of Listing 5.7, where we used four animations to make the Moon revolve around the Earth. Imagine that we wanted to turn the background of the Moon image black after the second animation (the one that moves it downward). If we just added a call to the `css()` method between the second and third animations as follows:

```
$("img[alt='moon']").animate({left:'+=256'},2500);
$("img[alt='moon']").animate({top:'+=256'},2500);
$("img[alt='moon']").css({'background-color':'black'});
$("img[alt='moon']").animate({left:'-=256'},2500);
$("img[alt='moon']").animate({top:'-=256'},2500);
```

we'd be very disappointed as this would cause the background to change immediately, perhaps even before the first animation has a chance to start.

Rather, consider the following code:

```
$("img[alt='moon']").animate({left:'+=256'},2500);
$("img[alt='moon']").animate({top:'+=256'},2500);
$("img[alt='moon']").queue('fx',
  function(){
    $(this).css({'background-color':'black'});
    $(this).dequeue('fx');
   }
);
$("img[alt='moon']").animate({left:'-=256'},2500);
$("img[alt='moon']").animate({top:'-=256'},2500);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Here, we wrap the `css()` method in a function that we place onto the 'fx' queue using the `queue()` method. (We could have omitted the queue name, as "fx" is the default, but we made it explicit here for clarity.) This puts it into place on the effects queue where it will be called as part of the function chain that executes as the animations progress, between the second and third animations.

But note! After we call the `css()` method, we call the `dequeue()` method on the "fx" queue.

This is absolute necessary to keep the animation queue chugging along. Failure to call `dequeue()` at this point will cause the animations to grind to a halt, as nothing is causing the next function in the chain to execute. The unexecuted animations will just sit there on the effects queues until either something causes a dequeue and the functions commence, or the page unloads and everything just gets discarded.

If you'd like to see this process in action, load the page of file `chapter5/revolutions.2.html` into your browser and click the button.

Queuing functions comes in handy whenever we want to execute function consecutively, but without the overhead, or complexity of nesting function in asynchronous callbacks; something that, as you might imagine, can come in handy when we throw Ajax into the equation.

But that's another chapter.

5.5 Summary

This chapter introduced us to the animated effects that jQuery makes available out-of-the-box, as well as to the `animate()`method that allows us to create our own custom animations.

The `show()` and `hide()` methods, when used without parameters, reveal and conceal elements from the display immediately, without any animation. We can perform animated versions of the hiding and showing of elements with these methods by passing parameters that control the speed of the animation, as well as providing an optional callback that's invoked when the animation completes. The `toggle()` method toggles the displayed state of an element between hidden and shown.

Another set of wrapper methods, `fadeOut()` and `fadeIn()`, also hides and shows elements by adjusting the opacity of elements when removing or revealing them in the display. A third method, `fadeTo()`, animates a change in opacity for its wrapped elements without removing the elements from the display.

A final set of three built-in effects animates the removal or display of our wrapped elements by adjusting their vertical height: `slideUp()`, `slideDown()`, and `slideToggle()`.

For building our own custom animations, jQuery provides the `animate()` method. Using this method, we can animate any CSS style property that accepts a numeric value, most commonly the opacity, position and dimensions of the elements. We explored writing some custom animations that remove elements from the page in novel fashions.

We also learned how jQuery queues animations for serial execution, and how we can use the jQuery queuing methods to add our own functions to the effects queue or our own custom queues.

When we explored writing our own custom animations, we wrote the code for these custom effects as inline code within the on-page JavaScript. A much more common, and useful, method is to package custom animations as custom jQuery methods. We'll learn how to do that in chapter 7, and you're

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

encouraged to revisit these effects after you've read that chapter. Repackaging the custom effects we developed in this chapter, and any that you could think up on your own, would be an excellent follow-up exercise.

But before we write our own jQuery extensions, let's take a look at some high-level functions and flags that jQuery provides that will be very useful for general tasks as well as extension writing.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

6

Beyond the DOM with jQuery Utility Functions

This chapter covers:

- The jQuery browser support information
- Using other libraries with jQuery
- Array manipulation functions
- Extending and merging objects
- Dynamically loading new scripts
- And more…

Up to this point, we've spent a fair number of chapters examining the jQuery methods that operate upon a set of DOM elements wrapped by the `$()` function. But you may recall that way back in chapter 1, we also introduced the concept of *utility functions*—functions name-spaced by `jQuery`/`$` that don't operate on a wrapped set. These functions could be thought of as top-level functions except that they are defined on the `$` instance rather than `window`, keeping them *out* of the global namespace.

Generally, these functions either operate upon JavaScript objects *other* than DOM elements (that's the purview of the wrapper methods after all), or they perform some non-object-related operation (such as Ajax).

In addition to functions, jQuery also provides some useful flags that are defined within the jQuery/$ namespace.

You may wonder why we waited until this chapter to introduce these functions and flags. Well, we had two reasons:

1. We wanted to guide you into thinking in terms of using jQuery methods rather than resorting to lower-level operations that might feel more familiar but not be as efficient or as easy to code as using the jQuery wrapper.
2. Because the wrapper methods take care of much of what we want to do when manipulating DOM elements on the pages, these lower-level functions are frequently most useful when writing the methods themselves (as well as other extensions) rather than in page-level code. (We'll be tackling how to write our own plugins to jQuery in the next chapter.)

In this chapter we'll finally get around to formally introducing most of the $-level utility functions, as well as a handful of useful flags. We'll put off talking about the utility functions that deal with Ajax until chapter 8, which deals exclusively with jQuery's Ajax functionality.

We'll start out with those flags that we mentioned.

6.1 Using the jQuery flags

Some of the information jQuery makes available to us as page authors, and even plugin authors, is available not via methods or functions but as variables defined on $. Many of these *flags* are generally focused on helping us divine the capabilities of the current browser so that we can make decisions based on such information when necessary, but others help us control the behavior of jQuery at a page-global level.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

The jQuery flags intended for public use are as follows:

- `$.fx.off`
- `$.support`
- `$.browser`

Let's start by looking at how jQuery lets us disable animations.

6.1.1 Disabling animations

There may be times when we might want to conditionally disable animations in a page that we've used various animated effects within. We might do so because we've detected that the platform or device is unlikely to deal with them well, or perhaps for accessibility reasons.

In any case, we don't need to resort to writing two pages: one with and one without animations. When we detect we are in an animation-adverse environment, we can simple set the value of:

`$.fx.off`

to `true`.

This will *not* suppress any effects that we've used on the page, it will simply disable the animation of those effects. For example, the fade effects will simply show and hide the elements immediately, without the intervening animations.

Similarly, calls to the `animate()` method will simply set the CSS properties to the specified final values without animating them.

One possible use-case for this flag might be for certain mobile devices or browsers that don't correctly support animations. In that case, you might want to turn off animations so that the core functionality still works.

The `$.fx.off` flag is a read/write flag. The remaining pre-defined flags are meant to be read-only. Let's take a look at the flag that gives us information on the environment provided by the user agent (browser).

6.1.2 Detecting user agent support

Thankfully, almost blissfully, the jQuery methods that we've examined so far shield us from having to deal with browser differences, even in traditionally problematic areas like event handling. But when we're the ones writing these methods (or other extensions), we may need to account for the differences in the ways browsers operate so that the users of our extensions don't have to.

But before we dive into seeing how jQuery helps us in this regard, let's talk about the whole concept of browser detection.

WHY BROWSER DETECTION IS HEINOUS

OK, maybe the word *heinous* is too strong, but unless it's absolutely necessary, browser detection is a technique that should only be used when no other options are available.

Browser detection might seem, at first, like a logical way to deal with browser differences. After all, it's easy to say: "I know what the set of capabilities of browser X are, so testing for the browser makes perfect sense, right?" But browser detection is full of pitfalls and problems.

One of the major arguments against this technique is that the proliferation of browsers, as well as varying levels of support within versions of the same browser, makes this technique an unscalable approach to the problem.

You could be thinking, "Well, all I need to test for is Internet Explorer and Firefox." But why would you exclude the growing number of Safari users? What about Opera and Google's Chrome? Moreover, there are some niche, but not insignificant, browsers that share capability profiles with the more popular browsers. Camino, for example, uses the same technology as Firefox behind its Mac-friendly UI. And OmniWeb uses the same rendering engine as Safari.

There's no need to exclude support for these browsers, but it *is* a royal pain to have to test for them. And that's without even considering differences between versions—IE6, IE7 and IE8, for example.

Yet another reason is that if we test for a specific browser, and a future release fixes that bug, our code may actually stop working. jQuery's alternative approach to this issue (which we'll discuss in the very next section) actually gives browser vendors an incentive to fix the bugs that jQuery has worked around.

A final argument against browser detection (or *sniffing* as it's sometimes called) is that it's getting harder and harder to know who's who.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Browsers identify themselves by setting a request header known as the *user agent* string. Parsing this string isn't for the faint-hearted. In addition, many browsers now allow their users to spoof this string, so we can't even believe what it tells us after we *do* go though all the trouble of parsing it!

A JavaScript object named `navigator` gives us a partial glimpse into the user agent information, but even *it* has browser differences. We almost need to do browser detection in order to do browser detection!

Stop the madness!

Browser detection is:

- *Imprecise*—Accidentally blocking browsers that our code would actually work within
- *Unscalable*—Leading to enormous nested if and if-else statements to sort things out
- *Inaccurate*—Due to users spoofing false user agent information

Obviously, we'd like to avoid using it whenever possible.

But what can we do instead?

WHAT'S THE ALTERNATIVE TO BROWSER DETECTION?

If we think about it, we're not *really* interested in which browser anyone is using, are we? The only reason we're even thinking about browser detection is so that we can know which capabilities and features we can use. It's the *capabilities* and *features* of a browser that we're really after; using browser detection is just a ham-handed way of trying to determine what those features and capabilities are.

So why don't we just figure out what those features are rather than trying to infer them from the browser identification? The technique known broadly as *feature detection* allows code to branch based on whether certain objects, properties, or even methods exist.

Let's think back to our chapter on event handling as an example. We remember that there are two advanced event-handling models: the W3C standard DOM Level 2 Event Model and the proprietary Internet Explorer Model. Both models define methods on the DOM elements that allow listeners to be established, but each uses different method names. The standard model defines the method `addEventListener()`, whereas the IE model defines `attachEvent()`.

Using browser detection, and assuming that we've gone through the pain and aggravation of determining what browser is being used (maybe even correctly), we could write

```
...
complex code to set flags: isIE, isFirefox and isSafari
...
if (isIE) {
  element.attachEvent('onclick',someHandler);
}
else if (isFirefox || isSafari) {
  element.addEventListener('click',someHandler);
}
else {
  throw new Error('event handling not supported');
}
```

Aside from the fact that this example glosses over whatever necessarily complex code we are using to set the flags `isIE`, `isFirefox`, and `isSafari`, we can't be sure if these flags accurately represent the browser being used. Moreover, this code will throw an error if used in Opera, Chrome, Camino, OmniWeb, or a host of other lesser-known browsers that might perfectly support the standard model.

Consider the following variation of this code:

```
if (element.attachEvent) {
  element.attachEvent('onclick',someHandler);
}
else if (element.addEventListener) {
  element.addEventListener('click',someHandler);
}
else {
  throw new Error('event handling not supported');
}
```

This code doesn't perform a lot of complex, and ultimately unreliable, browser detection, and it automatically supports all browsers that support either of the two competing event models. Much better!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Feature detection is vastly superior to browser detection. It's more reliable, and it doesn't accidentally block browsers that support the capability we are testing for simply because we don't know about the features of that browser, or even of the browser itself. Did you account for Google Chrome in your most recent web application?

NOTE

Even feature detection is best avoided unless absolutely required. If we can come up with a cross-browser solution, it should be preferred over *any* type of branching.

But as superior to browser detection as feature detection may be, it can still be no walk in the park. Branching and detection of any type can still be tedious and painful in our pages, and some feature differences can be decidedly difficult to detect, requiring non-trivial or down-right complex checks. jQuery comes to our aid by performing those checks for us and supplying the results in a set of flags that detect the most common user agent features that we might care about.

THE JQUERY BROWSER CAPABILITY FLAGS

The user agent capability flags are exposed to us as properties of an object that jQuery provides as: `$.support`

Table 6.1 summarizes the flags that available in this object.

Please keep table inline with text even if it means breaking the table across pages

Table 6.1 The $.support user agent capability flags

Flag property	Description
`boxModel`	Set to `true` if the user agent renders according to the standards compliant box model. This flag is not set until document-ready. More information regarding the box model issue is available at `http://www.quirksmode.org/css/box.html` and at `http://www.w3.org/TR/REC-CSS2/box.html`.
`cssFloat`	Set to `true` if the standard `cssFloat` property of the elements's `style` property is used.
`hrefNormalized`	Set to `true` if obtaining the `href` element attributes returns the value exactly as specified.
`htmlSerialize`	Set to `true` if the browser evaluates style sheets references by `<link>` elements when injected into the DOM via `innerHTML`.
`leadingWhitespace`	Set to `true` if the browser honors leading whitespace when text is inserted via `innerHTML`.
`noCloneEvent`	Set to `true` if the browser does *not* copy event handlers when an element is cloned.
`objectAll`	Set to `true` is the `getElementByTagName()` method returns all descendants of the element when passed "*".
`opacity`	Set to `true` if the browser correctly interprets the standard `opacity` CSS property.
`scriptEval`	Set to `true` of the browser evaluates `<script>` blocks when they are injected via calls to the `appendChild()` or `createTextNode()` methods.
`style`	Set to `true` if the attribute to obtain the inline style properties of an element is `style`.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`tbody`	Set to `true` if a browser does not automatically insert `<tbody>` elements into tables lacking them when injected via `innerHTML`.

Table 6.2 shows the values for each of these flags for the various browser families.

Please keep table inline with text even if it means breaking the table across pages

Table 6.2 Browser results for the `$.support` flags

Flag property	Gecko (Firefox, Camino, etc)	Webkit (Safari, OmniWeb, Chrome, etc)	Opera	IE
`boxModel`	`true`	`true`	`true`	`false` in quirks mode, `true` in standards mode
`cssFloat`	`true`	`true`	`true`	`false`
`hrefNormalized`	`true`	`true`	`true`	`false`
`htmlSerialize`	`true`	`true`	`true`	`false`
`leadingWhitespace`	`true`	`true`	`true`	`false`
`noCloneEvent`	`true`	`true`	`true`	`false`
`objectAll`	`true`	`true`	`true`	`false`
`opacity`	`true`	`true`	`true`	`false`
`scriptEval`	`true`	`true`	`true`	`false`
`style`	`true`	`true`	`true`	`false`
`tbody`	`true`	`true`	`true`	`false`

As expected, it comes down to the differences between Internet Explorer and the standards-compliant browsers. But lest this lull you into thinking that you can just fall back on browser detection, that approach fails miserably as bugs and differences may be fixed in future versions of IE. And bear in mind that the other browsers aren't immune from inadvertently introducing problems and differences.

Feature detection is always preferred over browser detection when we need to make capability decisions, but it doesn't always come to our rescue. There *are* those rare moments when we'll need to resort to making browser-specific decisions (we'll see an example in a moment) that can only be made using browser detection. For those times, jQuery provides a set of flags that allow direct browser detection.

6.1.3 The browser detection flags

For those times when only browser detection will do, jQuery provides a set of flags that we can use for branching that are set up when the library is loaded, making them available even before any ready handlers have executed. They are defined as properties of an object instance with a reference of `$.browser`.

Note that even though these flags are present in jQuery 1.3, they are regarded as deprecated – meaning that they could be removed from any future release of jQuery and should be used with that in mind. These flags may have made more sense during the period where browser development had stagnated somewhat, but now that we've entered an era where browser development has picked up the pace, the capability support flags make more sense and are likely to stick around for some time.

In fact, it is recommended that, for those times when you need something more than the core support flags provide, that you create new ones of your own. But we'll get to that in just a bit.

First, the browser support flags are described in table 6.3.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Table 6.3 The $.browser user agent detection flags

Flag property	Description
`msie`	Set to `true` if the user agent is identified as any version of Internet Explorer.
`mozilla`	Set to `true` if the user agent is identified as any of the Mozilla-based browsers. This includes browsers such as Firefox and Camino.
`safari`	Set to `true` if the user agent is identified as any of the Webkit-based browsers such as Safari, Chrome and OmniWeb.
`opera`	Set to `true` if the user agent is identified as Opera.
`version`	Set to the version number of the rendering engine for the browser.

Note that these flags don't attempt to identify the specific browser that's being used; jQuery classifies a user agent based upon which *family* of browsers it belongs to; usually determined by which rendering engine it uses. Browsers within each family will sport the same sets of characteristics; specific browser identification should not be necessary.

The vast majority of commonly used, modern browsers will fall into one of these four browser families including Google Chrome, which returns `true` for the `safari` flag due to its use of the WebKit engine.

The `version` property deserves special notice because it's not as handy as we might think. The value set into this property isn't the version of the browser (as we might initially believe) but the version of the browser's rendering engine. For example, when executed within Firefox 3.05., the reported version is `1.9.0.5`—the version of the Gecko rendering engine. This value *is* handy for distinguishing between IE6 and IE7, containing `6.0` and `7.0` respectively.

We mentioned earlier that there are times when we can't fall back on feature detection and must resort to browser detection. One example of such a situation is when the difference between browsers isn't that they present different object classes or different methods, but that the parameters passed to a method are interpreted differently across the browser implementations. In such a case, there's no object to perform detection on.

NOTE

Even in these cases, it's possible to set a feature flag by actually trying the operation in a hidden area of the page (as jQuery does to set some of its feature flags). But that's not a technique we often see used on many pages outside of jQuery.

Let's take the `add()` method of `<select>` elements as an example. It's defined as the following:

```
selectElement.add(element,before)
```

For this method, the first parameter identifies an `<option>` or `<optgroup>` element to add to the `<select>` element and the second identifies the existing `<option>` (or `<optgroup>`) that the new element is to be placed before. In standards-compliant browsers, this second parameter is a *reference* to the existing element; in Internet Explorer, it's the *ordinal index* of the existing element.

Because there's no way to perform feature detection to determine if we should pass an object reference or an integer value (short of actually trying it out as noted above), we can resort to browser detection, as shown in the following example:

```
var select = $('#aSelectElement')[0];
select.add(
  new Option('Two and \u00BD','2.5'), $.browser.msie ? 2 : select.options[2]
);
```

In this code, we perform a simple test of the `$.browser.msie` flag in order to determine whether it is appropriate to pass the ordinal value 2, or the reference to the third option in the `<select>` element.

The jQuery team however, recommends that we not directly use such browser detection in our code. Rather, it is recommended that we abstract away the browser detection by creation a custom support flag of our own. That way, should the browser support flags vanish, our code is insulated from the change by merely finding another way to set the flag in one location.

For example, somewhere in our JavaScript library code we could write:

```
$.support.useIntForSelectAdds = $.browser.msie;
```

and use that flag in our code. Should the browser detection flag ever be removed, we'd only have to change our library code; all the code that uses the custom flag would be insulated from the change.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Let us now leave the world of flags and look at the utility functions that jQuery provides.

6.2 Using other libraries with jQuery

Back in chapter 1, we introduced a means, thoughtfully provided for us by the jQuery team, to easily use jQuery on the same page as other libraries. Usually, the definition of the $ variable is the largest point of contention and conflict when using other libraries on the same page as jQuery. As we know, jQuery uses $ as an alias for the `jQuery` name, which is used for every feature that jQuery exposes. But other libraries, most notably Prototype, use the $ name as well.

jQuery provides the `$.noConflict()` utility function to relinquish control of the $ name to whatever other library might wish to use it. The syntax of this function is as follows:

Function syntax: $.noConflict

`$.noConflict(jqueryToo)`

Restores control of the $ name back to another library, allowing mixed library use on pages using jQuery. Once this function is executed, jQuery features will need to be invoked using the `jQuery` name rather than the $ name.
Optionally, the jQuery name can also be given up.
This method should be called after including jQuery, but before including the conflicting library.

Parameters

`jqueryToo`	(Boolean) If provided and set to `true`, the `jQuery` name is given up in addition to the $.

Returns

jQuery

Because $ is an alias for `jQuery`, all of jQuery's functionality is still available after the application of `$.noConflict()`, albeit by using the `jQuery` identifier. To compensate for the loss of the brief—yet beloved—$, we can define our own shorter, but non-conflicting, alias for `jQuery`, such as

```
var $j = jQuery;
```

Another idiom we may often see employed is to create an environment where the $ name is scoped to refer to the `jQuery` object. This technique is commonly used when extending jQuery, particularly by plugin authors who can't make any assumptions regarding whether page authors have called `$.noConflict()` and who, most certainly, can't subvert the wishes of the page authors by calling it themselves.

This idiom is as follows:

```
(function($) { /* function body here */ })(jQuery);
```

If this notation makes your head spin, don't worry! It's pretty straightforward, even if odd-looking to those encountering it for the first time.

Let's dissect the first part of this idiom:

```
(function($) { /* function body here */ })
```

This part declares a function and encloses it in parentheses to make an expression out of it, resulting in a reference to the anonymous function being returned as the value of the expression. The function expects a single parameter, which it names $; whatever is passed to this function can be referenced by the $ identifier within the body of the function. And because parameter declarations have precedence over any similarly named identifiers in the global scope, any value defined for $ outside of the function is superseded within the function by the passed argument.

The second part of the idiom

```
(jQuery)
```

performs a function call on the anonymous function passing the `jQuery` object as the argument.

As a result, the $ name refers to the `jQuery` object within the body of the function regardless of whether it's already defined by Prototype or some other library *outside* of the function. Pretty nifty, isn't it?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

When employing this technique, the external declaration of $ isn't available within the body of the function.

A variant of this idiom is also frequently used to form a third syntax for declaring a ready handler in addition to the means that we already examined in chapter 1. Consider the following:

```
jQuery(function($) {
  alert("I'm ready!");
});
```

By passing a function as the parameter to the `jQuery` function, we declare it as a ready handler as we saw in chapter 1. But this time, we declare a single parameter to be passed to the ready handler using the $ identifier. Because jQuery always passes a reference to `jQuery` to a ready handler as its first and only parameter, this guarantees that the $ name refers to `jQuery` inside the ready handler regardless of the definition $ might have outside the body of the handler.

Let's prove it to ourselves with a simple test. For the first part of the test, let's examine the HTML document of listing 6.1.

Listing 6.1 Ready handler test 1

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hi!</title>
    <script type="text/javascript" src="../scripts/jquery-1.3.2.js"></script>
    <script type="text/javascript">
      var $ = 'Hi!';                                    #1
      jQuery(function(){                                #2
        alert('$ = '+ $);                               #2
      });                                               #2
    </script>
  </head>
  <body></body>
</html>
```

#1 Overrides $ name with custom value
#2 Declares the ready handler

Replace #1-2 in the following paragraph with a cueball

In this example, we import jQuery, which (as we know) defines the global names `jQuery` and its alias $. We then redefine the global $ name to a string value #1, overriding the jQuery definition. We replace $ with a simple string value for simplicity within this example, but it could be redefined by including another library such as Prototype.

We then define the ready handler #2 whose only action is to display an alert showing the value of $.

When we load this page, we see the alert displayed, as shown in figure 6.1.

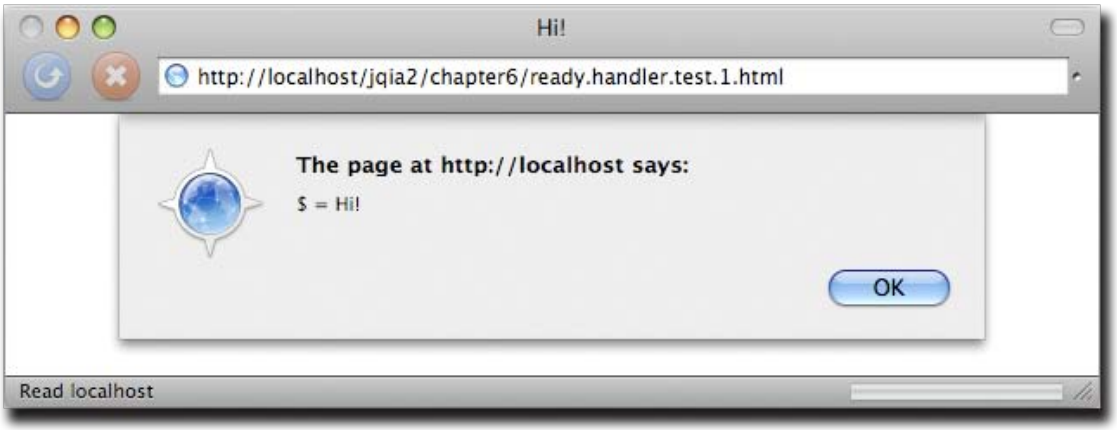


Figure 6.1 The $ says, "Hi!" as its redefinition takes effect within the ready handler.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Note that, within the ready handler, the *global* value of $ is in scope and has the expected redefined value resulting from our string assignment. How disappointing if we wanted to use the jQuery definition of $ within the handler.

Now let's make one change to this example document. The following code shows only the portion of the document that has been modified; the minimal change is highlighted in bold.

```
<script type="text/javascript">
  var $ = 'Hi!';
  jQuery(function($){
    alert('$ = '+ $);
  });
</script>
```

The only change we made was to add a parameter to the ready handler function named $. When we load this changed version, we see something completely different as shown in figure 6.4.

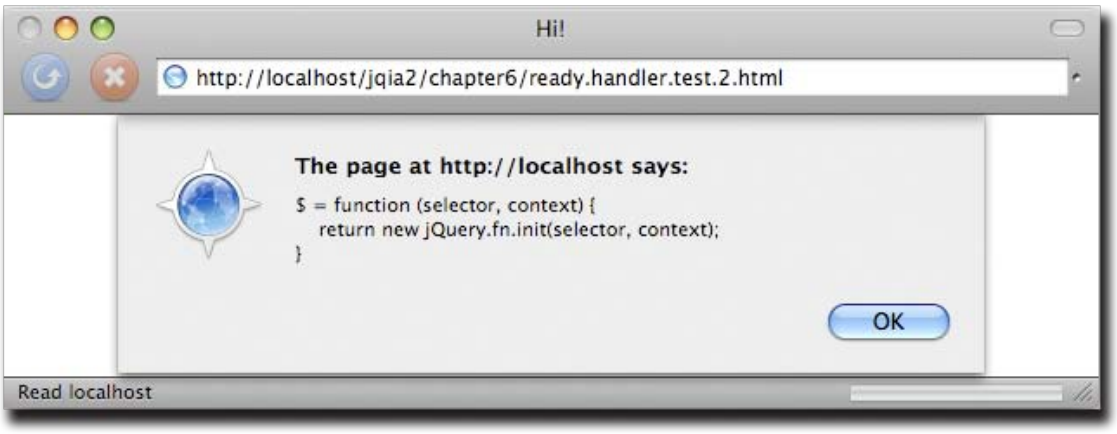


Figure 6.2 The alert now displays the jQuery version of $ because its definition has been enforced within the function.

