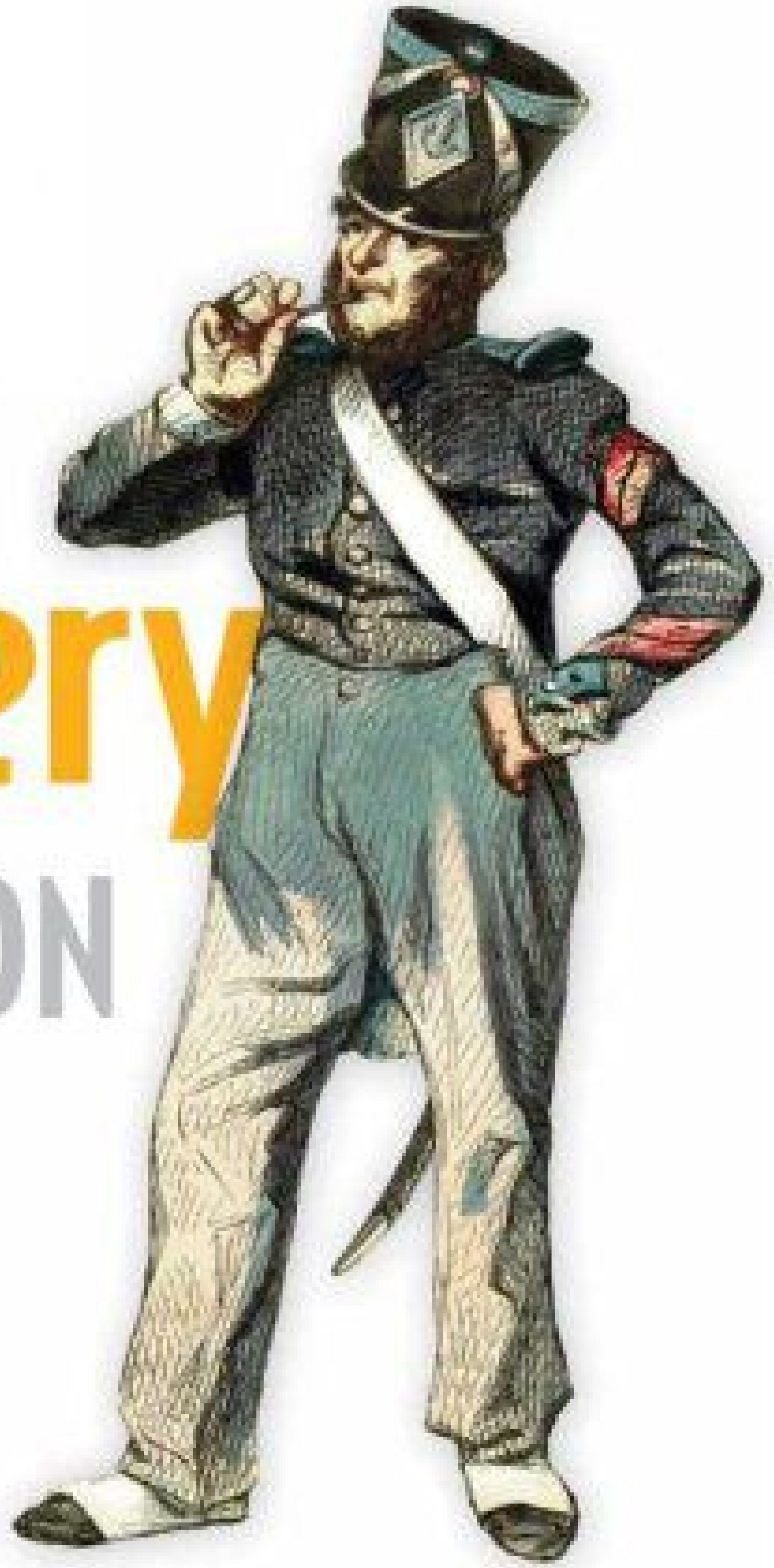
Covers jQuery 1.4 and jQuery UI 1.8

Bear Bibeault
Yehuda Katz

jQuery IN ACTION

SECOND EDITION



MANNING

Table of Contents

Chapter 6. Beyond the DOM with jQuery utility functions.. 1
Section 6.1. Using the jQuery flags.. 2
Section 6.2. Using other libraries with jQuery.. 9
Section 6.3. Manipulating JavaScript objects and collections.. 12
Section 6.4. Miscellaneous utility functions.. 26
Section 6.5. Summary.. 34

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 `$()` function. But you may recall that, way back in chapter 1, we also introduced the concept of *utility functions*—functions namespaced by `jQuery/$` that don't operate on a wrapped set. These functions could be thought of as top-level functions except that they're 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 an Ajax request).

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:

- We wanted to guide you into thinking in terms of using jQuery wrapper 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.
- 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 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 properties defined on `$`. Many of these *flags* are focused on helping us divine the capabilities of the current browser, but others help us control the behavior of jQuery at a page-global level.

The jQuery flags intended for public use are as follows:

- `$.fx.off`—Enables or disabled effects
- `$.support`—Details supported features
- `$.browser`—Exposes browser details (officially deprecated)

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 includes various animated effects. 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're in an animation-adverse environment, we can simply set the value of `$.fx.off` to `true`.

This will *not* suppress any effects we've used on the page; it will simply disable the animation of those effects. For example, the fade effects will show and hide the elements immediately, without the intervening animations.

Similarly, calls to the `animate()` method will 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 predefined 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.