Well, that may not have been exactly what we might have predicted in advance, but a quick glance at the jQuery source code shows that, because we declare the first parameter of the ready handler to be $ within that function, the $ name refers to the `jQuery` function that jQuery passes as the sole parameter to all ready handlers (so the alert displays the definition of that function).

When writing reusable components, which might or might not be used in pages where `$.noConflict()` is used, it's best to take such precautions regarding the definition of $.

A good number of the remaining jQuery utility functions are used to manipulate JavaScript objects. Let's take a good look at them.

6.3 Manipulating JavaScript objects and collections

The majority of jQuery features implemented as utility functions are designed to operate on JavaScript objects other than the DOM elements. Generally, anything designed to operate on the DOM is provided as a jQuery wrapper method. Although some of these functions can be used to operate on DOM elements—which *are* JavaScript objects, after all—the focus of the utility functions isn't DOM-centric. xx

Let's start with one that's pretty basic.

6.3.1 Trimming strings

Almost inexplicably, the JavaScript `String` implementation doesn't possess a method to remove whitespace characters from the beginning and end of a string instance. Such basic functionality is customarily part of a `String` class in most other languages, but JavaScript mysteriously lacks this useful feature.

Yet string trimming is a common need in many JavaScript applications; one prominent example is during form data validation. Because whitespace is invisible on the screen (hence its name), it's easy for users to accidentally enter extra space

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

characters after (or sometimes even before) valid entries in text boxes or text areas. During validation, we want to silently trim such whitespace from the data rather than alerting the user to the fact that something that they can't see is tripping them up.

To help us out, jQuery defines the `$.trim()` function as follows:

Function syntax: $.trim

`$.trim(value)`

Removes any leading or trailing whitespace characters from the passed string and returns the result.
Whitespace characters are defined by this function as any character matching the JavaScript regular expression `\s`, which matches not only the space character but also the form feed, new line, return, tab, and vertical tab characters, as well as the Unicode characters `\u00A0`, `\u2028`, and `\u2029`.

Parameters

`value` (String) The string value to be trimmed. This original value isn't modified.

Returns

The trimmed string.

A small example of using this function to trim the value of a text field in-place is

```
$('#someField').val($.trim($('#someField').val()));
```

Be aware that this function doesn't check the parameter we pass to ensure that it's a String value, so we'll likely get undefined and unfortunate results (probably a JavaScript error) if we pass any other value type to this function.

Now let's look at some functions that operate on arrays and other objects.

6.3.2 Iterating through properties and collections

Oftentimes when we have non-scalar values composed of other components, we'll need to iterate over the contained items. Whether the container element is a JavaScript array (containing any number of other JavaScript values, including other arrays) or instances of JavaScript objects (containing properties), the JavaScript language gives us means to iterate over them. For arrays, we iterate over their elements using the `for` loop; for objects, we iterate over their properties using the `for-in` loop.

We can code examples of each as follows:

```
var anArray = ['one','two','three'];
for (var n = 0; n < anArray.length; n++) {
  //do something here
}
var anObject = {one:1, two:2, three:3};
for (var p in anObject) {
  //do something here
}
```

Pretty easy stuff, but some might think that the syntax is needlessly wordy and complex—a criticism frequently targeted at the `for` loop. We know that, for a wrapped set of DOM elements, jQuery defines the `each()` method, allowing us to easily iterate over the elements in the set without the need for messy `for`–loop syntax. For general arrays and objects, jQuery provides an analogous utility function named `$.each()`.

The really nice thing is that the same syntax is used whether iterating over the items in an array or the properties of an object.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Function syntax: $.each

`$.each(container,callback)`

Iterates over the items in the passed container, invoking the passed callback function for each.

Parameters

`container`	(Array|Object) An array whose items, or an object whose properties, are to be iterated over.
`callback`	(Function) A function invoked for each element in the container. If the container is an array, this callback is invoked for each array item; if it's an object, the callback is invoked for each object property. The first parameter to this callback is the index of the array element or the name of the object property. The second parameter is the array item or property value. The function context (`this`) of the invocation is also set to the value passed as the second parameter.

Returns

The container object.

This unified syntax can be used to iterate over either arrays or objects using the same format. With this function, we can write the previous example as follows:

```
var anArray = ['one','two','three'];
$.each(anArray,function(n,value) {
  //do something here
});

var anObject = {one:1, two:2, three:3};
$.each(anObject,function(name,value) {
  //do something here
});
```

Although using `$.each()` with an inline function may seem like a six-of-one scenario in choosing syntax, this function makes it easy to write reusable iterator functions or to factor out the body of a loop into another function for purposes of code clarity as in the following:

```
$.each(anArray,someComplexFunction);
```

Note that when iterating over an array or object, we can break out of the loop by returning `false` from the iterator function.

PERFORMANCE NOTE

You may recall that we can also use the `each()` method to iterate over an array, but the `$.each()` function has a slight performance advantage over `each()`. However, if you need to be concerned with performance to that level, you'll get best performance from a good old-fashioned `for` loop.

Sometimes we may iterate over arrays to pick and choose elements to become part of a new array. While we could use `$.each()` for that purpose, let's see how jQuery makes that even easier.

6.3.3. Filtering arrays

Traversing an array to find elements that match certain criteria is a frequent need of applications that handle lots of data. We might wish to filter through the data looking for items that fall above or below a particular threshold or, perhaps, that match a certain pattern. For any filtering operation of this type, jQuery provides the `$.grep()` utility function.

The name of the `$.grep()` function might lead us to believe that the function employs the use of regular expressions like its namesake, the UNIX `grep` command. But the filtering criteria used by the `$.grep()` utility function isn't a regular expression; it's a callback function provided by the caller that defines the criteria to determine if a data value should be included

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

or excluded from the resulting set of values. Nothing prevents that callback from *using* regular expressions to accomplish its task, but the use of regular expressions is not automatic.

The syntax of the function is as follows:

Function syntax: $.grep

`$.grep(array,callback,invert)`

Traverses the passed array, invoking the callback function for each element. The return value of the callback function determines if the value is collected into a new array returned as the value of the `$.grep()` function. If the `invert` parameter is omitted or `false`, a callback value of `true` causes the data to be collected. If `invert` is `true`, a callback value of `false` causes the value to be collected.
The original array isn't modified.

Parameters

`array`	(Array) The traversed array whose data values are examined for collection. This array isn't modified in any way by this operation.
`callback`	(Function) A function whose return value determines if the current data value is to be collected. A return value of `true` causes the current value to be collected, unless the value of the `invert` parameter is `true` in which case the opposite occurs. This function is passed two parameters: the current data value and the index of that value within the original array.
`invert`	(Boolean) If specified as `true`, it inverts the normal operation of the function.

Returns

The array of collected values.

Let's say that we want to filter an array for all values that are greater than 100. We would do that with a statement such as the following:

```
var bigNumbers = $.grep(originalArray,function(value) {
                   return value > 100;
                 });
```

The callback function that we pass to `$.grep()` can use whatever processing it likes to determine if the value should be included. The decision could be as easy as this example or, perhaps, even as complex as making synchronous Ajax calls (with the requisite performance hit) to the server to determine if the value should be included or excluded.

Even though the `$.grep()` function doesn't directly use regular expressions (despite its name), JavaScript regular expressions can be powerful tools for us to use in our callback functions to determine whether to include or exclude values from the resultant array. Consider a situation in which we have an array of values and wish to identify any values that don't match the pattern for United States Postal Codes (also known as ZIP Codes).

US Postal Codes consist of five decimal digits optionally followed by a dash and four more decimal digits. A regular expression for such a pattern would be `/^\d{5}(-\d{4})?$/`, so we could filter a source array for non-conformant entries with the following:

```
var badZips = $.grep(
                originalArray,
                function(value) {
                  return value.match(/^\d{5}(-\d{4})?$/) != null;
                },
                true);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Notable in this example is the use of the `String` class's `match()` method to determine whether a value matches the pattern or not and the specification of the `invert` parameter to `$.grep()` as `true` to *exclude* any values that match the pattern.

Collecting subsets of data from arrays isn't the only operation we might perform upon them. Let's look at another array-targeted function that jQuery provides.

6.3.4 Translating arrays

Data might not always be in the format that we need it to be. Another common operation that's frequently performed in data-centric web applications is the *translation* of a set of values to another set. Although it's a simple matter to write a `for` loop to create one array from another, jQuery makes it even easier with the `$.map` utility function.

Function syntax: $.map

`$.map(array,callback)`

Iterates through the passed array, invoking the callback function for each array item and collecting the return values of the function invocations in a new array.

Parameters

`array`	(Array) The array whose values are to be transformed to values in the new array.
`callback`	(Function) A function whose return values are collected in the new array returned as the result of a call to the `$.map()` function. This function is passed two parameters: the current data value and the index of that value within the original array.

Returns

The wrapped set.

Let's look at a trivial example that shows the `$.map()` function in action.

```
var oneBased = $.map([0,1,2,3,4],function(value){return value+1;});
```

This statement converts an array of values, a zero-based set of indexes to a corresponding array of one-based indexes.

An important behavior to note is that if the function returns either `null` or `undefined`, the result isn't collected. In such cases, the resulting array will be smaller in length than the original, and one-to-one correspondence between items by order is lost.

Let's look at a slightly more involved example. Imagine that we have an array of strings, perhaps collected from form fields, that are expected to represent numeric values and that we want to convert this string array to an array of corresponding `Number` instances. Because there's no guarantee against the presence of an invalid numeric string, we need to take some precautions. Consider the following code:

```
var strings = ['1','2','3','4','S','6'];

var values = $.map(strings,function(value){
  var result = new Number(value);
  return isNaN(result) ? null : result;
});
```

We start with an array of string values, each of which is expected to represent a numeric value. But a typo (or perhaps user entry error) resulted in *S* instead of the expected *5*. Our code handles this case by checking the `Number` instance created by the constructor to see if the conversion from string to numeric was successful or not. If the conversion fails, the value returned will be the constant `Number.NaN`. But the funny thing about `Number.NaN` is that by definition, it doesn't equal anything else, *including* itself! Therefore the value of the expression `Number.NaN==Number.NaN` is `false`!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Because we can't use a comparison operator to test for `NaN` (which stands for *Not a Number*, by the way), JavaScript provides the `isNaN()` method, which we employ to test the result of the string-to-numeric conversion.

In this example, we return `null` in the case of failure, ensuring that the resulting array contains only the valid numeric values with any error values elided. If we want to collect all the values, we can allow the transformation function to return `Number.NaN` for bad values.

Another useful behavior of `$.map()` is that it gracefully handles the case where an *array* is returned from the transformation function, merging the returned value into the resulting array. Consider the following statement:

```
var characters = $.map(
  ['this','that','other thing'],
  function(value){return value.split('');}
);
```

This statement transforms an array of strings into an array of all of the characters that make up the strings. After execution, the value of the variable `characters` is as follows:

```
['t','h','i','s','t','h','a','t','o','t','h','e','r','','t','h', 'i','n','g']
```

This is accomplished by use of the `String.split()` method, which returns an array of the string's characters when passed an empty string as its delimiter. This array is returned as the result of the transformation function and is merged into the resultant array.

jQuery's support for arrays doesn't stop there. There are a handful of minor functions that we might find handy.

6.3.5. More fun with JavaScript arrays

Have you ever needed to know if a JavaScript array contained a specific value and, perhaps, even the location of that value in the array?

If so, you'll appreciate the `$.inArray()` function

Function syntax: $.inArray

`$.inArray(value,array)`

Returns the index position of the first occurrence of the passed value

Parameters

`value`	(Object) The value for which the array will be searched
`array`	(Array) The array to be searched

Returns

The index of the first occurrence of the value within the array or -1 if the value is not found

A trivial but illustrative example of using this function is

```
var index = $.inArray(2,[1,2,3,4,5]);
```

This results in the index value of `1` being assigned to the `index` variable.

Another useful array-related function creates JavaScript arrays from other array-like objects. "Other *array-like objects*? What on Earth is an array-like object?" you may ask.

jQuery considers other *array-like objects* to be any object that has a length and the concept of indexed entries. This capability is most useful for `NodeList` objects. Consider the following snippet:

```
var images = document.getElementsByTagName("img");
```

This populates the variable `images` with a `NodeList` of all the images on the page.

Dealing with a `NodeList` is a bit of a pain, so converting it to a JavaScript array makes things a lot nicer. The jQuery `$.makeArray` function makes converting the `NodeList` easy.

Function syntax: $.makeArray

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`$.makeArray(object)`

Converts the passed array-like object into a JavaScript array

Parameters

`object` (Object) The array-like object (such as a `NodeList`) to be converted

Returns

The resulting JavaScript array

This function is intended for use in code that doesn't make much use of jQuery, which internally handles this sort of thing on our behalves. This function also comes in handy when dealing with `NodeList` objects while traversing XML documents without jQuery, or when handling the `arguments` instance within functions (which, you may be surprised to learn, is not a standard JavaScript array)..

Another seldom-used function that might come in handy when dealing with arrays built outside of jQuery is the `$.unique()` function.

Function syntax: $.unique

`$.unique(array)`

Given an array of DOM elements, returns an array of the unique elements in the original array

Parameters

`array` (Array) The array of DOM elements to be examined

Returns

An array of DOM elements consisting of the unique elements in the passed array

Again, this is a function that jQuery uses internally to ensure that the lists of elements that we receive contain unique elements, and is intended for use on element arrays created outside the bounds of jQuery.

Want to merge two arrays? No problem; there's the `$.merge` function:

Function syntax: $.merge

`$.merge(array1,array2)`

Merges the values of the second array into the first and returns the result. The first array is modified by this operation and returned as the result.

Parameters

`array1` (Array) An array into which the other array's values will be merged.

`array2` (Array) An array whose values will be merged into the first array.

Returns

The first array, modified with the results of the merge,

Consider:

```
var a1 = [1,2,3,4,5];
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
var a2 = [5,6,7,8,9];
$.merge(a1,a2);
```

After this sequence executes, `a2` is untouched, but `a1` contains: `[1,2,3,4,5,5,6,7,8,9]`.

The final array-oriented function we'll examine is one that simply tests to see if a reference is an array or not. The syntax of `$.isArray()` is as follows:

Function syntax: $.merge

`$.merge(value)`

Test if the passed value is an array or not.

Parameters

`value`	(Any) The value to be tested

Returns

`true` if the value is an array, `false` otherwise

Now that we've seen how jQuery helps us to easily work with arrays, let's see a way that it helps us manipulate plain old JavaScript objects.

6.3.6 Extending objects

Although we all know that JavaScript provides some features that make it act in many ways like an object-oriented language, we know that JavaScript isn't what anyone would call purely object-oriented because of the features that it doesn't support. One of these important features is *inheritance*—the manner in which new classes are defined by extending the definitions of existing classes.

A pattern for mimicking inheritance in JavaScript is to extend an object by copying the properties of a base object into the new object, extending the new object with the capabilities of the base.

NOTE

If you're an aficionado of "object-oriented JavaScript", you'll no doubt be familiar with extending not only object instances but also their blueprints via the `prototype` property of object constructors. `$.extend()` can be used to effect such constructor-based inheritance by extending `prototype`, as well as object-based inheritance by extending existing object instances (something jQuery does itself internally). Because understanding such advanced topics isn't a requirement in order to use jQuery effectively, this is a subject—albeit an important one—that's beyond the scope of this book.

It's fairly easy to write JavaScript code to perform this extension by copy, but as with so many other procedures, jQuery anticipates this need and provides a ready-made utility function to help us out. As we'll see in the next chapter, this function is useful for much more than extending an object, but even so its name is `$.extend()`. Its syntax is as follows:

Function syntax: $.extend

`$.extend(deep,target,source1,source2, ... sourceN)`

Extends the object passed as `target` with the properties of the remaining passed objects.

Parameters

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`deep`	(Boolean) An optional flag that determines whether a deep or shallow copy is made. If omitted or `false`, a shallow copy is executed. If `true`, a deep copy is performed.
`target`	(Object) The object whose properties are augmented with the properties of the source objects. This object is directly modified with the new properties before being returned as the value of the function. Any properties with the same name as properties in any of the source elements are overridden with the values from the source elements.
`source1 ... sourceN`	(Object) One or more objects whose properties are added to the `target` object. When more than one source is provided and properties with the same name exist in the sources, sources later in the argument list take precedence over those earlier in the list.

Returns

The extended target object.

Let's take a look at this function doing its thing.

We'll set up three objects, a target and two sources, as follows:

```
var target =  { a: 1, b: 2, c: 3 };
var source1 = { c: 4, d: 5, e: 6 };
var source2 = { e: 7, f: 8, g: 9 };
```

Then we'll operate on these objects using `$.extend()` as follows:

```
$.extend(target,source1,source2);
```

This should take the contents of the source objects and merge them into the target. To test this, we've set up this example code in the file of `chapter6/$.extend.html`, which executes the code and displays the results on the page.

Loading this page into a browser results in the display of figure 6.3.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

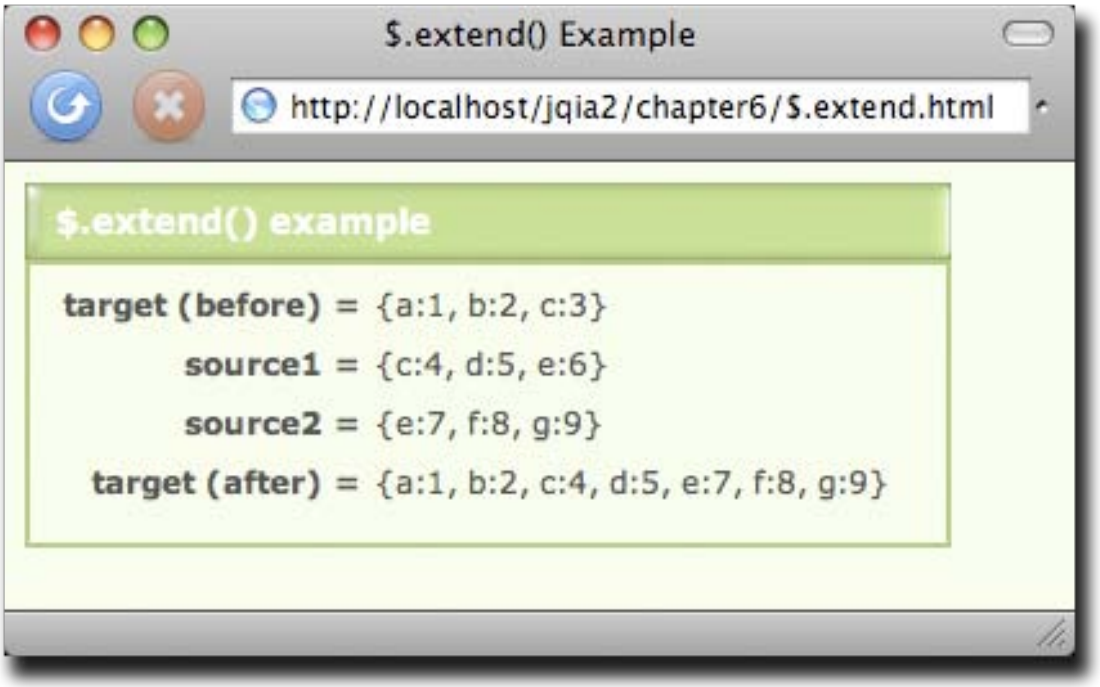


Figure 6.x [Add caption here]

As we can see, all properties of the source objects have been merged into the `target` object. But note the following important nuances:

- Both the `target` and `source1` contain a property named `c`. The value of `c` in `source1` replaces the value in the original target.
- Both `source1` and `source2` contain a property named `e`. Note that the value of `e` within `source2` takes precedence over the value within `source1` when merged into `target`, demonstrating how objects later in the list of arguments take precedence over those earlier in the list.

Although it's evident that this utility function can be useful in many scenarios where one object must be extended with properties from another object (or set of objects), we'll see a concrete and common use of this feature when learning how to define utility functions of our own in the next chapter.

But before we get to that, we've still got a few other utility functions to examine.

6.3.7 Serializing parameter values

It should come as no surprise that in a dynamic, DOM-scripted application, that the submitting of requests is a common occurrence. Heck, it's one of the things that makes the World Wide Web a web in the first place.

Frequently, these requests will be submitted as a result of a form submission, where the browser handles formatting the request body containing the request parameters on our behalves. Other times, we'll be submitting requests as URLs in the `href` attribute of `<a>` elements. In these latter cases, it becomes our responsibility to correctly create and format the query string that contains any request parameters that we wish to include with the request.

Server-side templating tools generally have great mechanisms that help us construct valid URLs, but when creating them dynamically on te client, JavaScript doesn't give us much in the way of support. Remember that not only do we need to correctly place all the ampersand (&) and equal signs (=) that format the query string parameters, we need to make sure that each name and value is properly URI-encoded. While JavaScript provides a handy function (`encodeURIComponent()`) for that, the formatting of the query string falls squarely into our laps.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

And as you might have come to expect, jQuery anticipates that burden and gives us a tool to make it easier: the `$.param()` utility function.

Function syntax: $.param

`$.param(params)`

Serializes the passed information into a string suitable for use as the query string of a submitted request. The passed value can be an array of form elements, a jQuery wrapped set, or a JavaScript object. The query string is properly formatted and each name and value in the string is properly URI-encoded.

Parameters

`params`	(Array|jQuery|Object) The value to be serialized into a query string. If an array of elements, or a jQuery wrapped set is passed, the name/value pairs represented by the included form controls are added to the query string. If a JavaScript object is passed, the object's properties form the parameter names and values.

Returns

The formatted query string.

Consider the following statement:

```
$.param({
  'a thing':'it&s=value',
  'another thing':'another value',
  'weird characters':'!@#$%^&*()_+='
});
```

Here, we pass an object with three properties to the `$.param()` function., in which the names and the values all contain characters that must be encoded within the query string in order for it to be valid. The result of this function call is:

```
a+thing=it%26s%3Dvalue&another+thing=another+value&weird+characters=!%40%23%24%25%5E%26*()_%2B%3D
```

Note how the query string is formatted correctly and that the non-alphanumeric characters in the names and values have been properly encoded. This might not make the string all that readable to us, but server-side code lives for such strings!

One note of caution: if you pass an array of elements, or a jQuery wrapped set, that contain elements other than those that represent form values, you'll end up with a bunch of entries such as:

```
&undefined=undefined
```

in the resulting string, as this function does not weed out inappropriate elements in its passed argument.

You might be thinking that this isn't a big deal because, after all, if the values are form elements, they're going to end up being submitted by the browser via the form, which is going to handle all of this for us. Well, hold onto your hat. In chapter 8, when we start talking about Ajax, we'll see that form elements aren't always submitted by their form!

But that's not going to be an issue because we'll also see later on that jQuery provides a higher-level means to handle this (that internally uses this utility function) to handle this sort of thing in a more sophisticated fashion.

6.3.8 Testing for functions

You may have noticed that many of the jQuery wrapper methods and utility function have rather malleable parameter lists; optional parameters can be omitted without the need to put `null` values as placeholders.

Take the `bind()` wrapper method as an example. Its function signature is:

```
bind(event,data,handler)
```

But if we have no data to pass to the event, we can simple call `bind()` with the handler function as the second parameter. jQuery handles this by testing the types of the parameters and if it sees that a function is passed as the second parameter, interprets it as the hander rather than a data argument.

Testing for functions will certainly come in handy if we want to create our own functions and methods that are just as friendly and versatile, so jQuery exposes the `$.isFunction()` utility function:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Function syntax: $.isFunction

`$.isFunction(candidate)`

Determines if the passed candidate object is a function.

Parameters

`candidate`	(Object) Thecandidate object to be tested.

Returns

`true` if the object is a function; `false` otherwise

Be aware that some built-in functions, such as `alert()`and DOM element methods, are not correctly identified as functions when running in Internet Explorer.

Let's wrap up our investigation of the utility functions with one that we can use to dynamically load new script into our pages.

6.4 Dynamically loading scripts

Most of the time—perhaps, almost always—we'll load the external scripts our page needs from script files when the page loads via `<script>` tags in the `<head>` of the page. But every now and again, we might want to load some script after the fact under script control.

We might do this because we don't know if the script will be needed until after some specific user activity has taken place but don't want to include the script unless absolutely needed, or perhaps, we might need to use some information not available at load time to make a conditional choice between various scripts.

Regardless of why we might want to dynamically load new script into the page, jQuery provides the `$.getScript()` utility function to make it easy.

Function syntax: $.getScript

`$.getScript(url,callback)`

Fetches the script specified by the `url` parameter using a GET request to the specified server, optionally invoking a callback upon success.

Parameters

`url`	(String) The URL of the script file to fetch. The URL is *not* restricted to the same domain as the containing page.
`callback`	(Function) An optional function invoked after the script resource has been loaded and evaluated, with the following parameters: the text loaded from the resource, and a text status message; "success" if all has gone well.

Returns

The XmlHTTPRequest instance used to fetch the script.

Under its covers, this function uses jQuery's built-in Ajax mechanisms to fetch the script file. We'll be covering these Ajax facilities in great detail in chapter 8, but we don't need to know anything about Ajax to use this function.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

After fetching, the script in the file is evaluated; any inline script is executed, and any defined variables or functions become available.

WARNING

In Safari 2 and older,, the script definitions loaded from the fetched file don't become available right away, even in the callback to the function. Any dynamically loaded script elements don't become available until after the script block within which it is loaded relinquishes control back to the browser. If your pages are going to support these older versions of Safari, plan accordingly!

Let's see this in action. Consider the following script file (available in chapter6/new.stuff.js):

```
alert("I'm inline!");
var someVariable = 'Value of someVariable';
function someFunction(value) {
  alert(value);
};
```

This trivial script file contains an inline statement (which issues an alert that leaves no doubt as to when the statement gets executed), a variable declaration, and a declaration for a function that issues an alert containing whatever value is passed to it when executed. Now let's write a page to include this script file dynamically. The page is shown in listing 6.2 and can be found in the file `chapter6/$.getScript.html`.

Listing 6.2 Dynamically loading a script file and examining the results

```
<!DOCTYPE html>
<html>
  <head>
    <title>$.getScript() Example</title>
    <link rel="stylesheet" type="text/css" href="../styles/core.css">
    <script type="text/javascript" src="../scripts/jquery-1.3.2.js"></script>
    <script type="text/javascript" src="../scripts/jqia2.support.js"></script>
    <script type="text/javascript">
      $(function(){
        $('#loadButton').click(function(){                              #1
          $.getScript(
            'new.stuff.js'
            //,function(){$('#inspectButton').click()}
          );
        });
        $('#inspectButton').click(function(){                           #2
          someFunction(someVariable);
        });
      });
    </script>
  </head>

  <body>                                                                #3
    <button type="button" id="loadButton">Load</button>
    <button type="button" id="inspectButton">Inspect</button>
  </body>
</html>
```

#1 Fetches the script on clicking the Load button
#2 Displays result on clicking the Inspect button
#3 Contains the test buttons

Cueballs in code and text

This page defines two buttons (#3) that we use to trigger the activity of the example. The first button, labeled Load, causes the `new.stuff.js` file to be dynamically loaded through use of the `$.getScript()` function (#1). Note that, initially, the second parameter (the callback) is commented out—we'll get to that in a moment.

On clicking the Load button, the `new.stuff.js` file is loaded, and its content is evaluated. As expected, the inline statement within the file triggers an alert message as shown in figure 6.4.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

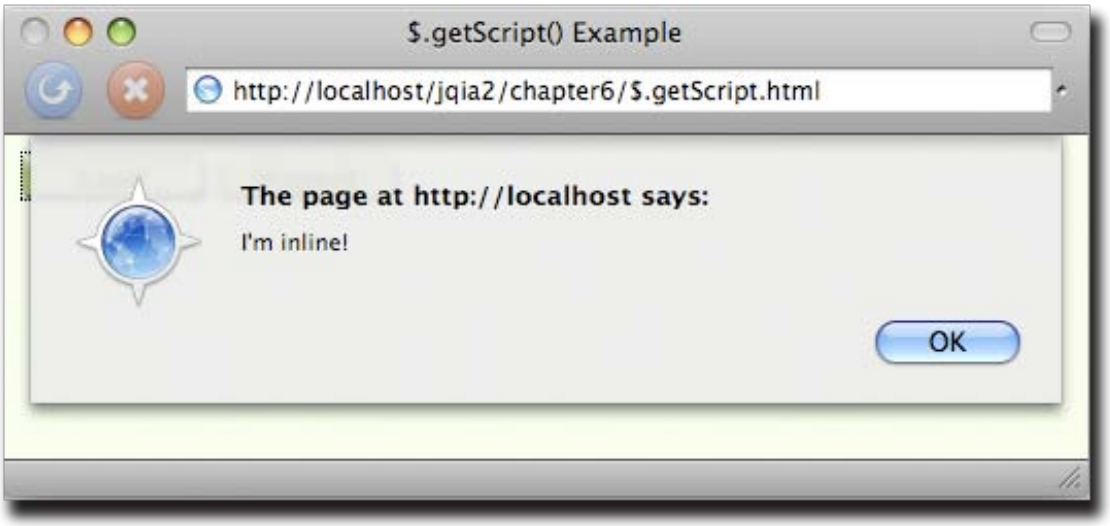


Figure 6.4 The dynamic loading and evaluation of the script file results in the inline alert statement being executed.

Clicking the Inspect button executes its `click` handler (`#2`), which executes the dynamically loaded `someFunction()` function passing the value of the dynamically loaded `someVariable` variable. If the alert appears as shown in figure 6.5, we know that both the variable and function are loaded correctly.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

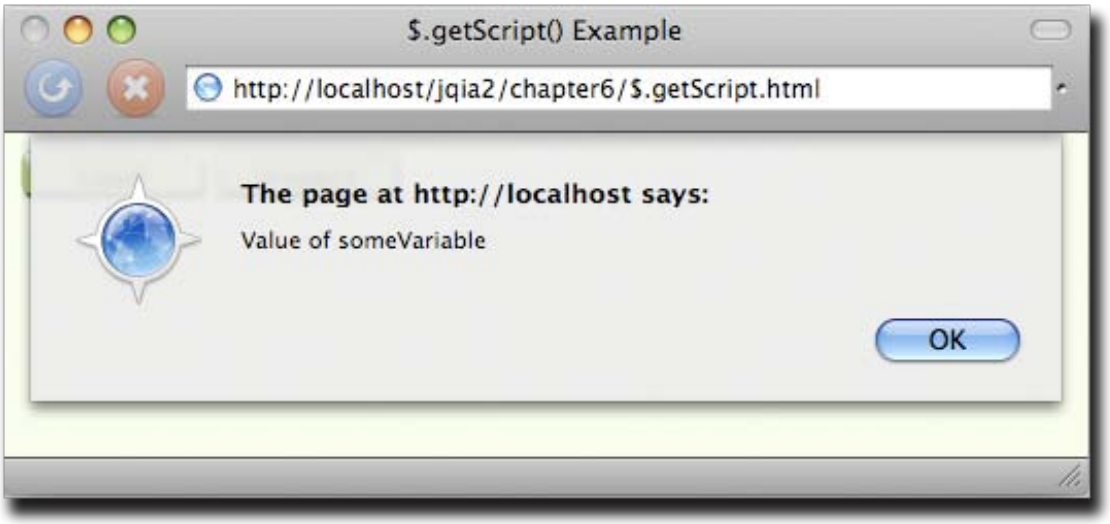


Figure 6.5 The appearance of the alert shows that the dynamic function is loaded correctly, and the correctly displayed value shows that the variable was dynamically loaded.

If are still running Safari 2 or older, and would like to observe the behavior of older versions of Safari that we warned you about earlier, make a copy of the HTML file of listing 6.5, and uncomment the callback parameter to the `$.getScript()` function. This callback executes the `click` handler for the Inspect button, calling the dynamically loaded function with the loaded variable as its parameter.

In browsers other than Safari, the function and variable loaded dynamically from the script are available within the callback function. But when executed on Safari, nothing happens! We need to take heed of this divergence of functionality when using the `$.getScript()` function in Safari's older versions.

6.5 Summary

In this chapter we surveyed the features that jQuery provides outside of the methods that operate upon a wrapped set of matched DOM elements. These included an assortment of functions, as well as a set of flags, defined directly on the `jQuery` top-level name (as well as its `$` alias).

We saw how jQuery informs us as to the capabilities of the containing browser using the various flags in the `$.support` object. When we need to resort to browser detection to account for differences in browser capabilities and operation beyond what `$.support` provides, the `$.browser` set of flags lets us determine within which browser family the page is being displayed. Browser detection should be used only as a last resort when it's impossible to write the code in a browser-independent fashion, and the preferred approach of feature detection can't be employed.

Recognizing that page authors may sometimes wish to use other libraries in conjunction with jQuery, jQuery provides `$.noConflict()`, which allows other libraries to use the `$` alias. After calling this function, all jQuery operations must use the `jQuery` name rather than `$`.

`$.trim()` exists to fill the gap left by the native JavaScript `String` class for trimming whitespace from the beginning and end of string values.

jQuery also provides a set of functions that are useful for dealing with data sets in arrays. `$.each()` makes it easy to traverse through every item in an array; `$.grep()` allows us to create new arrays by filtering through the data of a source array using whatever filtering criteria we would like to use; and `$.map()` allows us to easily apply our own transformations to a source array to produce a corresponding new array with the transformed values.

We can convert NodeList instances to JavaScript arrays with `$.makeArray()`, test to see if a value is in an array with `$.inArray()`, and even test if a value is an array itself with `$.isArray()`. We can also test for functions using `$.isFunction()`.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

We also saw how jQuery lets us construct properly formatted and encoded query strings with `$.param()`.

To merge objects, perhaps even to mimic a sort of inheritance scheme, jQuery also provides the `$.extend()` function. This function allows us to unite the properties of any number of source objects into a target object.

And for those times when we want to load a script file dynamically, jQuery defines `$.getScript()`, which can load and evaluate a script file at any point in the lifetime of a page, even from domains other than the page source.

With these additional tools safely tucked away in our toolbox, we're ready to tackle how to add our own extensions to jQuery. Let's get to it in the next chapter.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

7

Expand Your Reach by Extending jQuery

This chapter covers:

- Why to extend jQuery with custom code
- Guidelines for effectively extending jQuery
- Writing custom utility functions
- Writing custom wrapper methods

Over the course of the previous chapters, we've seen that jQuery gives us a large toolset of useful methods and functions, and we've also seen that we can easily tie these tools together to give our pages whatever behavior we choose. Sometimes that code follows common patterns we want to use again and again. When such patterns emerge, it makes sense to capture these repeated operations as reusable tools that we can add to our original toolset. In this chapter, we explore how to capture these reusable fragments of code as extensions to jQuery.

But before any of that, let's discuss *why* we'd want to pattern our own code as extensions to jQuery in the first place.

7.1 Why extend?

If you've been paying attention at all while reading through this book, as well as to the code examples presented within it, you undoubtedly have noted that adopting jQuery for use in our pages has a profound effect on how script is written within a page.

The use of jQuery promotes a certain style for a page's code, frequently in the guise of forming a wrapped set of elements and then applying a jQuery method, or chain of methods, to that set. When writing our own code, we can write it however we please, but most experienced developers agree that having all of the code on a site, or at least the great majority of it, adhere to a consistent style is a good practice.

So one good reason to pattern our code as jQuery extensions is to help maintain a consistent code style throughout the site.

Not reason enough? Need more? The whole point of jQuery is to provide a set of reusable tools and APIs. The creators of jQuery carefully planned the design of the library and the philosophy of how the tools are arranged to promote reusability. By following the precedent set by the design of these tools, we automatically reap the benefit of the planning that went into these designs—a compelling second reason to write our code as jQuery extensions.

Still not convinced? The final reason we'll consider (though it's quite possible others could list even more reasons) is that, by extending jQuery, we can leverage the existing code base that jQuery makes available to us. For example, by creating new jQuery methods (wrapper methods), we automatically inherit the use of jQuery's powerful selector mechanism. Why would we write everything from scratch when we can layer upon the powerful tools jQuery already provides?

Given these reasons, it's easy to see that writing our reusable components as jQuery extensions is a good practice and a smart way of working. In the remainder of this chapter, we'll examine the guidelines and patterns that allow us to create jQuery plugins and we'll create a few of our own. In the following chapter, which covers a completely different subject (Ajax),

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

we'll see even more evidence that creating our own reusable components as jQuery plugins in real-world scenarios helps to keep the code consistent and makes it a whole lot easier to write those components in the first place.

But first, the quidelines…

7.2 The jQuery plugin authoring guidelines

> Sign! Sign! Everywhere a sign! Blocking out the scenery, breaking my mind. Do this! Don't do that! Can't you read the sign?
>
> —Five Man Electric Band, 1971

Although the Five Man Electric Band may have lyrically asserted an anti-establishment stance against rules back in 1971, sometimes rules are a good thing. Without any, chaos would reign.

Such it is with the rules—more like common-sense guidelines—governing how to successfully extend jQuery with our own plugin code. These guidelines help us to ensure that not only does our new code plug into the jQuery architecture properly, but also that it will work and play well with other jQuery plugins, and even other JavaScript libraries.

Extending jQuery takes one of two forms:

- Utility functions defined directly on $ (an alias for `jQuery`)
- Methods to operate on a jQuery wrapped set (so-called jQuery *methods*)

In the remainder of this section, we'll go over some guidelines common to both types of extensions. Then in the following sections, we'll tackle the guidelines and techniques specific to writing each type of plugin.

7.2.1 Naming files and functions

To Tell the Truth was an American game show, first airing in the 1950's, in which multiple contestants claimed to be the same person with the same name, and a panel of celebrities was tasked with determining which person was really whom they claimed to be. Although fun for a television audience, *name collisions* are not fun at all when it comes to programming.

> **NOTE**
>
> Name collisions are especially bothersome in scripting languages like JavaScript where they frequently manifest themselves as galling run-time problems that can be difficult to debug, as opposed to the blatant compile-time errors produced by strongly-typed, compiled languages like Java or C++.

We'll discuss avoiding name collisions *within* our plugins, but first let's address naming the files within which we'll write our plugins so that they do not conflict with other files.

The guideline recommended by the jQuery team is simple but effective, advocating the following format:

- Prefix the filename with *jquery.*
- Follow that with the name of the plugin.
- Conclude with *.js.*

For example, if we write a plugin that we want to name "Fred", our JavaScript filename for this plugin is `jquery.fred.js`

The use of the "jquery" prefix eliminates any possible name collisions with files intended for use with other libraries. After all, anyone writing non-jQuery plugins has no business using the "jquery" prefix, but that leaves the plugin name itself still open for contention *within* the jQuery community.

When we're writing plugins for our own use, all we need to do is avoid conflicts with any other plugins that we plan to use. But when writing plugins that we plan to publish for others to use, we need to avoid conflicts with any other plugin that's already published.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

The best way to avoid conflicts is to stay in tune with the goings-on within the jQuery community. A good starting point is the page at http://docs.jquery.com/Plugins; but, beyond being aware of what's already out there, there are other precautions we can take.

One way to ensure that our plugin filenames are unlikely to conflict with others is to sub-prefix them with a name that's unique to us or our organization. For example, all of the plugins developed in this book use the filename prefix "jquery.jqia" (*jqia* being short for *jQuery in Action*) to help make sure that they won't conflict with anyone else's plugin filenames should anyone wish to use them in their own web applications. Likewise, the files for the jQuery Form Plugin begin with the prefix *jquery.form*. Not all plugins follow this convention, but as the number of plugins increases, it will become more and more important to follow such conventions.

Similar considerations need to be taken with the *names* we give to our functions, whether they're new utility functions or methods on the jQuery wrappers.

When creating plugins for our own use, we're usually aware of what other plugins we'll use; it's an easy matter to avoid any naming collisions. But what if we're creating our plugins for public consumption? Or what if our plugins, that we initially intended to use privately, turn out to be so useful that we want to share them with the rest of the community?

Once again, familiarity with the plugins that already exist will go a long way in avoiding API collisions, but we also encourage gathering collections of related functions under a common prefix (similar to the proposal for filenames) to avoid cluttering the namespace.

Now, what about conflicts with that $?

7.2.2 Beware the $

"Will the real $ please stand up?"

Having written a fair amount of jQuery code, we've seen how handy it is to use the $ alias in place of `jQuery`. But when writing plugins that may end up in other people's pages, we can't be quite so cavalier. As plugin authors, we have no way of knowing whether a page author intends to use the `$.noConflict()` function to allow the $ alias to be usurped by another library.

We could employ the sledgehammer approach and use the `jQuery` name in place of the $ alias, but dang it, we *like* using $ and are loath to give up on it so easily.

Chapter 6 introduced an idiom often used to make sure that the $ alias refers to the `jQuery` name in a localized manner without affecting the remainder of the page, and this little trick can also be (and often is) employed when defining jQuery plugins as follows:

```
(function($){
//
// Plugin definition goes here
//
})(jQuery);
```

By passing `jQuery` to a function that defines the parameter as $, $ is guaranteed to reference `jQuery` within the body of the function.

We can now happily use $ to our heart's content in the definition of the plugin.

Before we dive into learning how to add new elements to jQuery, let's look at one more technique plugin authors are encouraged to use.

7.2.3 Taming complex parameter lists

Most plugins tend to be simple affairs that require few, if any, parameters. We've seen ample evidence of this in the vast majority of the core jQuery methods and functions, which either take a small handful of parameters or none at all. Intelligent defaults are supplied when optional parameters are omitted, and parameter order can even take on a different meaning when some optional parameters are omitted.

The `bind()` method is a good example; if the optional data parameter is omitted, the listener function, which is normally specified as the third parameter, can be supplied as the second. The dynamic and interpretive nature of JavaScript allows us to write such flexible code, but this sort of thing can start to break down and get complex (both for page authors and ourselves as the plugin authors) as the number of parameters grows larger. The possibility of a breakdown increases when many of the parameters are optional.

Consider a somewhat complex function whose signature is as follows:

```
function complex(p1,p2,p3,p4,p5,p6,p7) {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

This function accepts seven arguments, and let's say that all but the first are optional. There are too many optional arguments to make any intelligent guesses about the intention of the caller when optional parameters are omitted. If a caller of this function is only omitting trailing parameters, this isn't much of a problem because the optional trailing arguments can be detected as nulls. But what if the caller wants to specify p7 but let p2 through p6 default? Callers would need to use placeholders for any omitted parameters and write

```
complex(valueA,null,null,null,null,null,valueB);
```

Yuck! Even worse is a call such as

```
complex(valueA,null,valueC,valueD,null,null,valueB);
```

along with other variations of this nature. Page authors using this function are forced to carefully keep track of counting nulls and the order of the parameters; plus, the code is difficult to read and understand.

But short of not allowing so many options to the caller, what can we do?

Again, the flexible nature of JavaScript comes to the rescue; a pattern that allows us to tame this chaos has arisen among the page-authoring communities—the *options hash*.

Using this pattern, optional parameters are gathered into a *single* parameter in the guise of a JavaScript `Object` instance whose property name/value pairs serve as the optional parameters.

Using this technique, our first example could be written as

```
complex(valueA, {p7: valueB});
```

And the second as the following:

```
complex(valueA, {
  p3: valueC,
  p4: valueD,
  p7: valueB
});
```

Much better!

We don't have to account for omitted parameters with placeholder nulls, and we also don't need to count parameters; each optional parameter is conveniently labeled so that it's clear to see exactly what it represents (when we use better parameter names than p1 through p7, that is).

Although this is obviously a great advantage to the caller of our complex functions, what about the ramifications for *us* as the function authors? As it turns out, we've already seen a jQuery-supplied mechanism that makes it easy for us to gather these optional parameters together and to merge them with default values. Let's reconsider our complex example function with a required parameter and six options. The new, simplified signature is

```
complex(p1,options)
```

Within our function, we can merge those options with default values with the handy `$.extend()` utility function. Consider the following:

```
function complex(p1,options) {
  var settings = $.extend({
    option1: defaultValue1,
    option2: defaultValue2,
    option3: defaultValue3,
    option4: defaultValue4,
    option5: defaultValue5,
    option6: defaultValue6
  },options||{});
  // remainder of function...
}
```

By merging the values passed to us by the page author in the `options` parameter with an object containing all the available options with their default values, the `settings` variable ends up with all possible option default values superseded by any explicit values specified by the page author.

Note that we guard against an `options` object that's `null` or undefined with `||{}`, which supplies an empty object if `options` evaluates to `false` (as we know `null` and `undefined` do).

Easy, versatile, and caller-friendly!

We'll see examples of this pattern in use later in this chapter and in jQuery functions that will be introduced in chapter 8, but for now, let's finally look at how we extend jQuery with our own utility functions.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

7.3 Writing custom utility functions

In this book, we use the term *utility function* to describe functions defined as properties of `jQuery` (and therefore $). These functions are not intended to operate on DOM elements—that's the job of methods defined to operate on a jQuery wrapped set—but to either operate on non-element JavaScript objects or perform some other non-object-specific operation. Some examples we've seen of these types of function are `$.each()` and `$.noConflict()`.

In this section, we'll learn how to add our own similar functions.

Adding a function as a property to an `Object` instance is as easy as declaring the function and assigning it to an object property. (If this seems like black magic to you and you have not yet read through the appendix, now would be a good time to do so.) Creating a trivial custom utility function should be as easy as

```
$.say = function(what) { alert('I say '+what); }
```

And in truth, it *is* that easy. But this manner of defining a utility function isn't without its pitfalls; remember our discussion in section 7.2.2 regarding the $? What if some page author is including this function on a page that uses Prototype and has called `$.noConflict()`? Rather than add a jQuery extension, we'd create a method on Prototype's `$()` function. (Get thee to the appendix if the concept of a *method* of a function makes your head hurt.)

This isn't a problem for a private function that we know will never be shared, but even then, what if some future changes to the pages usurp the $? It's a good idea to err on the side of caution.

One way to ensure that someone stomping on $ doesn't also stomp on us is not using the $ at all. We could write our trivial function as

```
jQuery.say = function(what) { alert('I say '+what); }
```

This seems like an easy way out but proves to be less than optimal for more complex functions. What if the function body utilizes lots of jQuery methods and functions internally to get its job done? We'd need to use `jQuery` rather than $ throughout the function. That's rather wordy and inelegant; besides, once we use the $, we don't want to let it go!

So looking back to the idiom we introduced in section 7.2.2, we can safely write our function as follows:

```
(function($){
  $.say = function(what) { alert('I say '+what); }
})(jQuery);
```

We highly encourage using this pattern (even though it may seem like overkill for such a trivial function) because it protects the use of $ when declaring and defining the function. Should the function ever need to become more complex, we can extend and modify it without wondering whether it's safe to use the $ or not.

With this pattern fresh in our minds, let's implement a non-trivial utility function of our own.

7.3.1 Creating a data manipulation utility function

Often, when emitting fixed-width output, it's necessary to take a numeric value and format it to fit into a fixed-width field (where *width* is defined as number of characters). Usually such operations will right-justify the value within the fixed-width field and prefix the value with enough *fill characters* to make up any difference between the length of the value and the length of the field.

Let's write such a utility function that's defined with the following syntax:

Function syntax: $.toFixedWidth

`$.toFixedWidth(value,length,fill)`

Formats the passed value as a fixed-width field of the specified length. An optional fill character can be supplied. If the numeric value exceeds the specified length, its higher order digits will be truncated to fit the length.

Parameters

`value`	(Number) The value to be formatted.
`length`	(Number) The length of the resulting field.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`fill` (String) The fill character used when front-padding the value. If omitted, `0` is used.

Returns

The fixed-width field.

The implementation of this function is shown in listing 7.1.

Listing 7.1 Implementation of the `$.toFixedWidth()` custom utility function

```
(function($){
 $.toFixedWidth = function(value,length,fill) {
   var result = value.toString();
   if (fill == null) fill = '0';            #1
   var padding = length - result.length;    #2
   if (padding < 0) {
     result = result.substr(-padding);      #3
   }
   else {
     for (var n = 0; n < padding; n++)
       result = fill + result;              #4
   }
   return result;                           #5
 };
})(jQuery);
```

#1 Assigns default value
#2 Computes padding
#3 Truncates if necessary
#4 Pads result
#5 Returns final result

Cueballs in code and text

This function is simple and straightforward. The passed value is converted to its string equivalent, and the fill character is determined either from the passed value or the default of `0` (#1). Then, we compute the amount of padding needed (#2).

If we end up with negative padding (the result is longer than the passed field length), we truncate from the beginning of the result to end up with the specified length (#3); otherwise, we pad the beginning of the result with the appropriate number of fill characters (#4) prior to returning it as the result of the function (#5).

NOTE

If you want to make sure that you utility functions aren't going to conflict with anybody else's, you can namespace the functions by creating a namespace object on `$` that in turn serves as the owner of your functions. For example, if we wanted to namespace all our date formatter functions under a namespace called jQiaDateFormatter, we would do the following:

```
$.jQiaDateFormatter = {};
$.jQiaDateFormatter.toFixedWidth = function(value,length,fill) {
```

This ensure that functions like `toFixedWidth()` can never conflict with another similarly named function. (Of course, we still need to worry about conflicting namespaces, but that's easier to deal with.)

Simple stuff, but it serves to show how easily we can add a utility function. And, as always, there's room for improvement. Consider the following exercises:

1. As with most examples in books, the error checking is minimal to focus on the lesson at hand. How would you beef up the function to account for caller errors such as not passing numeric values for `value` and `length`? What if they don't pass them at all?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

2. We were careful to truncate numeric values that were too long in order to guarantee that the result was always the specified length. But, if the caller passes more than a single-character string for the fill character, all bets are off. How would you handle that?

3. What if you don't want to truncate too-long values?

Now, let's tackle a more complex function in which we can make use of the `$.toFixedWidth()` function that we just wrote.

7.3.2 Writing a date formatter

If you've come to the world of client-side programming from the server, one of the things you may have longed for is a simple date formatter; something that the JavaScript `Date` type doesn't provide. Because such a function would operate on a `Date` instance, rather than any DOM element, it's a perfect candidate for a utility function. Let's write one that uses the following syntax:

Function syntax: $.formatDate

`$.formatDate(date,pattern)`

Formats the passed date according to the supplied pattern. The tokens that are substituted in the pattern are as follows:

yyyy: the 4-digit year
yy: the 2-digit year
MMMM: the full name of the month
MMM: the abbreviated name of the month
MM: the month number as a 0-filled, 2-character field
M: the month number
dd: the day in the month as a 0-filled, 2-character field
d: the day in the month
EEEE: the full name of the day of the week
EEE: the abbreviated name of the day of the week
a: the meridium (AM or PM)
HH: the 24-hour clock hour in the day as a 2-character, 0-filled field
H: the 24-hour clock hour in the day
hh: the 12-hour clock hour in the day as a 2-character, 0-filled field
h: the 12-hour clock hour in the day
mm: the minutes in the hour as a 2-character, 0-filled field
m: the minutes in the hour
ss: the seconds in the minute as a 2-character, 0-filled field
s: the seconds in the minute
S: the milliseconds in the second as a 3-character, 0-filled field

Parameters

`date`	(Date) The date to be formatted.
`pattern`	(String) The pattern to format the date into. Any characters not matching pattern tokens are copied as-is to the result.

Returns

The formatted date.

The implementation of this function is shown in listing 7.2. We're not going to go into great detail regarding the algorithm used to perform the formatting (after all, this isn't an algorithms book), but we're going to use this implementation to point out some interesting tactics that we can use when creating a somewhat complex utility function.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Listing 7.2 Implementation of the $.formatDate custom utility function

```
(function($){
 $.formatDate = function(date,pattern) {                        #1
   var result = [];
   while (pattern.length > 0) {
     $.formatDate.patternParts.lastIndex = 0;
     var matched = $.formatDate.patternParts.exec(pattern);
     if (matched) {
       result.push(
          $.formatDate.patternValue[matched[0]].call(this,date)
       );
       pattern = pattern.slice(matched[0].length);
     } else {
       result.push(pattern.charAt(0));
       pattern = pattern.slice(1);
     }
   }
   return result.join('');
 };
 $.formatDate.patternParts =                                     #2
   /^(yy(yy)?|M(M(M(M)?)?)?|d(d)?|EEE(E)?|a|H(H)?|h(h)?|m(m)?|s(s)?|S)/;

 $.formatDate.monthNames = [                                     #3
   'January','February','March','April','May','June','July',
   'August','September','October','November','December'
 ];
 $.formatDate.dayNames = [                                       #4
   'Sunday','Monday','Tuesday','Wednesday','Thursday','Friday',
   'Saturday'
 ];
 $.formatDate.patternValue = {                                   #5
   yy: function(date) {
     return $.toFixedWidth(date.getFullYear(),2);
   },
   yyyy: function(date) {
     return date.getFullYear().toString();
   },
   MMMM: function(date) {
     return $.formatDate.monthNames[date.getMonth()];
   },
   MMM: function(date) {
     return $.formatDate.monthNames[date.getMonth()].substr(0,3);
   },
   MM: function(date) {
     return $.toFixedWidth(date.getMonth() + 1,2);
   },
   M: function(date) {
     return date.getMonth()+1;
   },
   dd: function(date) {
     return $.toFixedWidth(date.getDate(),2);
   },
   d: function(date) {
     return date.getDate();
   },
   EEEE: function(date) {
     return $.formatDate.dayNames[date.getDay()];
   },
   EEE: function(date) {
     return $.formatDate.dayNames[date.getDay()].substr(0,3);
   },
   HH: function(date) {
     return $.toFixedWidth(date.getHours(),2);
   },
   H: function(date) {
     return date.getHours();
   },
   hh: function(date) {
     var hours = date.getHours();
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
      return $.toFixedWidth(hours > 12 ? hours - 12 : hours,2);
    },
    h: function(date) {
      return date.getHours() % 12;
    },
    mm: function(date) {
      return $.toFixedWidth(date.getMinutes(),2);
    },
    m: function(date) {
      return date.getMinutes();
    },
    ss: function(date) {
      return $.toFixedWidth(date.getSeconds(),2);
    },
    s: function(date) {
      return date.getSeconds();
    },
    S: function(date) {
      return $.toFixedWidth(date.getMilliseconds(),3);
    },
    a: function(date) {
      return date.getHours() < 12 ? 'AM' : 'PM';
    }
  };
})(jQuery);
```

#1 Implements the main body of the function
#2 Defines the regular expression
#3 Provides the name of the months
#4 Provides the name of the days
#5 Collects token-to-value translation functions

Cueballs in code and text

The most interesting aspect of this implementation, aside from a few JavaScript tricks used to keep the amount of code in check, is that the function (#1) needs some ancillary data to do its job—in particular:

- A regular expression used to match tokens in the pattern (#2)
- A list of the English names of the months (#3)
- A list of the English names of the days (#4)
- A set of sub-functions designed to provided the value for each token type given a source date (#5)

We could have included each of these as `var` definitions within the function body, but that would clutter an already somewhat involved algorithm; and because they're constants, it makes sense to segregate them from variable data.

We don't want to pollute the global namespace, or even the `$` namespace, with a bunch of names needed only by this function, so we make these declarations properties of our new function itself. Remember, JavaScript functions are first-class objects, and they can have their own properties like any other Java-Script object.

As for the formatting algorithm itself? In a nutshell, it operates as follows:

1. Creates an array to hold portions of the result.
2. Iterates over the pattern, consuming identified token and non-token characters until it has been completely inspected.
3. Resets the regular expression (stored in `$.formatDate.patternParts`) on each iteration by setting its `lastIndex` property to 0.
4. Tests the regular expression for a token match against the current beginning of the pattern.
5. Calls the function in the `$.formatDate.patternValue` collection of conversion functions to obtain the appropriate value from the `Date` instance if a match occurs. This value is pushed onto the end of the results array, and the matched token is removed from the beginning of the pattern.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

6. Removes the first character from the pattern and adds it to the end of the results array if a token isn't matched at the current beginning of the pattern.

7. Joins the results array into a string and returns it as the value of the function when the entire pattern has been consumed.

Note that the conversion functions in the `$.formatDate.patternValue` collection make use of the `$.toFixedWidth()` function that we created in the previous section.

You'll find both of these functions in the file `chapter7/jquery.jqia.dateFormat.js` and a rudimentary page to test it at `chapter7/test.dateFormat.html`.

Operating on run-of-the-mill JavaScript objects is all well and good, but the real power of jQuery lies in the wrapper methods that operate on a set of DOM elements collected via the power of jQuery selectors. Next, let's see how we can add our own powerful wrapper methods.

7.4 Adding new wrapper methods

The true power of jQuery lies in the ability to easily and quickly select and operate on DOM elements. Luckily, we can extend that power by adding wrapper methods of our own that manipulate selected DOM elements as we deem appropriate. By adding wrapper methods, we automatically gain the use of the powerful jQuery selectors to pick and choose which elements are to be operated on without having to do all the work ourselves.

Given what we know about JavaScript, we probably could have figured out on our own how to add utility functions to the $ namespace, but that's not necessarily true of wrapper functions. There's a tidbit of jQuery-specific information that we need to know; to add wrapper methods to jQuery, we must assign them as properties to an object named `fn` in the $ namespace.

The general pattern for creating a wrapper functions is

```
$.fn.wrapperFunctionName = function(params){function-body};
```

Let's concoct a trivial wrapper method to set the color of the matched DOM elements to blue.

```
(function($){
  $.fn.makeItBlue = function() {
    return this.css('color','blue');
  }
})(jQuery);
```

As with utility functions, we make the declaration within an outer function that guarantees that $ is an alias to `jQuery`. But unlike utility functions, we create the new wrapper method as a property of `$.fn` rather than of $.

NOTE

If you're familiar with "object-oriented" JavaScript and its prototype-based class declarations, you might be interested to know that `$.fn` is merely an alias for an internal `prototype` property of an object that jQuery uses to create its wrapper objects.

Within the body of the method, the function context (`this`) refers to the wrapped set. We can use all of the predefined jQuery methods on it; as in this example, we call the `css()` method on the wrapped set to set the `color` to `blue` for all matched DOM elements.

WARNING

The function context (`this`) within the main body of a wrapper method refers to the wrapped set, but when inline functions are declared within this function, they each have their own function contexts. You must take care when using `this` under such circumstances to make sure that it's referring to what you think it is! For example, if you use `each()` with its iterator function, `this` within the iterator function references the DOM element for the current iteration.

We can do almost anything we like to the DOM elements in the wrapped set, but there is one *very* important rule when defining new wrapper methods; unless the function is intended to return a specific value, it should always return the wrapped set as its return value. This allows our new method to take part in any jQuery method chains. In our example, because the `css()` method returns the wrapped set, we simply return the result of the call to `css()`.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

In this example, we apply the jQuery `css()` method to all the elements in the wrapped set by applying it to `this`. If, for some reason, we need to deal with each wrapped element individually (perhaps because we need to make conditional processing decisions), the following pattern can be used:

```
(function($){
  $.fn.someNewMethod = function() {
    return this.each(function(){
      //
      // Function body goes here -- this refers to individual
      // elements
      //
    });
  }
})(jQuery);
```

In this pattern, the `each()` method is used to iterate over every individual element in the wrapped set. Note that, within the iterator function, `this` refers to the current DOM element rather than the entire wrapped set. The wrapped set returned by `each()` is returned as the new method's value so that this method can participate in chaining.

Let's consider a variation of our previous blue-centric example that deals with each element individually:

```
(function($){
  $.fn.makeItBlueOrRed = function() {
    return this.each(function(){
      this.css('color',$(this).is('[id]') ? 'blue' : 'red');
    });
  }
})(jQuery);
```

In this variation, we want to apply the color "blue" or the color "red" based upon a condition that's unique to each element (in this case, whether it has an `id` attribute or not), so we iterate over the wrapped set so that we can examine and manipulate each element individually.

That's all there is to it, but (isn't there always a *but*?) there are some techniques we should be aware of when creating more involved jQuery wrapper methods. Let's define a couple more plugin methods of greater complexity to examine those techniques.

7.4.1 Applying multiple operations in a wrapper method

Let's develop another new plugin method that performs more than a single operation on the wrapped set. Imagine that we need to be able to flip the read-only status of text fields within a form and to simultaneously and consistently affect the appearance of the field. We could easily chain a couple of existing jQuery methods together to do this, but we want to be neat and tidy about it and bundle these operations together into a single method.

We'll name our new method `setReadOnly()`, and its syntax is as follows:

Method syntax: setReadOnly

`setReadOnly(state)`

Sets the read-only status of wrapped text fields to the state specified by `state`. The opacity of the fields will be adjusted: 100% if not read-only or 50% if read-only. Any elements in the wrapped set other than text fields are ignored.

Parameters

`state`	(Boolean) The read-only state to set. If `true`, the text fields are made read-only; otherwise, the read-only status is cleared.

Returns

The wrapped set.

The implementation of this plugin is shown in listing 7.3 and can be found in the file chapter7/jquery.jqia.setreadonly.js.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Listing 7.3 Implementation of the setReadOnly() custom wrapper method

```
(function($){
  $.fn.setReadOnly = function(readonly) {
    return this.filter('input:text')
      .attr('readonly',readonly)
      .css('opacity', readonly ? 0.5 : 1.0)
      .end();
  };
})(jQuery);
```

This example is only slightly more complicated than our initial example, but exhibits the following key concepts:

- A parameter is passed that affects how the method operates.
- Three jQuery methods are applied to the wrapped set by use of jQuery chaining.
- The new method can participate in a jQuery chain because it returns the wrapped set as its value.
- The `filter()` method is used to ensure that, no matter what set of wrapped elements the page author applied this method to, only text fields are affected.
- The `end()` method is invoked so that the original (not the filtered) wrapped set is returned as the value of the call.

How might we put this method to use?

Often, when defining an online order form, we may need to allow the user to enter two sets of address information: one for where the order is to be shipped and one for the billing information. Much more often than not, these two addresses are going to be the same; making the user enter the same information twice decreases our user-friendliness factor to less than we'd want it to be.

We could write our server-side code to assume that the billing address is the same as the shipping address if the form is left blank, but let's assume that our product manager is a bit paranoid and would like something more overt on the part of the user.

We'll satisfy him by adding a check box to the billing address that indicates whether the billing address is the same as the shipping address. When this box is checked, the billing address fields will be copied from the shipping fields and then made read-only. Unchecking the box will clear the value and read-only status from the fields.

Figure 7.1a shows a test form in its before state, and figure 7.1b in its after state.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

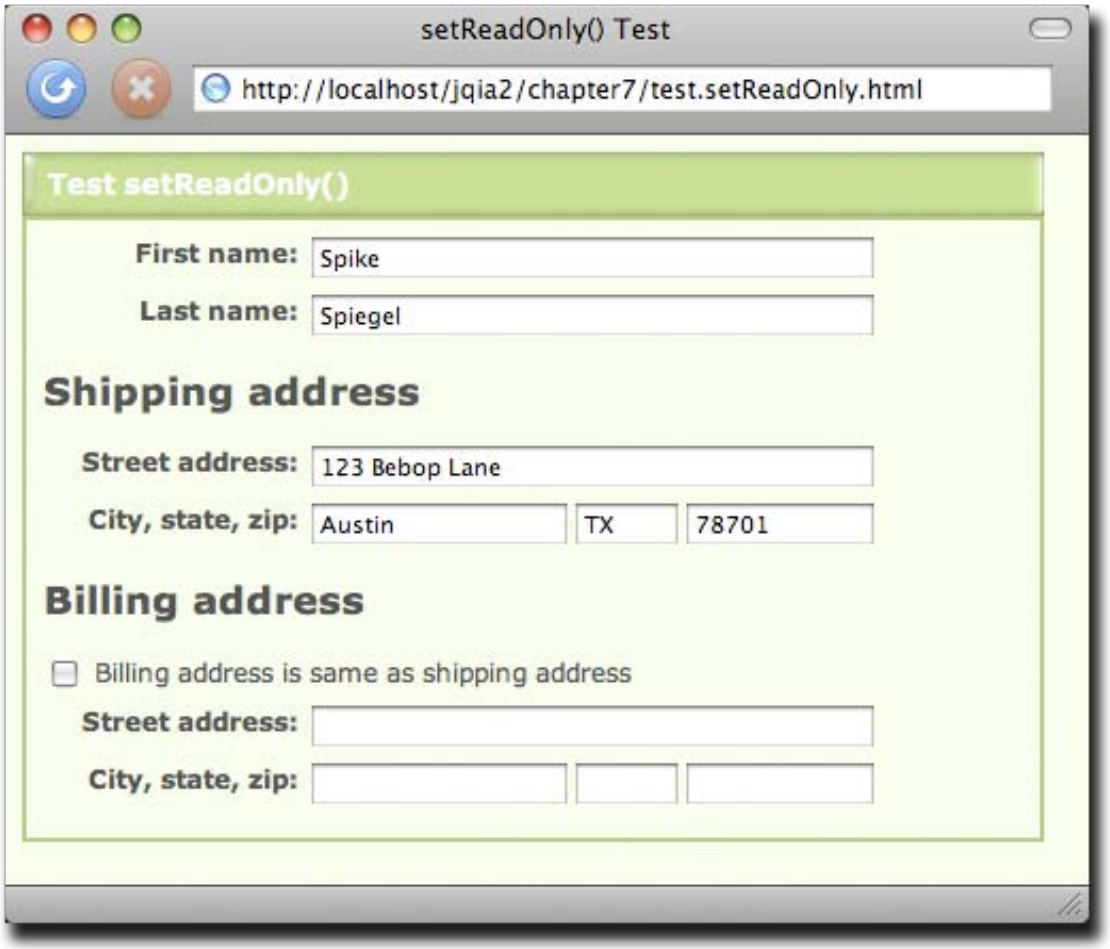


Figure 7.1a Our form for testing the setReadOnly() custom wrapper method before checking the checkbox

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

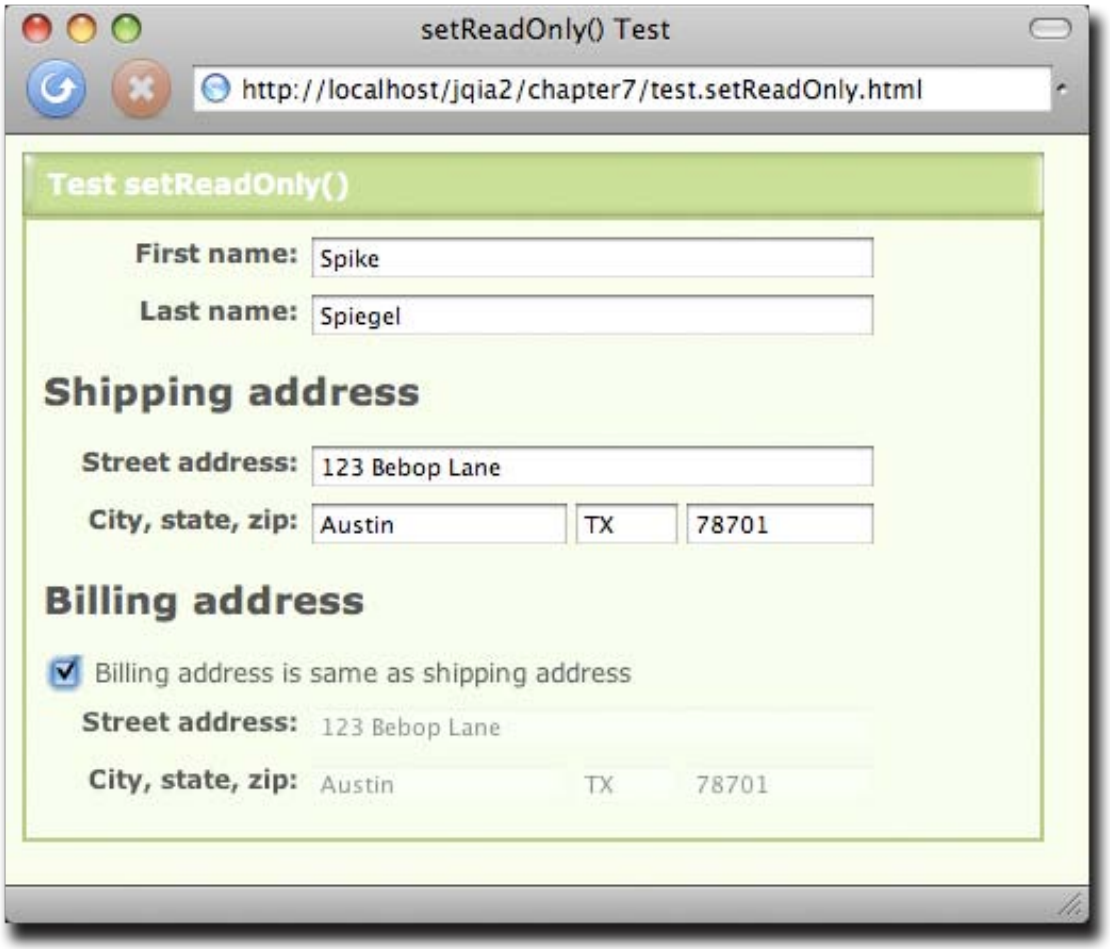


Figure 7.1b Our form for testing the setReadOnly() custom wrapper method before checking the checkbox – showing the results of applying the custom method.

The page for this test form is available in the file `chapter7/test.setReadOnly.html` and is shown in listing 7.4.

Listing 7.4 Implementation of the testpage for the custom setReadOnly() wrapper mathod

```
<!DOCTYPE html>
<html>
  <head>
    <title>setReadOnly() Test</title>
    <link rel="stylesheet" type="text/css" href="../styles/core.css">
    <link rel="stylesheet" type="text/css" href="test.setReadOnly.css">
    <script type="text/javascript" src="../scripts/jquery-1.3.2.min.js"></script>
    <script type="text/javascript" src="jquery.jqia.setReadOnly.js"></script>
    <script type="text/javascript">
      $(function(){
        $('#sameAddressControl').click(function(){
          var same = this.checked;
          $('#billAddress').val(same ? $('#shipAddress').val():'');
          $('#billCity').val(same ? $('#shipCity').val():'');
          $('#billState').val(same ? $('#shipState').val():'');
          $('#billZip').val(same ? $('#shipZip').val():'');
          $('#billingAddress input').setReadOnly(same);
        });
      });
    </script>
  </head>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
  <body>
    <div class="module">
      <div class="banner">
        <img src="../images/module.left.cap.png" alt="" style="float:left"/>
        <img src="../images/module.right.cap.png" alt="" style="float:right"/>
        <h2>Test setReadOnly()</h2>
      </div>
      <div class="body">

        <form name="testForm">
          <div>
            <label>First name:</label>
            <input type="text" name="firstName" id="firstName"/>
          </div>
          <div>
            <label>Last name:</label>
            <input type="text" name="lastName" id="lastName"/>
          </div>
          <div id="shippingAddress">
            <h2>Shipping address</h2>
            <div>
              <label>Street address:</label>
              <input type="text" name="shipAddress" id="shipAddress"/>
            </div>
            <div>
              <label>City, state, zip:</label>
              <input type="text" name="shipCity" id="shipCity"/>
              <input type="text" name="shipState" id="shipState"/>
              <input type="text" name="shipZip" id="shipZip"/>
            </div>
          </div>
          <div id="billingAddress">
            <h2>Billing address</h2>
            <div>
              <input type="checkbox" id="sameAddressControl"/>
              Billing address is same as shipping address
            </div>
            <div>
              <label>Street address:</label>
              <input type="text" name="billAddress"
                     id="billAddress"/>
            </div>
            <div>
              <label>City, state, zip:</label>
              <input type="text" name="billCity" id="billCity"/>
              <input type="text" name="billState" id="billState"/>
              <input type="text" name="billZip" id="billZip"/>
            </div>
          </div>
        </form>
      </div>
    </div>

  </body>
</html>
```

We won't belabor the operation of this page, as it's relatively straightforward. The only truly interesting aspect of this page is the click handler attached to the check box in the ready handler. When the state of the check box is changed by a click we:

1. Copy the checked state into variable same for easy reference in the remainder of the listener.
2. Set the values of the billing address fields. If they are to be the same, we set the values from the corresponding fields in the shipping address information. If not, we clear the fields.
3. Call the new `setReadOnly()` method on all input fields in the billing address container.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

But, oops! We were a little sloppy with that last step. The wrapped set that we create with `$('#billingAddress input')` contains not only the text fields in the billing address block but the check box too. The check box element doesn't have read-only semantics, but it can have its opacity changed—definitely not our intention!

Luckily, this sloppiness is countered by the fact that we were *not* sloppy when defining our plugin. Recall that we filtered out all but text fields before applying the remainder of the methods in that method. We highly recommend such attention to detail, particularly for plugins that are intended for public consumption.

What are some ways that this method could be improved? Consider making the following changes:

- We forgot about text areas! How would you modify the code to include text areas along with the text fields?
- The opacity levels applied to the fields in either state are hard-coded into the function. This is hardly caller-friendly. Modify the method to allow the levels to be caller-supplied.
- Oh heck, why force the page author to accept the ability to affect only the opacity? How would you modify the method to allow the page author to determine what the renditions for the fields should be in either state?

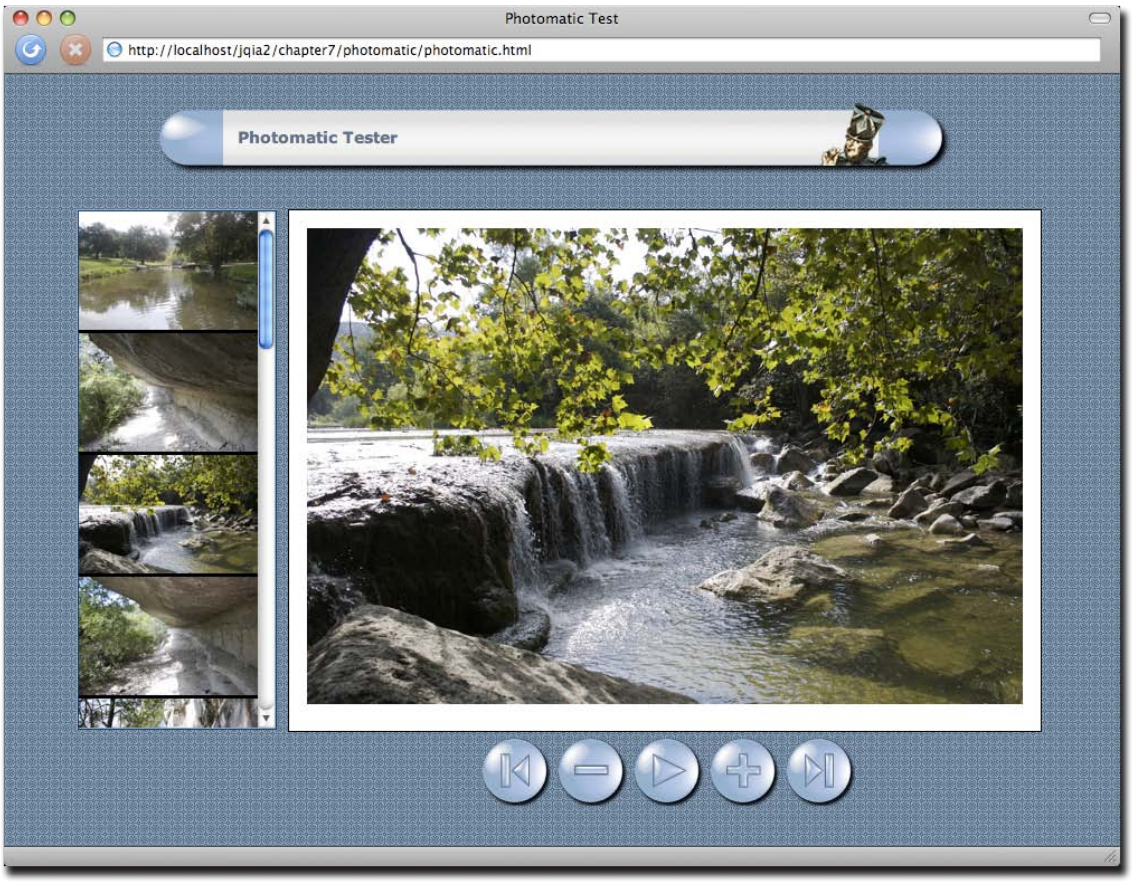
Now let's take on an even more complex plugin.

7.4.2 Retaining state within a wrapper method

Everybody loves a slideshow!

At least on the web, that is. Unlike hapless after-dinner guests forced to sit through a mind-numbingly endless display of badly focused vacation photos, visitors to a web slideshow can leave whenever they like without hurting anyone's feelings!

For our more complex plugin example, we're going to develop a jQuery method that will easily allow a page author to whip up a quick slideshow page. We'll create a jQuery plugin, which we'll name the *Photomatic*, and then we'll whip up a test page to put it through its paces. When complete, this test page will appear as shown in figure 7.2.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Figure 7.2 The test page that we'll use to put our Photomatic plugin through its paces

This page sports the following components:

- A set of thumbnail images
- A full-sized photo of one of the images available in the thumbnail list
- A set of buttons to move through the slideshow manually, or start/stop the automatic slide show

The behaviors of the page are as follows:

- Clicking any thumbnail displays the corresponding full-sized image.
- Clicking the full-sized image displays the next image.
- Clicking any button performs the following operations:
    - First—Displays the first image
    - Previous—Displays the previous image
    - Next—Displays the next image
    - Last—Displays the last image
    - Play—Commences moving through the photos automatically until clicked again
- Any operation that moves off the end of the list of images wraps back to the other end; clicking Next while on the last image displays the first image and vice versa.

We also want to grant the page authors as much freedom for layout and styling as possible; we'll define our plugin so that the page authors can set up the elements in any manner that they would like and then tell us which page element should be used for each purpose. Furthermore, in order to give the page authors as much leeway as possible, we'll define our plugin so that the authors can provide any wrapped set of images to serve as thumbnails. Usually, thumbnails will be gathered together as in our test page, but page authors are free to identify any image on the page as a thumbnail.

To start, let's introduce the syntax for the Photomatic Plugin.

Method syntax: photomatic

`photomatic(options)`

Instruments the wrapped set of thumbnails, as well as page elements identified in the `settings` hash, to operate as Photomatic controls.

Parameters

`Options`	(Object) An object hash that specifies the options for the Photomatic. See table 7.1 for details.

Returns

The wrapped set.

Because we have a non-trivial number of parameters to control the operation of the Photomatic (many of which can be omitted), we utilize an object hash to pass them as outlined in section 7.2.3. The possible settings that we can specify are shown in table 7.1.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Table 7.1 The options for the Photomatic custom plugin method

Setting name	Description
`firstControl`	(String|Element) Either a reference to or jQuery selector that identifies the DOM element(s) to serve as a *first* control. If omitted, no control is instrumented.
`lastControl`	(String|Element) Either a reference to or jQuery selector that identifies the DOM element(s) to serve as a *last* control. If omitted, no control is instrumented.
`nextControl`	(String|Element) Either a reference to or jQuery selector that identifies the DOM element(s) to serve as a *next* control. If omitted, no control is instrumented.
`photoElement`	(String|Element) Either a reference to or jQuery selector that identifies the `<img>` element that's to serve as the full-sized photo display. If omitted, defaults to the jQuery selector `'#photomaticPhoto'`.
`playControl`	(String|Element) Either a reference to or jQuery selector that identifies the DOM element(s) to serve as a *play* control. If omitted, no control is instrumented.
`previousControl`	(String| Element) Either a reference to or jQuery selector that identifies the DOM element(s) to serve as a *previous* control. If omitted, no control is instrumented.
`transformer`	(Function) A function used to transform the URL of a thumbnail image into the URL of its corresponding full-sized photo image. If omitted, the default transformation substitutes all instances of *thumbnail* with *photo* in the URL.

With a nod to the notion of *test-driven development*, let's create the test page for this plugin *before* we dive into creating the Photomatic Plugin itself. The code for this page, available in the file `chapter7/photomatic/photomatic.html`, is shown in listing 7.5.

Listing 7,5 The test page that creates the Photomatic display of figure 7.2

```
<!DOCTYPE html>
<html>
  <head>
    <title>Photomatic Test</title>
    <link rel="stylesheet" type="text/css"
          href="../../styles/core.css">
    <link rel="stylesheet" type="text/css" href="photomatic.css">
    <script type="text/javascript"
            src="../../scripts/jquery-1.3.2.min.js"></script>
    <script type="text/javascript"
            src="jquery.jqia.photomatic.js"></script>
    <script type="text/javascript">
      $(function(){
        $('#thumbnailsPane img').photomatic({                                    #1
          photoElement: '#photoDisplay',
          previousControl: '#previousButton',
          nextControl: '#nextButton',
          firstControl: '#firstButton',
          lastControl: '#lastButton',
          playControl: '#playButton',
          delay: 1000
        });
      });
    </script>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
  </head>

  <body class="fancy">

    <div id="pageContainer">
      <div id="pageContent">

        <h1>Photomatic Tester</h1>

        <div id="thumbnailsPane">                                          #2
          <img src="thumbnails/IMG_2212.jpg"/>
          <img src="thumbnails/IMG_2222.jpg"/>
          <img src="thumbnails/IMG_2227.jpg"/>
          <img src="thumbnails/IMG_2235.jpg"/>
          <img src="thumbnails/IMG_2259.jpg"/>
          <img src="thumbnails/IMG_2269.jpg"/>
          <img src="thumbnails/IMG_2273.jpg"/>
          <img src="thumbnails/IMG_2287.jpg"/>
          <img src="thumbnails/IMG_2292.jpg"/>
          <img src="thumbnails/IMG_2296.jpg"/>
          <img src="thumbnails/IMG_2298.jpg"/>
          <img src="thumbnails/IMG_2302.jpg"/>
          <img src="thumbnails/IMG_2310.jpg"/>
          <img src="thumbnails/IMG_2319.jpg"/>
          <img src="thumbnails/IMG_2331.jpg"/>
          <img src="thumbnails/IMG_2335.jpg"/>
        </div>

        <div id="photoPane">
          <img id="photoDisplay" src=""/>                              #3
        </div>

        <div id="buttonBar">                                               #4
          <img src="button.placeholder.png" id="firstButton"
               alt="First" title="First photo"/>
          <img src="button.placeholder.png" id="previousButton"
               alt="Previous" title="Previous photo"/>
          <img src="button.placeholder.png" id="playButton"
               alt="Play/Pause" title="Play or pause slideshow"/>
          <img src="button.placeholder.png" id="nextButton"
               alt="Next" title="Next photo"/>
          <img src="button.placeholder.png" id="lastButton"
               alt="Last" title="Last photo"/>
        </div>

      </div>
    </div>

  </body>
</html>
```

#1 Invokes the Photomatic Plugin
#2 Contains thumbnail images
#3 Defines the image element for full-sized photos
#4 Contains elements to serve as controls

Cueballs in code and text

If that looks simpler that you thought it would, you shouldn't be surprised at this point. By applying the principles of Unobtrusive JavaScript and by keeping all style information in an external style sheet, our markup is tidy and simple. In fact, even the on-page script has a tiny footprint, consisting of a single statement that invokes our plugin (#1).

The HTML markup consists of a container that holds the thumbnail images (#2), an image element (initially source-less) to hold the full-sized photo (#3), and a collection of images (#4) that will control the slideshow. Everything else is handled by our new plugin.

Let's develop that now.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

To start, let's set out a skeleton (we'll fill in the details as we go along). Our starting point should look rather familiar by now because it follows the same patterns we've been using so far.

```
(function($){
  $.fn.photomatic = function(options) {
  };
})(jQuery);
```

This defines our initially empty wrapper function, which (as expected from our syntax description) accepts a single hash parameter named `options`. First, within the body of the function, we merge these caller settings with the default settings as described by table 7.1. This will give us a single "settings" object that we can refer to throughout the remainder of the method.

We perform this merge operation using the following idiom (that we've already seen a few times):

```
var settings = $.extend({
      photoElement: 'img.photomaticPhoto',
      transformer: function(name) {
                     return name.replace(/thumbnail/,'photo');
                   },
      nextControl: null,
      previousControl: null,
      firstControl: null,
      lastControl: null,
      playControl: null,
      delay: 3000
    },options||{});
```

After the execution of this statement, the `settings` variable will contain the values supplied by the caller, with defaults supplied by the inline hash object. While it's not necessary to include the properties that have no defaults (those with a `null` value), we find it's useful and wise to list all the possible options here if for nothing other than documentation purposes.

We're also going to need to keep track of a few things. In order to know what concepts like *next* image and *previous* image mean, we need not only a list of the thumbnail images but also an indicator that identifies the *current* image being displayed.

The list of thumbnail images is the wrapped set that this method is operating on—or, at least, it should be. We don't know what the page authors collected in the wrapped set, so we want to filter it down to only image elements; we can easily do this with a jQuery selector. But where should we store the list?

We could easily create another variable to hold it, but there's a lot to be said for keeping things corralled. Let's store the list as another property on `settings` as follows:

```
settings.thumbnails = this.filter('img');
```

Filtering the wrapped set (available via `this` in the method) for only image elements results in a new wrapped set (containing only `<img>` elements) that we store in a property of `settings` that we name `thumbnails`.

Another piece of state that we need to keep track of is the *current* image. We'll keep track of that by maintaining an index into the list of thumbnails by adding another property to `settings` named `current` as follows:

```
settings.current = 0;
```

There is one more setup step that we need to take with regard to the thumbnails. If we're going to keep track of which photo is *current* by keeping track of its index, there will be at least one case where, given a reference to a thumbnail element, we'll need to know its index. The easiest way to handle this is to anticipate this need and use the handy jQuery `data()` method to record a thumbnail's index on each of the thumbnail elements. We do that with the following statement:

```
settings.thumbnails.each(
  function(n){ $(this).data('photomatic-index',n); }
);
```

This statement iterates through each of the thumbnail images, adding a data element named "photomatic-index" to it that records its order in the list. Now that our initial state is set up, we're ready to move on to the meat of the plugin – instrumenting the controls, thumbnails and photo display.

Wait a minute! Initial *state*? How can we expect to keep track of state in a *local* variable within a function that's about to finish executing? Won't the variable and all our settings go out of scope when the function returns?

In general that might be true, but there is one case where such a variable sticks around for longer than its usual scope—when it's part of a *closure*. We've seen closures before, but if you're still shaky on them, please review the appendix. You must understand closures not only for completing the implementation of the Photomatic Plugin but also when creating anything but the most trivial of plugins.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

When we think about the job remaining, we realize that we need to attach a number of event listeners to the controls and elements that we've taken such great pains to identify to this point. And because the `settings` variable is in scope when we declare the functions that represent those listeners, each listener will be part of a closure that includes the `settings` variable. So we can rest assured that, even though `settings` may appear to be transient, the state that it represents will stick around and be available to all the listeners that we define.

Speaking of those listeners, here's the list of `click` event listeners that we need to attach to the various elements:

- Clicking a thumbnail photo will cause its full-sized version to be displayed.
- Clicking the full-sized photo will cause the next photo to be displayed.
- Clicking the element defined as the "previous" control will cause the previous image to be displayed.
- Clicking the "next" control will cause the next image to be displayed.
- Clicking the "first" control will cause the first image in the list to be displayed.
- Clicking the "last" control will cause the last image in the list to be displayed.
- Clicking the "play" control will cause the slideshow to automatically proceed, progressing through the photos using a delay specified in the settings. A subsequent click on the control will stop the slideshow.

Looking over this list, we immediately note that all of these listeners have something in common; they all need to cause the full-sized photo of one of the thumbnail images to be displayed. And being the good and clever coders that we are, we want to factor out that common processing into a function so that we don't need to repeat the same code over and over again.

But how?

If we were writing normal on-page JavaScript, we could define a top-level function. If we were writing "object-oriented" JavaScript, we might define a method on a JavaScript object. But we're writing a jQuery plugin; where should we define implementation functions?

We don't want to infringe on either the global namespace, or event the $ namespace, for a function that should only be called internally from our plugin code, so what can we do? Oh, and just to add to our dilemma, let's try to make it so that the function participates in a closure including the settings variable so that we won't have to pass it as a parameter to each invocation.

The power of JavaScript as a *functional* language comes to our aid once again, and allows us to define this new function *within* the plugin function. By doing so, we limit its scope to within the plugin function itself (one of our goals), and because the settings variable is within scope, it forms a closure with the new function (out other goal). What could be simpler?

So we define a function named `showPhoto()`, which accept a single parameter indicating the index of the thumbnail that is to be shown full-sized, within the plugin function as follows:

```
function showPhoto(index) {
  $(settings.photoElement)
    .attr('src',
          settings.transformer(settings.thumbnails[index].src));
  settings.current = index;
};
```

This new function, when passed the `index` of the thumbnail whose full-sized photo is to be displayed, uses the values in the `settings` object (available via the closure created by the function declaration) to do the following:

1. Look up the `src` attribute of the thumbnail identified by `index`
2. Pass that value through the `transformer` function to convert it from a thumbnail URL to a photo URL
3. Assign the result of the transformation to the `src` attribute of the full-sized image element
4. Record the index of the displayed photo as the new current index

With that handy function available, we're ready to define the listeners that we listed earlier. Let's start by instrumenting the thumbnails themselves, which simply need to cause their corresponding full0-size photo to be displayed;

```
settings.thumbnails.click(function(){
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
    showPhoto($(this).data('photomatic-index'));
  });
```

In this handler, we simply obtain the value of the thumbnail's index (which we thoughtfully already stored in the "photomatic-index" data element), and call the `showPhoto()` function using it. The simplicity of this handler verifies that all that setup we coded earlier is going to pay off!

Instrumenting the photo display element to show the "next" photo in the list is just as simple:

```
$(settings.photoElement).click(function(){
  showPhoto((settings.current+1) % settings.thumbnails.length);
});
$(settings.photoElement)
  .attr('title',"Click for next photo")
  .css('cursor','pointer');
```

In this handler, we call the showPhoto() function with the next index value – note how we use the JavaScript modulo operator (%) to wrap around to the front of the list when we fall off the end.

After establishing the handler, we also add a thoughtful `title` attribute to the photo so users know that clicking on the photo will progress to the next one, and set the cursor to indicate that the element is active.

The handlers for the first, previous, next and last controls all follow a similar pattern: figure out what the appropriate index of the thumbnail whose full-sized photo is to be shown, and call `showPhoto()` with that index:

```
$(settings.nextControl).click(function(){
      showPhoto((settings.current+1) % settings.thumbnails.length);
    });
    $(settings.previousControl).click(function(){
      showPhoto((settings.thumbnails.length+settings.current-1) %
                settings.thumbnails.length);
    });
    $(settings.firstControl).click(function(){
      showPhoto(0);
    });
    $(settings.lastControl).click(function(){
      showPhoto(settings.thumbnails.length-1);
    });
```

The instrumentation of the "play" control is somewhat more complicated. Rather than showing a particular photo, this control must start a progression through the entire photo set, and then stop that progression on a subsequent click. Let's take a look at the code we can use to accomplish that:

```
$(settings.playControl).toggle(
      function(event){
        settings.tick = window.setInterval(
          function(){ $(settings.nextControl).click(); },
          settings.delay);
        $(event.target).addClass('photomatic-playing');
        $(settings.nextControl).click();
      },
      function(event){
        window.clearInterval(settings.tick);
        $(event.target).removeClass('photomatic-playing');
      });
```

First, note that we use the jQuery `toggle()` method to easily swap between two different listeners on every other click of the control. That saves us from having to figure out on our own whether we're starting or stopping the slideshow.

In the first handler, we employ the JavaScript-provided `setInterval()` method to cause a function to continually fire off using the `delay` value. We store the handle of that interval timer in the `settings` variable for later reference.

We also add the class `photomatic-playing` to the control so that the page author can effect any appearance changes using CSS, if desired.

As the last act in the handler, we emulate a click on the "next" control to progress to the next photo immediately (rather than having to wait of the first interval to expire).

In the second handler of the `toggle()` invocation, we want to stop the slideshow, so we clear the interval timeout using `clearInterval()`, and remove the `photomatic-playing` class from the control.

Bet you didn't think it would be that easy.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

We have two final tasks before we can declare success; we need to display the first photo in advance of any user action, and we need to return the original wrapped set so that our plugin can participate in jQuery method chains. We achieve these with

```
showPhoto(0);
return this;
```

Take a moment to do a short Victory Dance; we're finally done!

The completed plugin code, which you'll find in the file `chapter7/photomatic/jquery.jqia.photomatic.js`, is shown in listing 7.6.

Listing 7.6 The complete implementation of the Photomatic plugin

```
(function($){

  $.fn.photomatic = function(options) {
var settings = $.extend({
      photoElement: 'img.photomaticPhoto',
      transformer: function(name) {
                     return name.replace(/thumbnail/,'photo');
                   },
      nextControl: null,
      previousControl: null,
      firstControl: null,
      lastControl: null,
      playControl: null,
      delay: 3000
    },options||{});
    function showPhoto(index) {
      $(settings.photoElement)
        .attr('src',
              settings.transformer(settings.thumbnails[index].src));
      settings.current = index;
    };
    settings.current = 0;
    settings.thumbnails = this.filter('img');
    settings.thumbnails.each(
      function(n){ $(this).data('photomatic-index',n); }
    );
    settings.thumbnails.click(function(){
      showPhoto($(this).data('photomatic-index'));
    });
    $(settings.photoElement).click(function(){
      showPhoto((settings.current+1) % settings.thumbnails.length);
    });
    $(settings.photoElement)
      .attr('title',"Click for next photo")
      .css('cursor','pointer');
    $(settings.nextControl).click(function(){
      showPhoto((settings.current+1) % settings.thumbnails.length);
    });
    $(settings.previousControl).click(function(){
      showPhoto((settings.thumbnails.length+settings.current-1) %
                settings.thumbnails.length);
    });
    $(settings.firstControl).click(function(){
      showPhoto(0);
    });
    $(settings.lastControl).click(function(){
      showPhoto(settings.thumbnails.length-1);
    });
    $(settings.playControl).toggle(
      function(event){
        settings.tick = window.setInterval(
          function(){ $(settings.nextControl).click(); },
          settings.delay);
        $(event.target).addClass('photomatic-playing');
        $(settings.nextControl).click();
      },
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
        function(event){
          window.clearInterval(settings.tick);
          $(event.target).removeClass('photomatic-playing');
        });
      showPhoto(0);
      return this;
    };

})(jQuery);
```

This plugin is typical of jQuery-enabled code; it packs a big wallop in some compact code. But it serves to demonstrate an important set of techniques—using closures to maintain state across the scope of a jQuery plugin and to enable the creation of private implementation functions that plugins can define and use without resorting to any namespace infringements.

Also note that because we took such care to not let state "leak out" of the plugin, we are free to add as many Photomatic "widgets" to a page as we like without fear that they will interfere with one another.

But is it complete? You be the judge and consider the following exercises:

1. Again, error checking and handling has been glossed over. How would you go about making the plugin as bullet-proof as possible?
2. The transition from photo to photo is instantaneous. Leveraging your knowledge from chapter 5, change the plugin so that photos cross-fade to one another.
3. For maximum flexibility, we coded this plugin to instrument HTML elements already created by the user. How would you create an analogous plugin, but with less display freedom, that generated all the required HTML elements on the fly?

You're now primed and ready to write your own jQuery plugins. When you come up with some useful ones, consider sharing them with the rest of the jQuery community. Visit http://jquery.com/plugins/ for more information.

7.5 Summary

This chapter introduced us to writing reusable code that extends jQuery.

Writing our own code as extensions to jQuery has a number of advantages. Not only does it keep our code consistent across our web application regardless of whether it's employing jQuery APIs or our own, but it also makes all of the power of jQuery available to our own code.

Following a few naming rules helps avoid naming collisions between file names, as well as problems that might be encountered when the $ name is reassigned by a page that will use our plugin.

Creating new utility functions is as easy as creating new function properties on $, and new wrapper methods are as easily created as properties of `$.fn`.

If plugin authoring intrigues you, we highly recommend that you download and comb through the code of existing plugins to see how their authors implemented their own features. You'll see how the techniques presented in this chapter are used in a wide range of code and learn new techniques that are beyond the scope of this book.

Having yet more jQuery knowledge at our disposal, let's move on to learning how jQuery makes incorporating Ajax into our DOM-scripted Applications practically a no-brainer.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

8

Talk to the server with Ajax

This chapter covers:

- A brief overview of Ajax
- Loading preformatted HTML from the server
- Making general GET and POST requests
- Exerting fine-grained control over requests
- Setting default Ajax properties
- Handling Ajax events
- Comprehensive examples
- And more…

It can be successfully argued that no single technology has transformed the landscape of the web since its inception than the adoption of Ajax. The ability to make asynchronous requests back to the server without the need to reload entire pages has enabled a whole new set of user interaction paradigms and made DOM-scripted Applications possible.

Ajax is a less recent addition to the web toolbox than many people may realize. In 1998, Microsoft introduced the ability to perform asynchronous requests under script control (discounting the use of `<iframe>` elements for such activity) as an ActiveX control to enable the creation of Outlook Web Access (OWA). Although OWA was a moderate success, few people seemed to take notice of the underlying technology.

A few years passed, and a handful of events launched Ajax into the collective consciousness of the web development community. The non-Microsoft browsers implemented a standardized version of the technology as the XMLHttpRequest (XHR) object; Google began using XHR; and, in 2005, Jesse James Garrett of Adaptive Path coined the term *Ajax* (for Asynchronous JavaScript and XML).

As if they were only waiting for the technology to be given a catchy name, the web development masses suddenly took note of Ajax in a *big* way, and it has become one of the primary tools by which we can enable DOM-scripted Applications.

In this chapter, we'll take a brief tour of Ajax (if you're already an Ajax guru, you might want to skip ahead to section 8.2) and then look at how jQuery makes using Ajax a snap.

Let's start off with a refresher on what Ajax technology is all about.

8.1 Brushing up on Ajax

Although we'll take a quick look at Ajax in this section, please note that it's not intended as a complete Ajax tutorial or as an Ajax primer. If you're completely unfamiliar with Ajax (or worse, still think that we're talking about a dishwashing liquid or a mythological Greek hero), we encourage you to familiarize yourself with the technology through resources that are geared towards teaching you *all* about Ajax; the Manning books *Ajax in Action* and *Ajax in Practice* are both excellent examples.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Some people may argue that the term Ajax applies to *any* means to make server requests without the need to refresh the user-facing page (such as by submitting a request to a hidden `<iframe>` element), but most people associate the term with the use of XHR (XMLHttpRequest) or the Microsoft XMLHTTP ActiveX control.

Let's take a look at how those objects are used to generate requests to the server, beginning with creating an XHR instance.

8.1.1 Creating an XHR instance

In a perfect world, code written for one browser would work in all commonly used browsers. We've already learned that we don't live in that world, and things don't change with Ajax. There is a standard means to make asynchronous requests via the JavaScript XHR object, and an Internet Explorer proprietary means that uses an ActiveX control. With IE7, a wrapper that emulates the standard interface is available, but IE6 requires divergent code.

Once created (thankfully) the code to set up, initiate, and respond to the request is relatively browser-independent, and creating an instance of XHR is easy for any particular browser. The problem is that different browsers implement XHR in different ways, and we need to create the instance in the manner appropriate for the current browser.

But rather than relying on detecting which browser a user is running to determine which path to take, we'll use the preferred technique of *capability detection* that we introduced in chapter 6. Using this technique, we try to figure out what the browser's capabilities are, not which browser is being used. Capability detection results in more robust code because it can work in any browser that supports the tested capability.

The code of listing 8.1 shows a typical idiom used to instantiate an instance of XHR using this technique.

Listing 8.1 Capability detection resulting in code that can use Ajax in many browsers

```
var xhr;
if (window.XMLHttpRequest) {                          #A
  xhr = new XMLHttpRequest();
}
else if (window.ActiveXObject) {                      #B
  xhr = new ActiveXObject("Msxml2.XMLHTTP");
}
else {                                                #C
  throw new Error("Ajax is not supported by this browser");
}
```

#A Tests to see if XHR is defined
#B Tests to see if ActiveX is present
#C Throws error if there's no Ajax support

Cueballs in code only

After creation, the XHR instance sports a conveniently consistent set of properties and methods across all supporting browser instances. These properties and methods are shown in table 8.1, and the most commonly used of these will be discussed in the sections that follow.

Table 8.1 XMLHttpRequest (XHR) methods and properties

Table 1

Methods	Description
`abort()`	Causes the currentl executing request to be cancelled.
`getAllResponseHeaders()`	Returns a single string containing the names and values of all response headers.
`getResponseHeader(name)`	Returns the value of the named response header.
`open(method,url,async,username,password)`	Sets the method and destination URL of the request. Optionally, the request can be declared synchronous, and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

	a username and password can be supplied for requests requiring container-based authentication.
`send(context)`	Initiates the request with the specified (optional) body content.
`setRequestHeader(name,value)`	Sets a request header using the specified name and value.
Properties	**Description**
`onreadystatechange`	Assigns the event handler used when the state of the request changes.
`readyState`	An integer value that indicates the current state of the active request as follows: 0 = uninitialized 1 = loading 2 = loaded 3 = interactive 4 = complete
`responseText`	The body content returned in the response.
`responseXML`	If the body content is identified as XML, the XML DOM created from the body content.
`status`	The response status code returned from the server. For example: 200 for *success* or 404 for *not found*. See the HTTP Specification[1] for the full set of codes.
`statusText`	The status text message returned by the response.

[1] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10

With an XHR instance created, let's look at what it takes to set up and fire off the request to the server.

8.1.2 Initiating the request

Before we can send a request to the server, we need to perform the following setup steps:

1. Specify the HTTP method such as (POST or GET)
2. Provide the URL of the server-side resource to be contacted
3. Let the XHR instance know how it can inform us of its progress
4. Provide any body content for POST requests

We set up the first two items by calling the `open()` method of XHR as follows:

```
xhr.open('GET','/some/resource/url');
```

Note that this method does *not* cause the request to be sent to the server. It merely sets up the URL and HTTP method to be used. The `open()` method can also be passed a third Boolean parameter that specifies if the request is to be asynchronous (if `true`, which is the default) or synchronous (if `false`). There's seldom a good reason not to make the request asynchronous (even if it means we don't have to deal with callback functions); after all, the asynchronous nature of the request is usually the whole point of making a request in this fashion.

Third, we must provide a means for the XHR instance to tap us on the shoulder to let us know what's going on. We accomplish this by assigning a callback function to the `onreadystatechange` property of the XHR object. This function, known as the *ready state handler*, is invoked by the XHR instance at various stages of its processing. By looking at the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

settings of the various other properties of XHR, we can find out exactly what's going on with the request. We'll take a look at how a typical ready state handler operates in the next section.

The final steps to initiating the request are to provide any body content for POST requests and send it off to the server. Both of these are accomplished via the `send()` method. For GET requests, which typically have no body, no body content parameter is passed as follows:

```
xhr.send(null);
```

When request parameters are passed to POST requests, the string passed to the `send()` method must be in the proper format (which we might think of as *query string* format) in which the names and values must be properly URI-encoded. URI encoding is beyond the scope of this section (and as it turns out, jQuery is going to handle all of that for us), but if you're curious, do a web search for the term `encodeURIComponent`, and you'll be suitably rewarded.

An example of such a call is as follows:

```
xhr.send('a=1&b=2&c=3');
```

Now let's see what the ready state handler is all about.

8.1.3 Keeping track of progress

An XHR instance informs us of its progress through the *ready state handler*. This handler is established by assigning a reference to the function to serve as the ready handler to the `onreadystatechange` property of the XHR instance.

Once the request is initiated via the `send()` method, this callback will be invoked numerous times as the request makes transitions through its various states. The current state of the request is available as a numeric code in the `readyState` property (see the description of this property in table 8.1).

That's nice, but more times than not, we're only interested in when the request completes and whether it was successful or not. So frequently, we'll see ready handlers implemented using the pattern shown in listing 8.2.

Listing 8.2 Ready state handlers are often written to ginore all by the complete state

```
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {                    #A
    if (xhr.status >= 200 &&                    #B
        xhr.status < 300) {                     #B
      //success                                 #C
    }
    else {
      //error                                   #D
    }
  }
}
```

#A Ignores all but completed state
#B Branches on response status
#C Executes on success
#D Executes on failure

Cueballs in code only

This code ignores all but the completed state and, once complete, examines the value of the `status` property to determine if the request succeeded or not. The HTTP Specification defines all status codes in the 200 to 299 range as success and those with values of 300 or above as various types of failures.

We should note one thing about this ready handler; it references the XHR instance through a top-level, global variable. But shouldn't we expect the instance to be passed to the handler as a parameter?

Well, we could have *expected* that, but that's not what happens. The instance must be located by some other means, and in most script that's usually a top-level (global) variable. This could be a problem when we want to have more than one request firing simultaneously. We've already seen how such situations can be handled through the clever use of closures, but luckily, we shall see that the jQuery Ajax API handily solves this problem for us.

Now let's explore dealing with the response from a completed request.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

8.1.4 Getting the response

Once the ready handler has determined that the `readyState` is complete and that the request completed successfully, the body of the response can be retrieved from the XHR instance.

Despite the moniker Ajax (where the *X* stands for XML), the format of the response body can be any text format; it's not limited to just XML. In fact, most of the time, the response to Ajax requests is a format *other* than XML. It could be plain text or, perhaps, an HTML fragment; it could even be a text representation of a JavaScript object or array in JavaScript Object Notation (JSON) format (which is becoming increasingly popular as an exchange format).

Regardless of its format, the body of the response is available via the `responseText` property of the XHR instance (assuming that the request completes successfully). If the response indicates that the format of its body is XML by including a content-type header specifying a MIME type of *text/xml* (or any other XML MIME type), the response body will be parsed as XML. The resulting DOM will be available in the `responseXML` property. JavaScript (and jQuery itself, using its selector API) can then be used to process the XML DOM.

Processing XML on the client isn't rocket science, but—even with jQuery's help—it can still be a pain. Although there are times when nothing but XML will do for returning complex hierarchical data, frequently page authors will use other formats when the full power (and corresponding headache) of XML isn't absolutely necessary.

But some of those other formats aren't without their own pain. When JSON is returned, it must be converted into its runtime equivalent. When HTML is returned, it must be loaded into the appropriate destination element. And what if the HTML markup returned contains `<script>` blocks that need evaluation? We're not going to deal with these issues in this section because it isn't meant to be a complete Ajax reference and, more importantly, because we're going to find out that jQuery handles most of these issues on our behalf.

A diagram of this whole process is shown in figure 8.1.

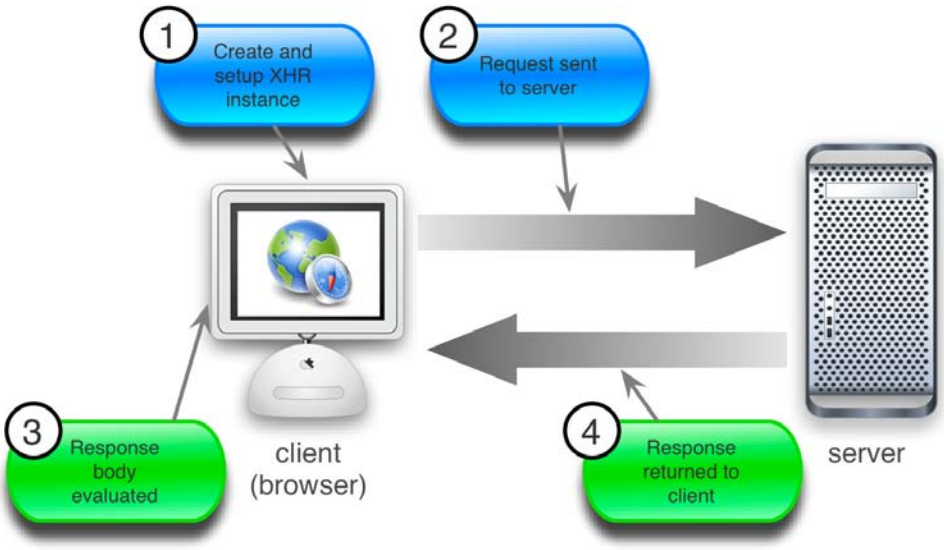


Figure 8.1 The life cycle of an Ajax request as it makes its way from the client to the server and back again

In this short overview of Ajax, we've identified the following pain points that page authors using Ajax need to deal with:

- Instantiating an XHR object requires browser-specific code.
- Ready handlers need to sift through a lot of uninteresting state changes.
- Ready handlers don't automatically get a reference to invoking XHR instances.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

- The response body needs to be dealt with in numerous ways depending upon its format.

The remainder of this chapter will describe how the jQuery Ajax methods and utility functions make Ajax a lot easier (and cleaner) to use on our pages. There are a lot of choices in the jQuery Ajax API, and we'll start with some of the simplest and most often-used tools.

8.2 Loading content into elements

Perhaps one of the most common uses of Ajax is to grab a chunk of content from the server and stuff it into the DOM at some strategic location. The content could be an HTML fragment that's to become the child content of a target container element, or it could be plain text that will become the content of the target element.

Setting up for the examples

Unlike all of the example code that we've examined so far in this book, the code examples for this chapter require the services of a web server to receive the Ajax requests to server-side resources. Because it's well beyond the scope of this book to discuss the operation of server-side mechanisms, we're going to set up some minimal server-side resources that send data back to the client without worrying about doing it for real, treating the server as a "black box"; we don't need or want to know how it's doing its job.

To enable the serving of these smoke-and-mirrors resources, you'll need to set up a web server of some type. For your convenience, the server-side resources have been set up in two formats: Java Server Pages (JSP) and PHP. The JSP resources can be used if you're running (or wish to run) a servlet/JSP engine; if you want to enable PHP for your web server of choice, you can use the PHP resources.

If you want to use the JSP resources but aren't already running a suitable server, instructions on setting up the free Tomcat web server are included with the sample code for this chapter. You'll find these instructions in the file `chapter8/tomcat.pdf`. And don't be concerned; even if you've never looked at a single line of Java, it's easier than you might think!

The examples found in the downloaded code are set up to use the JSP resources. If you want to switch the examples to use PHP, do a search-and-replace of all instances of the string *.jsp* with *.php*. Note that not all server-side resources have been translated from JSP to PHP, but the existing PHP resources should be enough to let the PHP-savvy fill in the rest of the resources.

Once you have the server of your choice set up, you can hit the URL `http://localhost:8080/chapter8/test.jsp` (to check your Tomcat installation) or `http://localhost/chapter8/test.php` (to check your PHP installation). The latter assumes that you have set up your web server (Apache or any other you have chosen) to use the example code root folder as a document base.

When you can successfully view the appropriate test page, you'll be ready to run the examples in this chapter.

Let's imagine that, on page load, we want to grab a chunk of HTML from the server using a resource named `someResource` and make it the content of a `<div>` element with an `id` of `someContainer`. For the final time in this chapter, let's look at how we'd do this without jQuery assistance. Using the patterns we set out earlier in this chapter, the body of the `onload` handler is as shown in listing 8.3. The full HTML file for this example can be found in the file `chapter8/listing.8.3.html`.

Listing 8.3 Using native XHR to fetch and include an HTML fragment

```
var xhr;

if (window.XMLHttpRequest) {
  xhr = new XMLHttpRequest();
}
else if (window.ActiveXObject) {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
    xhr = new ActiveXObject("Msxml2.XMLHTTP");
  }
  else {
    throw new Error("Ajax is not supported by this browser");
  }

  xhr.onreadystatechange = function() {
    if (xhr.readyState == 4) {
      if (xhr.status >= 200 && xhr.status < 300) {
        document.getElementById('someContainer')
          .innerHTML = xhr.responseText;
      }
    }
  }

  xhr.open('GET','someResource');
  xhr.send();
```

Although there's nothing tricky going on here, that's a non-trivial amount of code; 20-something lines or so, even accounting for blank lines that we added for readability.

The equivalent code we'd write as the body of a ready handler using jQuery is as follows:

```
$('#someContainer').load('someResource');
```

We're betting that we know which code you'd rather write and maintain! Let's take a close look at the jQuery method that we used in this statement.

8.2.1 Loading content with jQuery

The simple jQuery statement from the previous section easily loads content from the server-side resource using one of the most basic, but useful, jQuery Ajax methods: `load()`. The full syntax description of this method is as follows:

Method syntax: load

`load(url,parameters,callback)`

Initiates an Ajax request to the specified URL with optional request parameters. A callback function can be specified that's invoked when the request completes. The response text replaces the content of all matched elements.

Parameters

`url`	(String) The URL of the server-side resource to which the request is sent.
`parameters`	(String|Object|Array) Specifies any data that is to be passed as request parameters. This argument can be a string that will be used as the query string, an object whose properties are serialized into properly encoded parameters to be passed to the request, or an array of objects whose `name` and `value` properties specify the name/value pairs. If specified as an object or array, the request is made using the POST method. If omitted or specified as a string, the GET method is used.
`callback`	(Function) An optional callback function invoked after the response data has been loaded into the elements of the matched set. The parameters passed to this function are the response text, a status string (usually "success"), and the XHR instance. This function will be invoked once for each element in the wrapped set with the target element set as the function context (`this`).

Returns

The wrapped set.

Though simple to use, this method has some important nuances. For example, when the `parameters` parameter is used to supply the request parameters, the request is made using the POST HTTP method if an object hash or array is used;

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

otherwise, a GET request is initiated. If we want to make a GET request with parameters, we can include them as a query string on the URL. But be aware that when we do so, we're responsible for ensuring that the query string is properly formatted and that the names and values of the request parameters are URI-encoded. The JavaScript encodeURIComponent() method is handy for this, or you can employ the services of the jQuery `$.param()` utility function that we covered in chapter 6.

Most of the time, we'll use the `load()` method to inject the complete response into whatever elements are contained within the wrapped set, but sometimes we may want to filter elements coming back as the response. If we want to filter response elements, jQuery allows us to specify a selector on the URL that will be used to limit which response elements are injected into the wrapped elements. We specify the selector by suffixing the URL with a space and pound sign character (#) followed by the selector.

For example, to filter response elements so that only `<div>` instances are injected, we write

```
$('.injectMe').load('/someResource #div');
```

When it comes to supplying the data to be submitted with a request, some times we'll be winging it with ad hoc data, but frequently we'll find ourselves wanting to gather data that a user has entered into form controls.

As you might expect, jQuery's got some assistance up its sleeve.

SERIALIZING FORM DATA

If the data that we want to send as request parameters come from form controls, a helpful jQuery method for building a query string is `serialize()`, whose syntax is as follows:

Method syntax: serialize

`serialize()`

Creates a properly formatted and encoded query string from all successful form elements in the wrapped set, or all successful form elements of forms in the wrapped set.

Parameters

none

Returns

The formatted query string

The `serialize()` method is smart enough to only collect information from form control elements in the wrapped set, and only from those qualifying elements that are deemed *successful*. A successful control is one that would be included as part of a form submission according to the rules of the HTML Specification.[1] Controls such as unchecked check boxes and radio buttons, dropdowns with no selections, and any disabled controls are not considered successful and do not participate in form submission, so they are also ignored by `serialize()`.

If we'd rather get the form data in a JavaScript array (as opposed to a query string), jQuery provides the `serializeArray()` method.

Method syntax: serializeArray

`serializeArray()`

Collects the values of all successful form controls into an array of objects containing the names and values of the controls.

[1] http://www.w3.org/TR/html401/interact/forms.html#h-17.13.2

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Parameters

none

Returns

The array of form data

The array returned by `serializeArray()` is composed of anonymous object instances, each of which contains a `name` property and a `value` property that contain the name and value of each successful form control. Note that this is (not accidentally) one of the formats suitable for passing to the `load()` method to specify the request parameter data.

With the `load()` method at our disposal, let's put it to work solving some common real-world problems that many web developers encounter.

8.2.2 Loading dynamic HTML fragments

Often in business applications, particularly for commerce web sites, we want to grab real-time data from the server in order to present our users with the most up-to-date information. After all, we wouldn't want to mislead customers into thinking that they can buy something that's not available, would we? In this section, we'll begin to develop a page that we'll add onto throughout the course of the chapter. This page is part of a web site for a fictitious firm named *The Boot Closet*, an online retailer of overstock and closeout motorcycle boots. Unlike the fixed product catalogs of other online retailers, this inventory of overstock and closeouts is fluid, depending on what deals the proprietor was able to make that day and what's already been sold from the inventory. So it will be important for us to always make sure that we're displaying the latest info!

To begin our page (which will omit site navigation and other boilerplate to concentrate on the lessons at hand), we want to present our customers with a dropdown containing the styles that are currently available and, on a selection, display detailed information regarding that style. On initial display, the page will look as shown in figure 8.2.

Figure 8.2 The initial display of our commerce page with a simple drop-down just inviting customers to click on it

After the page first loads, a dropdown with the list of the styles currently available in the inventory is displayed. When no style is selected, we'll display a helpful message as a placeholder for the selection: "—choose a style –". This invites the user to interact with the dropdown, and when a user selects a boot style from this dropdown, here's what we want to do:

- Display the detailed information about that style in the area below the dropdown
- Remove the " – choose a style – " entry; once the user picks a style, it's served its purpose and is no longer meaningful

Let's start by taking a look at the HTML markup for the body that defines this page structure:

```
<body>
  <div id="banner">
    <img src="images/banner.boot.closet.png" alt="The Boot Closet"/>
  </div>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
  <div id="pageContent">
    <h1>Choose your boots</h1>
    <div>
      <div id="selectionsPane">                                    #1
        <label for="bootChooserControl">Boot style:</label> 
        <select id="bootChooserControl" name="bootStyle"></select>
      </div>
      <div id="productDetailPane"></div>                           #2
    </div>
  </div>
</body>
```

#1 Contains the selection control
#2 Holds place for product details

Cueballs in code and text

Not much to it, is there?

As would be expected, we've defined all the visual rendition information in an external stylesheet, and (adhering to the precepts of Unobtrusive JavaScript) we've included no behavioral aspects in the HTML markup.

The most interesting parts of this markup are a container (`#1`) that holds the `<select>` element that will allow customers to choose a boot style, and another container (`#2`) into which product details will be injected.

Note that the boot style control needs to have its option elements added before the user can interact with the page. So let's set about to adding the necessary behavior to this page.

The first thing we'll add is an Ajax request to fetch and populate the boot style dropdown.

> **NOTE**
>
> Under most circumstance, initial values such as this would be handled on the server prior to sending the HTML to the browser in the first place. But there are circumstances where pre-fetching data via Ajax may be appropriate, and we're doing that here, of course, if for nothing else than instructional purposes.

To add the options to the boot style control, we define a ready handler and within it we make use of the handy `load()` method:

```
$('#bootChooserControl')
  .load('/jqia2/action/fetchBootStyleOptions');
```

How simple is that? The only complicated part of this statement is the URL, which isn't really all that long or complicated, which specifies a request to a server-side action named `fetchBootStyleOptions`.

One of the nice things about using Ajax (with the ease of jQuery making it even nicer) is that it's completely independent of the server-side technology. Obviously the choice of server-side tech has an influence on the structure of the URLs, but beyond that we don't need to worry ourselves very much about what's going to transpire on the server. We simply make HTTP requests, sometimes with appropriate parameter data, and as long as the server returns the expected response, we could care less if the server is being powered by Java, Ruby, PHP, or even old-fashioned CGI.

In this particular case, we expect that the server-side resource will return the HTML markup representing the boot style options – supposedly from the inventory database. Our faux back-end code returns the following as the response:

```
<option value="">-- choose a style --</option>
<option value="7177382">Caterpillar Tradesman Work Boot</option>
<option value="7269643">Caterpillar Logger Boot</option>
<option value="7332058">Chippewa 9" Briar Waterproof Bison Boot</option>
<option value="7141832">Chippewa 17" Engineer Boot</option>
<option value="7141833">Chippewa 17" Snakeproof Boot</option>
<option value="7173656">Chippewa 11" Engineer Boot</option>
<option value="7141922">Chippewa Harness Boot</option>
<option value="7141730">Danner Foreman Pro Work Boot</option>
<option value="7257914">Danner Grouse GTX Boot</option>
```

which gets injected into the `<select>` element, resulting in a fully functional control.

Our next act is to instrument the dropdown so that it can react to changes, carrying out the duties that we listed earlier. The code for that is only slightly more complicated:

```
$('#bootChooserControl').change(function(event){                   #1
  $('#productDetailPane').load(                                    #2
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
    '/jqia2/action/fetchProductDetails',                          #2
    {style: $(event.target).val()},                               #2
    function() { $('[value=""]',event.target).remove(); }         #2
  );
});
```

#1 Establishes the change handler
#2 Fetches and displays the product detail

Cueballs in code and text

In this code, we select the boot style dropdown and bind a `change` handler (`#1`) to it. In the callback for the change handler, which will be invoked whenever a customer changes the selection, we obtain the current value of the selection by applying the `val()` method to the event target, which is the `<select>` element that triggered the event. We then once again employ the `load()` method (`#2`) to the `detailsDisplay` element to initiate an Ajax callback to a server-side resource, in this case `fetchProductDetails`, passing the style value as a parameter named `style`.

After the customer chooses an available boot style, the page will appear as shown in figure 8.3.

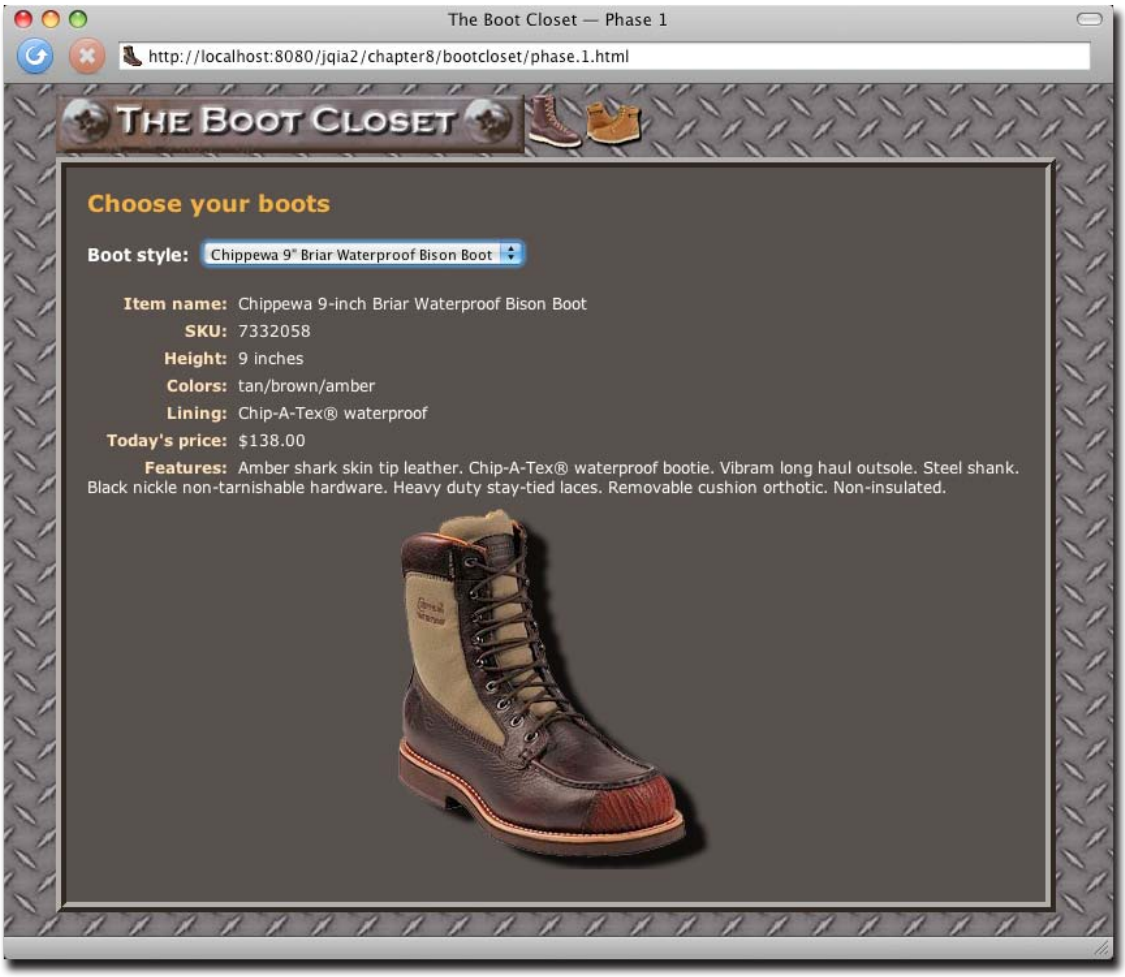


Figure 8.3 The server-side resource returns a pre-formatted fragment of HTML to display the detailed boot information.

The most notable operation performed in the ready handler is the use of the `load()` method to quickly and easily fetch snippets of HTML from the server and place them within the DOM as the children of an existing elements. This method is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

extremely handy and well suited to web applications powered by servers capable of server-side templating such as JSP and PHP.

Listing 8.4 shows the complete code for our Boot Closet page, which can be found in the file `chapter8/bootcloset/phase.1.html`. We'll be revisiting this page to add further capabilities to it as we progress through this chapter.

Listin 8.4 The first phase of the Boot Closet commerce page

```
<!DOCTYPE html>
<html>
  <head>
    <title>The Boot Closet &mdash; Phase 1</title>
    <link rel="stylesheet" type="text/css" href="../../styles/core.css">
    <link rel="stylesheet" type="text/css" href="bootcloset.css">
    <link rel="icon" type="image/gif" href="images/favicon.gif">
    <script type="text/javascript" src="../../scripts/jquery-1.3.2.min.js"></script>
    <script type="text/javascript" src="../../scripts/jqia2.support.js"></script>
    <script type="text/javascript">
      $(function() {

        $('#bootChooserControl')
          .load('/jqia2/action/fetchBootStyleOptions');

        $('#bootChooserControl').change(function(event){
          $('#productDetailPane').load(
            '/jqia2/action/fetchProductDetails',
            {style: $(event.target).val()},
            function() { $('[value=""]',event.target).remove(); }
          );
        });

      });
    </script>
  </head>

  <body>

    <div id="banner">
      <img src="images/banner.boot.closet.png" alt="The Boot Closet"/>
    </div>

    <div id="pageContent">

      <h1>Choose your boots</h1>

      <div>

        <div id="selectionsPane">
          <label for="bootChooserControl">Boot style:</label> 
          <select id="bootChooserControl" name="bootStyle"></select>
        </div>

        <div id="productDetailPane"></div>

      </div>

    </div>

  </body>

</html>
```

The `load()` method is tremendously useful when we want to grab a fragment of HTML to stuff into the content of an element (or set of elements). But there may be times when we either want more control over how the Ajax request gets made, or we need to do something more esoteric with the returned data in the response body.

Let's continue our investigation of what jQuery has to offer for these more complex situations.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

8.3 Making GET and POST requests

The `load()` method makes either a GET or a POST request, depending on how any request parameter data is provided, but sometimes we want to have a bit more control over which HTTP method gets used. Why should *we* care? Because, just maybe, our *servers* care.

Web authors have traditionally played fast and loose with the GET and POST methods, using one or the other without heeding how the HTTP protocol intends for these methods to be used. The intentions for each method are as follows:

- *GET requests*—Intended to be *idempotent*; the state of the server and the model data for the application should be unaffected by a GET operation. The same GET operation, made again and again and again, should return exactly the same results (assuming no other force is at work changing the server state).
- *POST requests*—Can be *non-idempotent*; the data they send to the server can be used to change the model state of the application; for example, adding or updating records in a database or removing information from the server.

A GET request should, therefore, be used whenever the purpose of the request is to merely get data; as its name implies. It may be required to *send* some parameter data to the server for the GET; for example, to identify a style number to retrieve color information. But when data is being sent to the server in order to effect a change, POST should be used.

WARNING

This is more than theoretical. Browsers make decisions about caching based upon the HTTP method used; GET requests are highly subject to caching. Using the proper HTTP method ensures that you don't get cross-ways with the browser's or server's expectations regarding the intentions of the requests.

With that in mind, if we look back to our "phase 1" implementation of The Boot Closet, we discover that *we're doing it wrong*! Because jQuery initiates a POST request when we supply an object hash for parameter data, we're making a POST when we really should be making a GET.

If we glance at a Firebug log (as shown in figure 8.4) when we display our page in Firefox, we can see that our second request, submitted when we make a selection from the style dropdown, is indeed a POST.

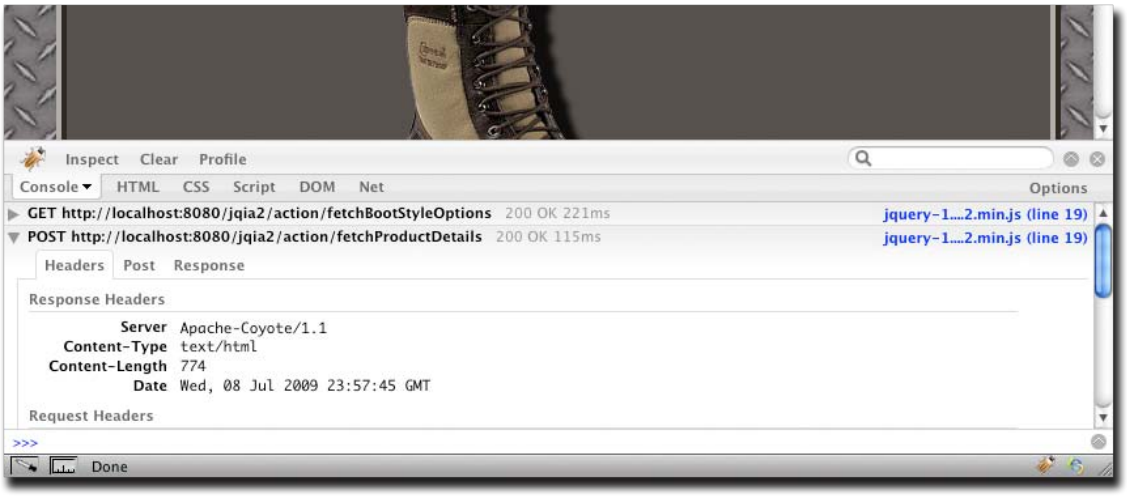


Figure 8.4 An inspection of the Firebug console shows that we're making a POST request when we should be making a GET

Get Firebug

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Trying to develop DOM-scripted application without the aid of a debugging tool is like trying to play concert piano with welding gloves. Why would you do that to yourself?

One important such tool to have in your tool chest is *Firebug*, a plugin for the Firefox browser. As shown in Figure 8.4, Firebug not only lets us inspect the JavaScript console, it lets us inspect the live DOM, the CSS, the script, and many other aspects of our page as we work through its development.

One feature most relevant for our current purposes is its ability to log Ajax requests along with both the request and response information.

For browsers other than Firefox, there's *Firebug Lite*, which simply loads as a JavaScript library while we're debugging,

Get Firebug at `http://getfirebug.com` and Firebug Lite at `http://getfirebug.com/lite.html`.

Does it really matter? That's up to you, but if we want to use HTTP in the manner in which it was intended, our request to fetch the boot detail should be a GET rather than a POST.

As we recall, we could simply make the parameter that specifies the request information a string rather than an object hash (and we'll revisit that a little later), but for now, let's take advantage of another way that jQuery lets us initiate Ajax requests.

8.3.1 Getting data with GET

jQuery gives us a few means to make GET requests, which unlike `load()`, aren't implemented as jQuery methods on a wrapped set. Rather, a handful of *utility functions* are provided to make various types of GET requests. As we pointed out in chapter 1, jQuery utility functions are top-level functions that are name-spaced with the `jQuery` global name (and its $ alias).

When we want to fetch some data from the server and decide what to do with it ourselves (rather than letting the `load()` method set it as the content of an HTML element), we can use the `$.get()` utility function. Its syntax is as follows:

Function syntax: $.get

`$.get(url,parameters,callback,type)`

Initiates a GET request to the server using the specified URL with any passed parameters as the query string.

Parameters

`url`	(String) The URL of the server-side resource to contact via the GET method.
`parameters`	(String|Object|Array) Specifies any data that is to be passed as request parameters. This parameter can be a string that will be used as the query string, an object whose properties are serialized into properly encoded parameters to be passed to the request, or an array of objects whose `name` and `value` properties specify the name/value pairs.
`callback`	(Function) An optional function invoked when the request completes successfully. The response body is passed as the first parameter to this callback, interpreted according to the setting of the `type` parameter, and the text status is passed as the second parameter.
`type`	(String) Optionally specifies how the response body is to be interpreted; one of "html", "text", "xml", "json", "script", or "jsonp". See the description of `$.ajax()` later in this chapter for more details.

Returns

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

The XHR instance.

The $.get() utility function allows us to initiate GET requests with a lot of versatility. We can specify request parameters (if appropriate) in numerous handy formats, provide a callback to be invoked upon a successful response, and even direct how the response is to be interpreted and passed to the callback. If even that's not enough versatility, we'll be seeing a more general function, `$.ajax()`, later on.

We'll be examining the `type` parameter in greater detail when we look at the `$.ajax()` utility function, but for now we'll just let it default to "html" or "xml" depending upon the content type of the response.

Applying `$.get()` to our Boot Closet page, we'll replace the use of the load() method with the $.get() function as shown in listing 8.5.

Listing 8.5 Changes to the Boot Closet style detail fetch to enforce a GET

```
$('#bootChooserControl').change(function(event){
  $.get(                                                    #1
    '/jqia2/action/fetchProductDetails',
    {style: $(event.target).val()},
    function(response) {
      $('#productDetailPane').html(response                 #2
      $('[value=""]',event.target).remove();
    }
  );
});
```

#1 Initiates GET request
#2 Injects response HTML

Cueballs in code and text

The changes are subtle, but significant. We call `$.get()` (#1) in place of `load()`, passing the same URL and the same request parameters. Because `$.get()` does no automatic injection of the response anywhere within the DOM, we need to do that ourselves, which is easily accomplished via a call to the `html()` method (#2).

The code for this version of our page can found in file `chapter8/bootcloset/phase.2.html`, and when we display it and select a style dropdown, we can see that a GET request has been made, as shown in figure 8.5.

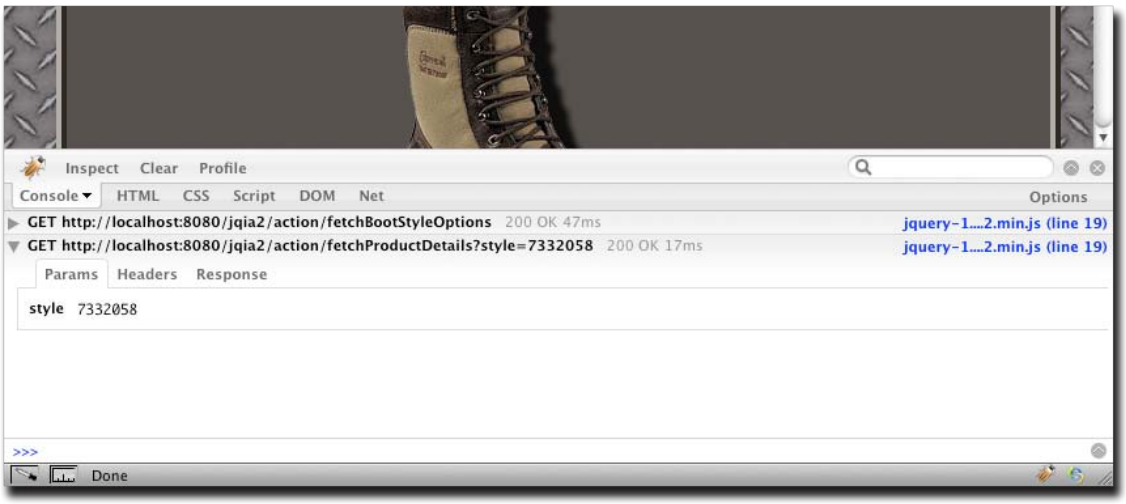


Figure 8.5 Now we can see that the second request is a GET rather than a POST as befitting the operation

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

In this example, we returned formatted HTML from the server and inserted it into the DOM, but as we can see by the `type` parameter to `$.get()` there are many other possibilities than just HTML. In fact, the term Ajax began its life as the acronym AJAX, where the X stood for XML.

When we pass the `type` as "xml" (remember, we'll be talking about type in more detail in just a little bit), and return XML from the server, the data passed to the callback is a parsed XML DOM. And while XML is great when we need its flexibility and our data is highly hierarchical in nature, it can be painful to traverse and digest the data. Let's see another jQuery utility function that's quite useful when our data needs are more straight-forward.

8.3.2 Getting JSON data

As stated in the previous section, when an XML document is returned from the server, the XML document is automatically parsed, and the resulting DOM is made available to the callback function. When XML is overkill or otherwise unsuitable as a data transfer mechanism, JSON is often used in its place; one reason being that JSON is astoundingly easy to digest in client-side script. Well, jQuery makes it even easier.

For times when we know that the response will be JSON, the `$.getJSON()` utility function automatically parses the returned JSON string and makes the resulting JavaScript data item available to its callback. The syntax of this utility function is as follows:

Function syntax: $.getJSON

`$.getJSON(url,parameters,callback)`

Initiates a GET request to the server using the specified URL with any passed parameters as the query string. The response is interpreted as a JSON string, and the resulting data is passed to the callback function.

Parameters

`url`	(String) The URL of the server-side resource contacted via the GET method.
`parameters`	(String|Object|Array) Specifies any data that is to be passed as request parameters. This parameter can be a string that will be used as the query string, an object whose properties are serialized into properly encoded parameters to be passed to the request, or an array of objects whose `name` and `value` properties specify the name/value pairs.
`callback`	(Function) A function invoked when the request completes. The data value resulting from digesting the response body as a JSON representation is passed as the first parameter to this callback, and the status text is passed as the second parameter.

Returns

The XHR instance.

This function – which is simply a convenience function for `$.get()` with a `type` of "json" -- is great for those times when we want to get data from the server without the overhead of dealing with XML.

Between `$.get()` and `$.getJSON()`, jQuery gives us some powerful tools when it comes to making GET requests, but man does not live by GETs alone!

8.3.3. Making POST requests

"Sometimes you feel like a nut, sometimes you don't." What's true of choosing between an Almond Joy or a Mounds candy bar is also true of making requests to the server. Sometimes we want to make a GET, but at other times we want (or even need) to make a POST request.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

There are any number of reasons why we might choose a POST over a GET. First, the intention of the HTTP protocol is that POST will be used for any non-idempotent requests. Therefore, if our request has the potential to cause a change in the server-side state, it should be a POST. Moreover, accepted practices and conventions aside, a POST operation must sometimes be used when the data to be passed to the server exceeds the small amount that can be passed by URL in a query string; a limit that's a browser-dependent value. And sometimes, the server-side resource we contact may only support POST operations, or it might even perform different functions depending upon whether our request uses the GET or POST method.

For those occasions when a POST is desired or mandated, jQuery offers the `$.post()` utility function, which operates in a similar fashion to `$.get()`, except for employing the POST HTTP method. Its syntax is as follows:

Function syntax: $.post

`$.post(url,parameters,callback,type)`

Initiates a POST request to the server using the specified URL with any parameters passed within the body of the request.

Parameters

`url`	(String) The URL of the server-side resource to contact via the POST method.
`parameters`	(String|Object|Array) Specifies any data that is to be passed as request parameters. This parameter can be a string that will be used as the query string, an object whose properties are serialized into properly encoded parameters to be passed to the request, or an array of objects whose `name` and `value` properties specify the name/value pairs.
`callback`	(Function) A function invoked when the request completes. The response body is passed as the single parameter to this callback, and the status text as the second.
`type`	(String) Optionally specifies how the response body is to be interpreted; one of "html", "text", "xml", "json", "script", or "jsonp". See the description of $.ajax() for more details.

Returns

The XHR instance.

Except for making a POST request, using `$.post()` is identical to using `$.get()`. jQuery takes care of the details of passing the request data in the request body (as opposed to the query string), and sets the HTTP method appropriately.

Now, getting back to our Boot Closet project, we've made a really good start, but there's more to buying a pair of boots than just selecting a style; customers are sure to want to pick which color they want, and certainly they'll need to specify their size. We'll use these additional requirements to show how to solve one of the most-asked questions in online Ajax forums, that of...

8.3.4 Implementing cascading dropdowns

The implementation of cascading dropdowns -- where subsequent dropdown options depend upon the selections of previous dropdowns -- has become sort of a poster child for Ajax on the Web. And while you will find thousands, perhaps tens of thousands of solutions, we're going to implement a solution on our Boot Closet page that demonstrates how ridiculously simple jQuery makes it.

We've already seen how easy it was to load a dropdown dynamically with server-powered option data. We'll see that tying multiple dropdowns together in a cascading relationship will be only slightly more work.

Lets dig in by listing the changes we need to make for the next phase of our page:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

- Add dropdowns for color and size.
- When a style is selected, add options to the color dropdown that show the colors available for that style.
- When a color is selected, add options to the size dropdown that show the sizes available for the selected combination of style and color.
- Make sure things remain consistent. This includes removing the "—please make a selection –-" options from newly created dropdowns once they've been used once, and making sure that the three dropdowns never show an invalid combination.

We're also going to revert to using `load()` again, this time coercing it to initiate a GET rather than a POST. It's not that we have anything against `$.get()`, but `load()` just seems more natural when we're using Ajax to load HTML fragments.

To start off, let's examine the new HTML markup added to define the additional dropdowns. A new container for the select elements is defined to contain three labeled elements:

```
<div id="selectionsPane">
  <label for="bootChooserControl">Boot style:</label>
  <select id="bootChooserControl" name="style"></select>
  <label for="colorChooserControl">Color:</label>
  <select id="colorChooserControl" name="color" disabled="disabled"></select>
  <label for="sizeChooserControl">Size:</label>
  <select id="sizeChooserControl" name="size" disabled="disabled"></select>
</div>
```

The previous style selection element remains, but has been joined by two more: one for color, and one for size, each of which is initially empty and disabled.

That was easy, and takes care of the additions to the structure. Now let's add the additional behaviors.

The style selection dropdown must now perform double duty. Not only must it continue to fetch and display the boot details when a selection is made, its change handler must now also populate and enable the color selection dropdown with the colors available for whatever style was chosen. And, we also wanted to revert to using `load()`.

Let's refactor the fetching of the details first. We want to use `load()`, but we also want to force a GET, as opposed to the POST that we were initiating earlier. In order to have `load()` induce a GET, we need to pass a string, rather than an object hash, to specify the request parameter data. Luckily, with jQuery's help, we won't have to build that string ourselves. So the first part of the change handler for the style dropdown gets refactored to:

```
$('#bootChooserControl').change(function(event){
  $('#productDetailPane').load(
    '/jqia2/action/fetchProductDetails',
    $('#bootChooserControl').serialize()              #A
  );
  // more to follow
});
```

#A Provides query string

Cueballs in code

By using the `serialize()` method, we create a string representation of the value of the style dropdown, thereby coercing the `load()` method to initiate a GET, just as we wanted.

The second duty that the change handler needs to perform is to load the color choice dropdown with appropriate values for the chosen style, and then enable it. Let's take a look at the rest of the code to be added to the handler:

```
$('#colorChooserControl').load(
  '/jqia2/action/fetchColorOptions',                  #1
  $('#bootChooserControl').serialize(),               #1
  function(){
    $('#colorChooserControl')                         #2
      .attr('disabled',false);                        #2
    $('#sizeChooserControl')                          #3
      .attr('disabled',true)                          #3
      .find('option').remove();                       #3
  }
);
```

#1 Fetches and loads color options

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

#2 Enables color control
#3 Empties and disables size control

Cueballs in code and text

This code should look familiar, as it's just another use of `load()`, this time to an action named `fetchColorOptions` which is designed to return a set of formatted `<option>` elements representing the colors available for the chosen style (which we again passed as request data) (#1). This time, we've also specified a callback to be executed when the GET request successfully returns a response.

In this callback we perform two important tasks. First, we enable the color chooser control (#2). The call to `load()` injected the `<option>` elements, but once populated it would still be disabled if we did not enable it.

Secondly, the callback disables and empties the size chooser control (#3). But why? (Pause a moment and think about it.)

Even though the size control will already be disabled and empty the first time that the style chooser's value is changed, what about later on? What if, after the customer chooses a style and a color (which we will soon see results in the population of the size control), he or she (hey, the ladies ride motorcycles too!) changes the selected style?

Since the sizes displayed depend upon the combination of style and color, the sizes previously displayed are no longer applicable and do not reflect a consistent view of what's chosen. Therefore, whenever the style changes, we simply blow the size options away and reset the size control to initial conditions.

Before we sit back and enjoy a lovely beverage, we've got more work to do. We still have to instrument the color chooser dropdown to use the selected style and color values to fetch and load the size chooser dropdown. The code to do this follows a familiar pattern:

```
$('#colorChooserControl').change(function(event){
  $('#sizeChooserControl').load(
    '/jqia2/action/fetchSizeOptions',
    $('#bootChooserControl,#colorChooserControl').serialize(),
    function(){
      $('#sizeChooserControl').attr('disabled',false);
    }
  );
});
```

Upon a change event, the size information is obtained via the `fetchSizeOptions` action, passing both the boot style and color selections, and the size control is enabled.

There's just one more thing that we need to do. When each dropdown is initially populated, it's seeded with an `<option>` element with a blank value and display text along the lines of "— choose a something --". You may recall that in the previous phases of this page, we had added code to remove that option from the style dropdown upon selection.

Well, we could add such code the change handlers for the style and color dropdowns, and add instrumentation for the size dropdown (which currently has none) to add that. But let's be a bit more suave about it.

One capability of the event model that often gets ignored by many a web developer is event bubbling. Page authors frequently focus only on the targets of events, and forget that events will bubble up the DOM tree, where handlers can deal with those events in more general ways than at the target level.

If we recognize that removing the option with a blank value from any of the three dropdowns can be handled in the exact same fashion *regardless* of which dropdown is the target of the event, we can avoid copying the same code in three places by establishing a *single* handler, higher in the DOM, that will recognize and handle the change events.

Recalling the structure of the document, the three dropdown are contained within a `<div>` element with an id of `selectionsPane`, and so we can handle the removal of the temporary option for all three dropdowns with the following, single listener:

```
$('#selectionsPane').change(function(event){
  $('[value=""]',event.target).remove();
});
```

This listener will be triggered whenever a change event happens on any of the enclosed dropdowns, and remove the option with the blank value within the context of the target of the event (which will be the changed dropdown). How slick is that?

Keeping event bubbling in mind to prevent having to repeat the same code in lower level handlers can really elevate your script to the big leagues!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

With that, we've completed phase 3 of The Boot Closet, adding cascading dropdowns into the mix as shown in figure 8.6. We can use the same techniques in any pages where dropdown values depend upon previous selections.

Figure 8.6 The 3rd phase of The Boot Closet showed how easy it is to implement Cascading Dropdowns

The full code of the page is now as shown in listing 8.6.

Listing 8.6 The Boot Closet, now with cascading dropdowns!

```
<!DOCTYPE html>
<html>
  <head>
    <title>The Boot Closet &mdash; Phase 2</title>
    <link rel="stylesheet" type="text/css" href="../../styles/core.css">
    <link rel="stylesheet" type="text/css" href="bootcloset.css">
    <link rel="icon" type="image/gif" href="images/favicon.gif">
    <script type="text/javascript" src="../../scripts/jquery-1.3.2.min.js"></script>
    <script type="text/javascript" src="../../scripts/jqia2.support.js"></script>
    <script type="text/javascript">
      $(function() {

        $('#bootChooserControl')
          .load('/jqia2/action/fetchBootStyleOptions');

        $('#bootChooserControl').change(function(event){
          $('#productDetailPane').load(
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
            '/jqia2/action/fetchProductDetails',
            $('#bootChooserControl').serialize()
          );
          $('#colorChooserControl').load(
            '/jqia2/action/fetchColorOptions',
            $('#bootChooserControl').serialize(),
            function(){
              $('#colorChooserControl').attr('disabled',false);
              $('#sizeChooserControl')
                .attr('disabled',true)
                .find('option').remove();
            }
          );
        });

        $('#colorChooserControl').change(function(event){
          $('#sizeChooserControl').load(
            '/jqia2/action/fetchSizeOptions',
            $('#bootChooserControl,#colorChooserControl').serialize(),
            function(){
              $('#sizeChooserControl').attr('disabled',false);
            }
          );
        });

        $('#selectionsPane').change(function(event){
          $('[value=""]',event.target).remove();
        });

      });
    </script>
  </head>

  <body>

    <div id="banner">
      <img src="images/banner.boot.closet.png" alt="The Boot Closet"/>
    </div>

    <div id="pageContent">

      <h1>Choose your boots</h1>

      <div>

        <div id="selectionsPane">
          <label for="bootChooserControl">Boot style:</label>
          <select id="bootChooserControl" name="style"></select>
          <label for="colorChooserControl">Color:</label>
          <select id="colorChooserControl" name="color" disabled="disabled"></select>
          <label for="sizeChooserControl">Size:</label>
          <select id="sizeChooserControl" name="size" disabled="disabled"></select>
        </div>

        <div id="productDetailPane"></div>

      </div>

    </div>

  </body>

</html>
```

As we have seen, with the `load()` method and the various GET and POST jQuery Ajax functions at our disposal, we can exert some measure of control over how our request is initiated and how we're notified of its completion. But for those times when we need *full* control over an Ajax request, jQuery has a means for us to get as picky as we want.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

8.4 Taking full control of an Ajax request

The functions and methods that we've seen so far are convenient for many cases, but there may be times when we want to take control of all the nitty-gritty details into our own hands.

In this section, we'll explore how jQuery lets us exert such dominion.

8.4.1 Making Ajax requests with all the trimmings

For those times when we want or need to exert a fine-grained level of control over how we make Ajax requests, jQuery provides a general utility function for making Ajax requests named `$.ajax()`. Under the covers, all other jQuery features that make Ajax requests eventually use this function to initiate the request. Its syntax is as follows:

Function syntax: $.ajax

`$.ajax(options)`

Initiates an Ajax request using the passed options to control how the request is made and callbacks notified.

Parameters

`options` (Object) An object whose properties define the parameters to this operation. See table 8.2 for details.

Returns

The XHR instance.

Looks simple, doesn't it? But don't be deceived. The `options` parameter can specify a large range of values that can be used to tune the operation of this function. These options (in general order of their importance and the likelihood of their use) are defined in table 8.2.

Table 8.2 Options for the `$.ajax()` utility function

Name	Type	Description
`url`	String	The URL for the request.
`type`	String	The HTTP method to use. Usually either "post" or "get". If omitted, the default is GET.
`data`	String or Object or Array	Defines the values to serve as the query parameters to be passed to the request. If the request is a GET, this data is passed as the query string. If a POST, the data is passed as the request body. In either case, the encoding of the values is handled by the `$.ajax()` utility function. This parameter can be a string that will be used as the query string or response body, an object whose properties are serialized, or an array of objects whose `name` and `value` properties specify the name/value pairs.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`dataType`	String	A keyword that identifies the type of data that's expected to be returned by the response. This value determines what, if any, post-processing occurs upon the data before being passed to callback functions. The valid values are as follows: `xml`—The response text is parsed as an XML document and the resulting XML DOM is passed to the callbacks. `html`—The response text is passed unprocessed to the callbacks functions. Any `<script>` blocks within the returned HTML fragment are evaluated. `json`—The response text is evaluated as a JSON string, and the resulting object is passed to the callbacks. `jsonp`—Similar to `json` except that remote scripting is allowed, assuming the remote server supports it. `script`—The response text is passed to the callbacks. Prior to any callbacks being invoked, the response is processed as a JavaScript statement or statements. `text`—The response text is assumed to be plain text. The server resource is responsible for setting the appropriate content-type response header. If this property is omitted, the response text is passed to the callbacks without any processing or evaluation.
`cache`	Boolean	If `true`, ensures that the response will not be cached by the browser. Defaults to `false` except when `dataType` is specified as one of "script" or "jsonp".
`timeout`	Number	Sets a timeout for the Ajax request in milliseconds. If the request does not complete before the timeout expires, the request is aborted and the error callback (if defined) is called.
`global`	Boolean	Enables (if `true`) or disables (if `false`) the triggering of global Ajax events. These are jQuery-specific custom events that trigger at various points or conditions during the processing of an Ajax request. We'll be discussing them in detail in the upcoming section. If omitted, the default is to enable the triggering of global events.
`contentType`	String	The content type to be specified on the request. If omitted, the default is "application/x-www-form-urlencoded", the same type used as the default for form submissions.
`success`	Function	A function invoked if the response to the request indicates a success status code. The response body is returned as the first parameter to this function and evaluated according to the specification of the `dataType` property. The second parameter is a string containing a status value—in this case, always "success".

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`error`	Function	A function invoked if the response to the request returns an error status code. Three arguments are passed to this function: the XHR instance, a status message string (in this case, always "error"), and an optional exception object returned from the XHR instance, if any.
`complete`	Function	A function called upon completion of the request. Two arguments are passed: the XHR instance and a status message string of either *success* or *error*. If either a success or error callback is also specified, this function is invoked after that callback is called.
`beforeSend`	Function	A function invoked prior to initiating the request. This function is passed the XHR instance and can be used to set custom headers or to perform other pre-request operations.
`async`	Boolean	If specified as `false`, the request is submitted as a synchronous request, By default, the request is asynchronous.
`processData`	Boolean	If set to `false`, prevents the data passed from being processed into URL-encoded format. By default, the data is URL-encoded into a format suitable for use with requests of type "application/x-www-form-urlencoded".
`dataFilter`	Function	A callback invoked to filter the response data. This function is passed the raw response data and the `dataType` value, and is expected to return the "sanitized" data.
`ifModified`	Boolean	If `true`, allows a request to succeed only if the response content has not changed since the last request according to the *Last-Modified* header. If omitted, no header check is performed.
`jsonp`	String	Specifies a query parameter name to override the default jsonp callback parameter name of '"callback".
`username`	String	The username to be used in the event of an HTTP authentication request.
`password`	String	The password to be used in the case of an HTTP authentication request.
`scriptCharset`	String	The character set to be used for "script" and "jsonp" requests when the remote and local content are of different character sets.
`xhr`	Function	A callback used to provide a custom implementation of the XHR instance.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

That's a lot of options to keep track of, but it's unlikely that more than a few of them will be used for any one request. Even so, wouldn't it be convenient if we could set default values for these options for pages where we're planning to make a large number of requests?

8.4.2 Setting request defaults

Obviously the last question in the previous section was a setup. As you might have suspected, jQuery provides a way for us to define a default set of Ajax properties that will be used when we don't override their values. This can make pages that initiate lots of similar Ajax calls much simpler.

The function to set up the list of Ajax defaults is `$.ajaxSetup()`, and its syntax is as follows:

Method syntax: $.ajaxSetup

`$.ajaxSetup(options)`

Establishes the passed set of option properties as the defaults for subsequent calls to `$.ajax()`.

Parameters

`options`	(Object) An object instance whose properties define the set of default Ajax options. These are the same properties described for the `$.ajax()` function in table 8.2. This function should not be used to set callback handlers for success, error and completion. (We'll see how to set these up using an alternative means in an upcoming section.)

Returns

Undefined.

At any point in script processing, usually at page load (but can be at any point of the page authors' choosing), this function can be used to set up defaults to be used for all subsequent calls to `$.ajax()`.

NOTE

Defaults set with this function aren't applied to the `load()` method. Also, for utility functions such as `$.get()` and `$.post()`, the HTTP method can't be overridden by use of these defaults. For example, setting a default `type` of "get" won't cause `$.post()` to use the GET HTTP method.

Let's say that we are setting up a page where, for the majority of Ajax requests (made with the utility functions rather than the `load()` method), we want to set up some defaults so that we don't need to specify them on every call. We can, as the first statement in the header `<script>` element, write

```
$.ajaxSetup({
  type: 'POST',
  timeout: 5000,
  dataType: 'html'
})
```

This would ensure that every subsequent Ajax call (except as noted above) would use these defaults, unless explicitly overridden in the properties passed to the Ajax utility function being used.

Now, what about those *global events* we mentioned that were controlled by the `global` property?

8.4.3 Ajax events

Throughout the execution of jQuery Ajax requests, jQuery triggers a series of custom events for which we can establish handlers in order to be informed of the progress of a request, or to take action at various points along the way.

jQuery classifies these events as "local" events and "global" events.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

Local events are handled by the callback functions that we can directly specify using the `beforeSend`, `success`, `error` and `complete` options of the `$.ajax()` function, or indirectly by providing callbacks to the convenience methods (which in turn use the `$.ajax()` function to make the actual request)s. So we've been handling local events all along, without even knowing it, whenever we've registered a callback function to any jQuery Ajax function.

Global events are those that are triggered just like other custom events within jQuery, and for which we can establish event handlers via the `bind()` method (just like any other event). The global events, many of which mirror local events, are: "ajaxStart", "ajaxSend', "ajaxSuccess", "ajaxError", and "ajaxComplete".

When triggered, the global events are broadcast to every element in the DOM, so we can establish these handlers on any DOM element, or elements, of our choosing. When executed, the handlers' function context is set to the DOM element upon which the handler was established.

Because we do not need to consider a bubbling hierarchy, we can establish a handler on any element for which it would be convenient to have ready access via `this`. If we don't care about a specific element, we could just establish the handler on the <body> for lack of a better location. But if we have something specific to do to an element, let's say hide and show an animated graphics while and Ajax request is processing, we could establish the handle on that element and have easy access to it via the function context.

In addition to the function context, more information is available via three parameters passed to the Ajax event handlers: the jQuery.Event instance, the XHR instance, and the options passed to `$.ajax()`.

Table 8.3 shows the jQuery Ajax events in the order in which they are delivered.

Table 8.3 jQuery Ajax Event Types

Event name	Type	Description
`ajaxStart`	global	Triggered when an Ajax request is started, as long as no other requests are active. For concurrent requests, this event is triggered only for the first of the requests.
`beforeSend`	local	Invoked prior to initiating the request in order to allow modification of the XHR instance prior to send the request to the server.
`ajaxSend`	global	Triggered prior to initiating the request in order to allow modification of the XHR instance prior to send the request to the server.
`success`	local	Invoked when a request returns a successful response.
`ajaxSucces`	global	Triggered when a request returns a successful response.
`error`	local	Invoked when a request returns an error response.
`ajaxError`	global	Triggered when a request returns an error response.
`complete`	local	Invoked when a request completes, regardless of status. This callback is invoked even for synchronous requests.
`ajaxComplete`	global	Triggered when a request completes, regardless of status. This callback is invoked even for synchronous requests.
`ajaxStop`	global	Triggered when an Ajax request completes *and* there are no other concurrent requests active.

Once again (just to make sure things are clear), "local" events represent callbacks passed to `$.ajax()` (and its cohorts), while "global" events are custom events that are triggered and can be handled by established handlers, just like other event types.

In addition to using `bind()` to establish event handlers, jQuery also provides a handful of convenience functions to establish the handlers, as follows:

Method syntax: jQuery Ajax event establishers

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

ajaxComplete(callback)
ajaxError(callback)
ajaxSend(callback)
ajaxStart(callback)
ajaxStop(callback)
ajaxSuccess(callback)

Establishes the passed callback as an event handler for the jQuery Ajax event specified by the method name.

Parameters

`callback`	(Function) The function to be established as the Ajax event handler. The function context (`this`) is the DOM element upon which the handler is established, and three parameters are passed: the jQuery.Event instance, the XHR instance, and the options object.

Returns

The wrapped set.

Let's put together a simple example of how some of these methods can be used to easily track the progress of Ajax requests. The layout of our test page (it's too simple to be called a Lab) is as shown in figure 8.7 and is available in the file `chapter8/ajax.events.html`.

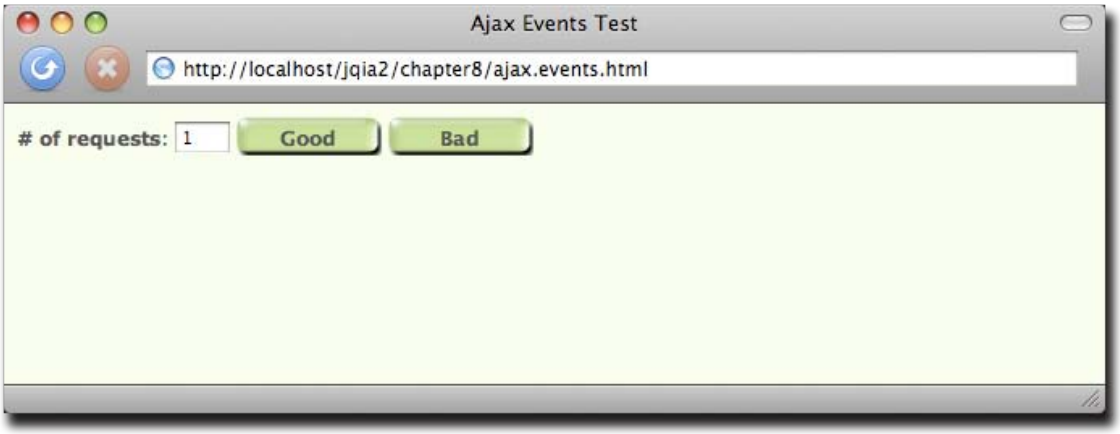


Figure 8.7 Initial display of the page we'll use to examine the jQuery Ajax Events by firing multiple events and observing the handlers

This page exhibits three controls: a count field, a "Good" button, and a "Bad" button. These buttons are instrumented to issue the number of requests specified by the count field. The "Good" button will issues request to a valid resource, while the "Bad" button will issue that number of requests to an invalid resource that will result in failures.

Within the ready handler on the page we also establish a number of event handlers as follows:

```
$('body').bind(
  'ajaxStart ajaxStop ajaxSend ajaxSuccess ajaxError ajaxComplete',
  function(event){ say(event.type); }
);
```

This statement establishes a handler for each of the various jQuery Ajax event types that emits a message to the on-page "console" (which we placed below the controls) showing the event type that was triggered.

Leaving the request count at 1, click the "Good" button and observe the results. You will see that each jQuery Ajax event type is triggered in the order depicted in table 8.3.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

But to understand the distinctive behavior of the “ajaxStart” and “ajaxStop” events, set the count control to 2, and click the “Good” button. You will see a display as shown in figure 8.8.

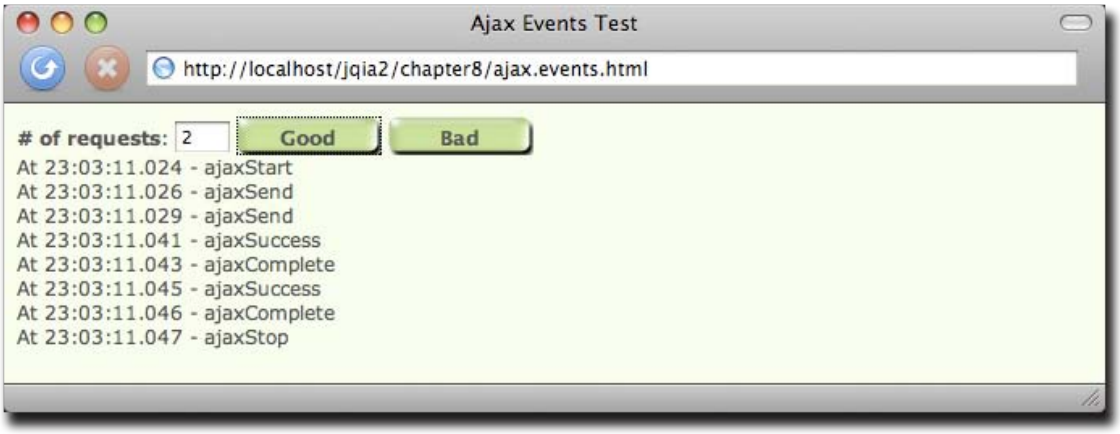


Figure 8.8 When multiple requests are active, the ajaxStart and ajaxStop events are called around the set of requests rather than for each.

Here we can see how, when multiple request are active, that the “ajaxStart” and “ajaxStop” events are triggered only once for entire set of concurrent requests, while the other event types are triggered on a per-request basis

Before we move on to the next chapter, let’s put all this grand knowledge to use, shall we?

8.5 Putting it all together

It’s time for another comprehensive example. Let’s put a little of everything we’ve learned so far to work: selectors, DOM manipulation, advanced JavaScript, events, effects, and Ajax. And to top it all off, we’ll implement another custom jQuery method!

For this example, we’ll once again return to The Boot Closet page. To review, look back at figure 8.6 to remind yourself where we left off, because we’re going to continue to enhance this page.

In the detailed information of the boots listed for sale (evident in figure 8.6), terms are used that our customers may not be familiar with—terms like “Goodyear welt” and “stitch-down construction”. We’d like to make it easy for customers to find out what these terms mean because an informed customer is usually a happy customer. And happy customers buy things! We like that.

We could be all 1998 about it and provide a glossary page that customers navigate to for reference, but that would move the focus away from where we want it—the pages where they can buy our stuff! We could be a *little* more modern about it and open a pop-up window to show the glossary or even the definition of the term in question. But even that’s being terribly old-fashioned.

If you’re thinking ahead, you might be wondering if we could use the `title` attribute of DOM elements to display a *tooltip* (sometimes called a *flyout*) containing the definition when customers hover over the term with the mouse cursor. Good thinking! That would allow the definition to be shown in-place without the need for the customers to have to move their focus elsewhere.

But the `title` attribute approach presents some problems for us. First, the flyout only appears if the mouse cursor hovers over the element for a few seconds—and we’d like to be a bit more overt about it, displaying the information immediately after clicking a term—but, more importantly, some browsers will truncate the text of a `title` flyout to a length far too short for our purposes.

So we’ll build our own!

We’ll somehow identify terms that have definitions, change their appearance to allow the user to easily identify such elements as clickable (giving them what’s termed an “invitation to engage”), and instrument the elements so that a mouse click will display a flyout containing a description of the term. Subsequently clicking the flyout will remove it from the display.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

We're also going to write it as a generally reusable plugin, so we need to make sure of two very important things:

1. There'll be no hard-coding of anything that's specific to The Boot Closet
2. We'll give the page authors maximum flexibility for styling and layout (within reason)

We'll call our new plugin *The Termifier*, and figures 8.9a through 8.9c display a portion of our page showing the behavior that we'll be adding.

Figure 8.9a The terms "full-grain" and "oil-tanned" have been instrumented for "termifying" by our handy new plugin

Figure 8.9b The Termifier pane deployed using very simple styling specified by CSS external to the plugin

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

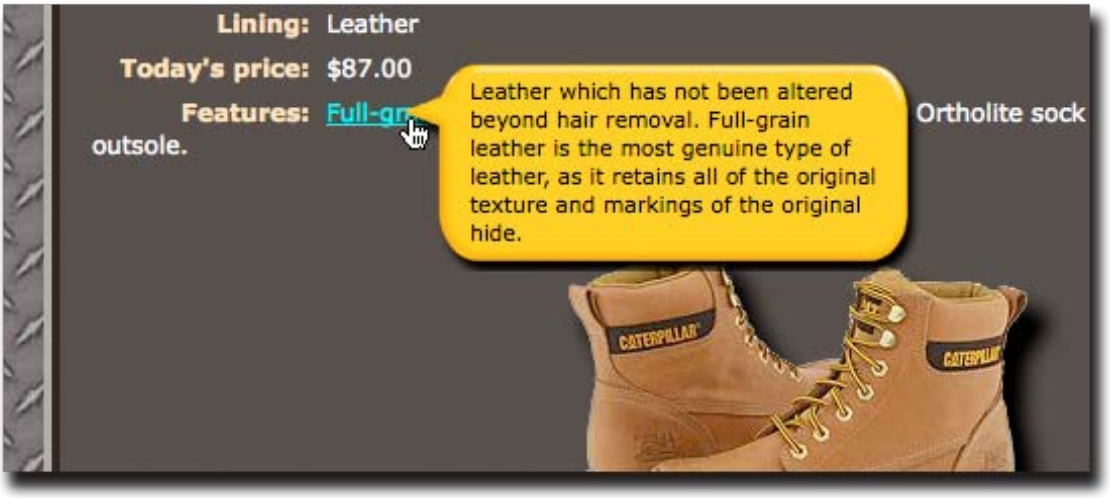


Figure 8.9c The Termifier pane with fancier styling – we need to give a user of our plugin this kind of flexibility

In figure 8.9a we see the description of the item with the terms *Full-grain* and *oil-tanned* highlighted. Clicking Full-grain causes the Termifier flyout containing the term's definition to be displayed as shown in figures 8.9b and 8.9c. In the rendition of figure 8.9b, we've supplied some rather simple CSS styling, while in figure 8.9c we've gotten a bit more grandiose. We need to make sure that the plugin code allows for such flexibility.

Let's get started

8.5.1 Implementing the Termifier

As we recall, adding a jQuery method is accomplished by use of the `$.fn` property. Because we've called our new plugin *The Termifier*, we'll name the method `termifier()`.

The `termifier()` method will be responsible for instrumenting each element in its matched set to achieve the following plan:

- Establish a `click` handler on each matched element that initiates the display of The Termifier flyout.
- Once clicked, the term defined by the current element will be looked up using a server-side resource.
- Once received, the definition of the term will be displayed in a flyout using a fade-in effect.
- The flyout will be instrumented to fade out once clicked within its boundaries.
- The URL of the server-side resource will be the only required parameter; all other options will have reasonable defaults.

The syntax for our plugin is as follows:

Method syntax: termifier

`termifier(url,options)`

Instruments the wrapped elements as Termifier terms. The class name `termified` is added to all wrapped elements.

Parameters

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

`url` (String) The URL of the server-side action that will retrieve term definitions.

`options` (Object) Specifies the options as follows:

- `paramName`, the request parameter name to use to send the term to the server-side action. If omitted, a default of "term" is used.
- `addClass`, a class name to be added to the outer container of the generated Termifier pane. This is in addition to the class name `termifier`, which is always added.
- `origin`, an object hash containing the properties `top` and `left` that specify an offset for the Termifier pane from cursor position. If omitted, the origin is placed exactly at the cursor position.
- `zIndex`, the z-index to assign to the Termifier pane. Defaults to 100.

Returns

The wrapped set.

We'll begin the implementation by creating a skeleton for our new `termifier()` method in a file named `jquery.jqia.termifier.js`:

```
(function($){

  $.fn.termifier = function(actionURL,options) {
    //
    // implementation goes here
    //
    return this;
  };

})(jQuery);
```

This skeleton uses the pattern outlined in the previous chapter to ensure that we can freely use the `$` in our implementation, and creates the wrapper method by adding the new function to the `fn` prototype. Also note how we set up the return value right away to ensure that our new method plays nice with jQuery chaining.

Now, onto processing the options. We want to merge the user-supplied options with our own defaults, so:

```
var settings = $.extend({
  origin: {top:0,left:0},
  paramName: 'term',
  addClass: null,
  zIndex: 100,
  actionURL: actionURL
},options||{});
```

We've seen this pattern before, so we won't belabor its operation, but note how we've added the `actionURL` parameter value into the `settings` variable. This collects everything we'll need later into one tidy place for the closures that we'll be creating to look.

Having gathered all the data, we'll now move on to defining the click handler on the wrapped elements that will create and display the Termifier pane. We start setting that up as follows:

```
this.click(function(event){
  $('div.termifier').remove();
  //
  // create new Termifier here
  //
});
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

When a "termified" element is clicked, before we create a new Termifier pane, we want to get rid of any previous ones that are lying around. Otherwise, we could end up with a screen littered with them, so we locate all previous instances and remove them from the DOM.

With that, we're now ready to create the structure of our Termifier pane. You might think that all we need to do is to create a single `<div>` into which we can shove the term definition, but while that *would* work, it would also limit the options that we afford the users of our plugin. Consider the example of figure 8.9c where the text needs to be placed precisely in relation to the background "bubble" image.

So we'll create an outer `<div>`, and then an inner `<div>` into which the text will be placed. And this won't only be useful for placement; consider the situation of figure 8.10 in which we have a fixed-height construct and text that's longer than will fit. The presence of the inner `<div>` allows the page author to user the `overflow` CSS rule to add scroll bars to the flyout text.

Figure 8.10 Having two `<div>` elements to play with gives the page author some leeway to do things like scroll the inner text

Let's examine the code that creates the outer `<div>`:

```
$('<div>')                                                    #1
  .addClass('termifier' +                                     #2
    (settings.addClass ? (' ') + settings.addClass : ''))
  .css({                                                      #3
    position: 'absolute',
    top: event.pageY - settings.origin.top,
    left: event.pageX - settings.origin.left,
    display: 'none'
  })
  .click(function(event){                                     #4
    $(this).fadeOut('slow');
  })
  .appendTo('body')                                           #5
```

**#1 Creates the outer **
#2 Adds class name(s)
#3 Assigns CSS positioning
#4 Removes from display on click
#5 Attaches to DOM

Cueballs in code and text

In this code we create a new `<div>` element (#1) and proceed to adjust it. First, we assign the class name `termifier` to the element (#2) so that we can easily find it later as well as to give the page author a hook onto which to hang CSS rules. If the caller provided an `addClass` option, it is also added.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

We then apply some CSS styling (#3). All we do here is the minimum that is necessary to make the whole thing work (we'll let the page author provide any additional styling through CSS rules). The element is initially hidden, and absolutely positioned at the location of the mouse event, adjusted by any `origin` provided by the caller. The latter is what allows a page author to adjust the position so that the tip of the bubble's pointer in figure 8.9c appears at the click location.

After that, we establish a click event handler (#4) that removes the element from the display when clicked upon. Finally, the element is attached to the DOM (#5).

OK, so far so good, now we need to create the inner `<div>` -- the one that will carry the text – and append it to the element we just created, so we continue with:

```
.append(                                                    #1
  $('<div>').load(                                          #2
    settings.actionURL,
    encodeURIComponent(settings.paramName) + '=' +          #3
      encodeURIComponent($(event.target).text()),           #3
    function(){                                             #4
      $(this).closest('.termifier').fadeIn('slow');
    }
  )
```

#1 Appends the inner to the outer
#2 Fetches and injects the definition
#3 Provides the term
#4 Fades the Termifier into view

Note that this is a continuation of the same statement that created the outer `<div>` -- have we not been telling you all along how powerful jQuery chaining is?

In this code fragment, we create and append (#1) the inner `<div>` and initiate an Ajax request to fetch and inject the definition of the term (#2). Because we are using `load()` and want to force a GET request, we need to supply the parameter info as a text string. We can't rely on `serialize()` here as we're not dealing with any form controls, so we make use of JavaScript's `encodeURIComponent()` method to format the query string ourselves (#3).

In the completion callback for the request, we find the parent `<div>` and fade it into view (#4).

Before dancing a jig, we need to perform one last act before we can declare our plugin complete; we must add the class name `termified` to the wrapped elements to give the page author a way to style "termified" elements:

```
this.addClass('termified');
```

There! Now we're done and can enjoy our lovely beverage.

The code for our plugin is shown in its entirety in listing 8.7, and can be found in file `chapter8/jquery.jqia.termifier.js`.

Listing 8.7 The complete implementation of the Termifier plugin

```
(function($){

  $.fn.termifier = function(actionURL,options) {
    var settings = $.extend({
      origin: {top:0,left:0},
      paramName: 'term',
      addClass: null,
      actionURL: actionURL
    },options||{});
    this.click(function(event){
      $('div.termifier').remove();
      $('<div>')
        .addClass('termifier' +
          (settings.addClass ? (' ') + settings.addClass : ''))
        .css({
          position: 'absolute',
          top: event.pageY - settings.origin.top,
          left: event.pageX - settings.origin.left,
          display: 'none'
        })
        .click(function(event){
          $(this).fadeOut('slow');
        })
        .appendTo('body')
        .append(
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
            $('<div>').load(
              settings.actionURL,
              encodeURIComponent(settings.paramName) + '=' +
                encodeURIComponent($(event.target).text()),
              function(){
                $(this).closest('.termifier').fadeIn('slow');
              }
            )
          );
      });
      this.addClass('termified');
      return this;
    };

})(jQuery);
```

That was the hard part. The easy part should be putting the Termifier to use on our Boot Closet page – at least if we did it right. Let's find out if we did.

Now, let's see what it takes to apply this method to our Boot Closet page.

8.5.2 Putting the Termifier to the test

Because we rolled all the complex logic of creating and manipulating The Termifier flyout into the `termifier()` method, using this new jQuery method on the Boot Closet page is relatively simple. But first we have some interesting decisions to make.

For example, we need to decide how to identify the terms on the page that we wish to *termify*. Remember, we need to construct a wrapped set of elements whose content contains the term elements for the method to operate on. We *could* use a `<span>` element with a specific class name; perhaps something like

```
<span class="term">Goodyear welt</span>
```

in which case creating a wrapped set of these elements would be as easy as `$('span.term')`.

But some might feel that the `<span>` markup is a bit wordy. Instead, we'll leverage the little-used HTML tag: `<abbr>`. The `<abbr>` tag was added to HTML 4 in order to help identify abbreviations in the document. Because the tag is intended purely for identifying document elements, none of the browsers do much with these tags, either in the way of semantics or visual rendition, so it's perfect for our use.

NOTE

HTML 4[2] defines a few more of these document-centric tags such as `<cite>`, `<dfn>`, and `<acronym>`. The HTML 5 Draft Specification[3] proposal adds even more of these document-centric tags whose purpose is to provide structure rather than provide layout or visual rendition directives. Among such tags are `<section>`, `<article>`, and `<aside>`.

Therefore, the first thing that we need to do is to modify the server-side resource that returns the item details to enclose terms that have glossary definitions in `<abbr>` tags. Well, as it turns out, the `fetchProductDetails` action already does that. But because the browsers don't do anything with the `<abbr>` tag, you might not have even noticed unless you've already taken a look inside the action file or inspected the action's response. A typical response (for style 7141922) contains:

```
<div>
  <label>Features:</label> <abbr>Full-grain</abbr> leather uppers. Leather
  lining. <abbr>Vibram</abbr> sole. <abbr>Goodyear welt</abbr>.
</div>
```

Note how the terms *Full-grain*, *Vibram* and *Goodyear welt* are identified using the `<abbr>` tag.

Now, onward to the page itself. Starting with the code of "phase 3" as a starting point, let's see what we need to add to the page in order to use The Termifier. We need to bring the new method into the page, so we add the following statement to the `<head>` section (after jQuery itself has loaded):

[2] http://www.w3.org/TR/html4/

[3] http://www.w3.org/html/wg/html5/

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
<script type="text/javascript" src="jquery.jqia.termifier.js"></script>
```

We need to apply the `termifier()` method to any `<abbr>` tags added to the page when item information is loaded, so we add a callback to the `load()` method that fetches the product detail information. That callback uses The Termifier to instrument all `<abbr>` elements. The augmented `load()` method (with changes in bold) is as follows:

```
$('#productDetailPane').load(
  '/jqia2/action/fetchProductDetails',
  $('#bootChooserControl').serialize(),
  function(){ $('abbr').termifier('/jqia2/action/fetchTerm'); }
);
```

The added callback creates a wrapped set of all `<abbr>` elements and applies the `termifier()` method to them, specifying a server-side action of `fetchTerm` and letting all options default.

And that's it! (Well, almost.)

CLEANING UP AFTER OURSELVES

Because we wisely encapsulated all the heavy lifting in our reusable jQuery plugin, using it on the page is even easier than pie! And we can as easily use it on any other page or any other site. Now that's what *engineering* is all about!

But there is one little thing we forget. We built into our plugin the removal of any Termifier flyouts when another one is displayed, but what happens if the user chooses a new style? Whoops! We'd be left with a Termifier pane that is no longer relevant. So we need to remove any displayed Termifiers whenever we reload the product detail.

We *could* just add some code to the `load()` callback, but that seems wrong, tightly coupling the Termifier to the loading of the product details. We'd be happier if we could keep the two decoupled and just listen for an event that tells us when its time to remove any Termifiers.

If the "ajaxComplete" event came to mind, treat yourself to a Maple Walnut Sundae or whatever other indulgence you use to reward yourself for a great idea.

We can listen for "AjaxComplete" events and remove any Termifiers that exist when the event is tied to the `getProductDetails` action with:

```
$('body').ajaxComplete(function(event,xhr,options){
 if (options.url.indexOf('fetchProductDetails') != -1) {
   $('div.termifier').remove();
 }
});
```

Now let's take a look at how we applied those various styles to the termifier flyouts.

TERMIFYING IN STYLE

Styling the elements is quite a simple matter. In our style sheet we can easily apply rules to make the termified terms, and the Termifier pane itself, look like the display of figure 8.9b. Looking in `bootcloset.css`, we find:

```
abbr.termified {
  text-decoration: underline;
  color: aqua;
  cursor: pointer;
}

div.termifier {
  background-color: cornsilk;
  width: 256px;
  color: brown;
  padding: 8px;
  font-size: 0.8em;
}
```

These rules give the terms a "link-ish" appearance that invites the users to click the terms, and gives the Termifier flyouts the simple appearance shown in figure 8.9b.

The code for this version of the page can be found in `chapter8/bootcloset/phase4a.html`.

To take the Termifier panes to the next level shown in figure 8.9c, we only need to be a little clever and to use some of the options we provided in our plugin.

For the fancier version, we call the Termifier plugin (within the `load()` callback) with:

```
$('#productDetailPane').load(
  '/jqia2/action/fetchProductDetails',
  $('#bootChooserControl').serialize(),
  function(){ $('abbr')
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

```
      .termifier(
        '/jqia2/action/fetchTerm',
        {
          addClass: 'fancy',
          origin: {top: 28, left: 2}
        }
      );
    }
  );
```

which differs from the previous example only by specifying that the class name `fancy` be added to the Termifiers, and that the origin be adjusted so that the tip of the "bubble" appears at the mouse event location.

To the style sheet we add (leaving the original rule):

```
div.termifier.fancy {
  background: url('images/termifier.bubble.png') no-repeat transparent;
  width: 256px;
  height: 104px;
}

div.termifier.fancy div {
  height: 86px;
  width: 208px;
  overflow: auto;
  color: black;
  margin-left: 24px;
}
```

which adds all the fancy stuff that can be seen in figure 8.9c.

This new page can be found in the file `chapter8/bootcloset/phase.4b.html`. Our new plugin is useful and powerful, but as always, we can make improvements.

8.5.3 Improving the Termifier

Our brand-spankin'-new jQuery plugin is quite useful as is, but it does have some minor issues and the potential for some major improvements. To hone your skills, here's a list of possible changes you could make to this method or to the Boot Closet page:

- Add an option (or options) that allows the page author to control the fade durations or, perhaps, even to use alternate effects.
- The Termifier flyout stays around until the customer clicks it, another one is displayed, or the product details are reloaded. Add a timeout option to the Termifier plugin that automatically makes the flyout go away if it's still displayed after the timeout has expired.
- Clicking the flyout to close it introduces a usability issue because the text of the flyout can't be selected for cut-and-paste. Modify the plugin so that it closes the flyout if the user clicks anywhere on the page *except* on the flyout.
- We don't do any error handling in our plugin. How would you enhance the code to gracefully deal with invalid caller info, or server-side errors?
- We achieved the appealing drop shadows in our images by using PNG files with partial transparencies. Although most browsers handle this file format well, IE6 does not and displays the PNG files with white backgrounds. To deal with this we could also supply GIF formats for the images without the drop shadows. How would you enhance the page to detect when IE6 is being used and to replace all the PNG references with their corresponding GIFs?
- While we're talking about the images, we only have one photo per boot style, even when multiple colors are available. Assuming that we have photo images for each possible color, how would you enhance the page to show the appropriate image when the color is changed?

Can you think of other improvements to make to this page or the `termifier()` plugin? Share your ideas and solutions at this book's discussion forum, which you can find at http://www.manning.com/bibeault2.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566

Licensed to Zhanna Kulyk <janetkulyk@yahoo.com>

8.6 Summary

Not surprisingly, this is one of the longest chapters in this book. Ajax is a key part of the new wave of DOM-scripted Applications, and jQuery is no slouch in providing a rich set of tools for us to work with.

For loading HTML content into DOM elements, the `load()` method provides an easy way to grab the content from the server and make it the contents of any wrapped set of elements. Whether a GET or POST method is used is determined by how any parameter data to be passed to the server is provided.

When a GET is required, jQuery provides the utility functions `$.get()` and `$.getJSON()`; the latter being useful when JSON data is returned from the server. To force a POST, the `$.post()` utility function can be used.

When maximum flexibility is required, the `$.ajax()` utility function, with its ample assortment of options, lets us control the most minute aspects of an Ajax request. All other Ajax features in jQuery use the services of this function to provide their functionality.

To make managing the bevy of options less of a chore, jQuery provides the `$.ajaxSetup()` utility function that allows us to set default values for any frequently used options to the `$.ajax()` function (and to all of the other Ajax functions that use the services of `$.ajax()`).

To round out the Ajax toolset, jQuery also allows us to monitor the progress of Ajax requests by triggering Ajax events at the various stages, allowing us to establish handlers to listen for those events. We can `bind()` the handlers, or use the convenience methods: `ajaxStart()`, `ajaxSend()`, `ajaxSuccess()`, `ajaxError()`, `ajaxComplete()`, and `ajaxStop()`.

With this impressive collection of Ajax tools under our belts, it's easy to enable rich functionality in our web applications. And remember, if there's something that jQuery doesn't provide, we've seen that it's easy to extend jQuery by leveraging its existing features. Or, perhaps, there's already a plugin—official or otherwise—that adds exactly what you need!

But that's the subject of a later chapter.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=566