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, the browser detection technique should be avoided.

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 and Chrome.

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—IE 6, IE 7, and IE 8, 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 next section) 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.

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 within which our code would actually work
- Unscalable, leading to enormous nested if and if-else statements to sort things out
- Inaccurate, due to users spoofing 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 chapter 4 on event handling as an example. Remember that there are two advanced event-handling models: the W3C standard DOM Level 2 Event Model and the proprietary Internet Explorer Event 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 which 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're 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!

Feature detection is vastly superior to browser detection. It's more reliable, and it doesn't accidentally block browsers that support the capability we're 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? iCab? Epiphany? Konqueror?

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 nontrivial or downright 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 browser capability flags are exposed to us as properties of jQuery's `$.support` object.

Table 6.1 summarizes the flags that are available in this object.

Table 6.1 The `$.support` browser capability flags

Flag property	Description
`boxModel`	Set to `true` if the user agent renders according to the standards-compliant box model. This flag isn't set until the document is 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 element's `style` property is used.

Table 6.1 The `$.support` browser capability flags *(continued)*

Flag property	Description
`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 sheet 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` if the JavaScript `getElementsByTagName()` 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` if the browser evaluates `<script>` blocks when they’re injected via calls to the `appendChild()` or `createTextNode()` methods.
`style`	Set to `true` if the attribute for obtaining the inline style properties of an element is `style`.
`tbody`	Set to `true` if a browser doesn’t 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.

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 because 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 that can only be made using browser detection (we'll see an example in a moment). 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. They're set up when the library is loaded, making them available even before any ready handlers have executed, and they're defined as properties of an object instance with a reference of `$.browser`.

Note that even though these flags remain present in jQuery 1.3 and beyond, they're 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 when browser development had stagnated somewhat, but now that we've entered an era when 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's recommended that, when you need something more than the core support flags provide, you create new ones of your own. But we'll get to that in just a bit.

The browser support flags are described in table 6.3.

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, so specific browser identification should not be necessary.

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.

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 in 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 3.6, the reported version is `1.9.2`—the version of the Gecko rendering engine. This value *is* handy for distinguishing between versions of Internet Explorer, as the rendering engine versions and browser versions match.

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 or feature on which to perform detection.

> **NOTE** Even in these cases, it's possible to set a feature flag by 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 by the W3C as follows (at http://www.w3.org/TR/DOM-Level-2-HTML/html.html#ID-14493106, for those of us who like to look at the specifications):

```
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>`) before which the new element is to be placed. In standards-compliant browsers, this second parameter is a *reference* to the existing element as specified; in Internet Explorer, however, it's the *ordinal index* of the existing element.

Because there's no way to perform feature detection to determine whether we should pass an object reference or an integer value (short of trying it out, as noted previously), 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 to determine whether it's 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's recommended that we abstract away the browser detection by creating 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 own 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.

Let's 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 `$` global name is the largest point of contention and conflict when using other libraries on the same page as jQuery. As we know, jQuery uses `$` 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 `$.noConflict()` utility function to relinquish control of the `$` identifier 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 `$` identifier back to another library, allowing mixed library use on pages using jQuery. Once this function is executed, jQuery features will need to be invoked using the `jQuery` identifier rather than the `$` identifier.
Optionally, the `jQuery` identifier 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` identifier 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 nonconflicting, alias for `jQuery`, such as

```
var $j = jQuery;
```

Another idiom we may often see employed is to create an environment where the `$` identifier is scoped to refer to the `jQuery` object. This technique is commonly used when extending jQuery, particularly by plugin authors who can't make any assumptions regarding whether page authors have called `$.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 `$` identifier 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?

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 whatever 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 in listing 6.1 (available in chapter6/ready.handler.test.1.html).

Listing 6.1 Ready handler test 1

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hi!</title>
    <script type="text/javascript" src="../scripts/jquery-1.4.js"></script>
    <script type="text/javascript">
      var $ = 'Hi!';                                    ❶ Overrides $ name with custom value
      jQuery(function(){                      ❷ Declares the ready handler
        alert('$ = '+ $);
      });
    </script>
  </head>
  <body></body>
</html>
```

In this example, we import jQuery, which (as we know) defines the global names `jQuery` and its alias `$`. We then redefine the global `$` variable to a string value ❶, 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.

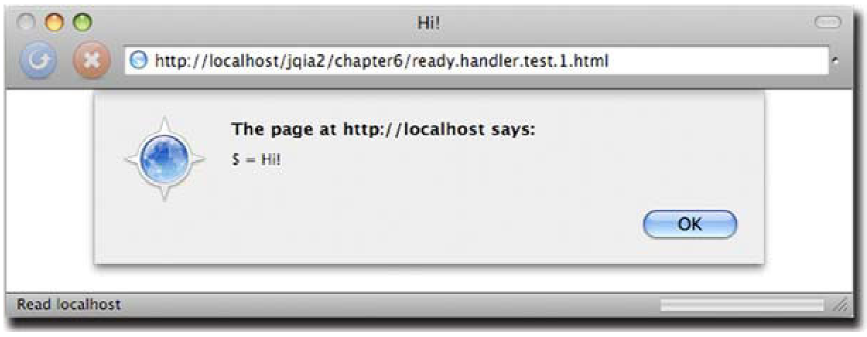
We then define the ready handler ❷ whose only action is to display an alert showing the value of `$`.

When we load this page, we see the alert displayed, as shown in figure 6.1.

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.

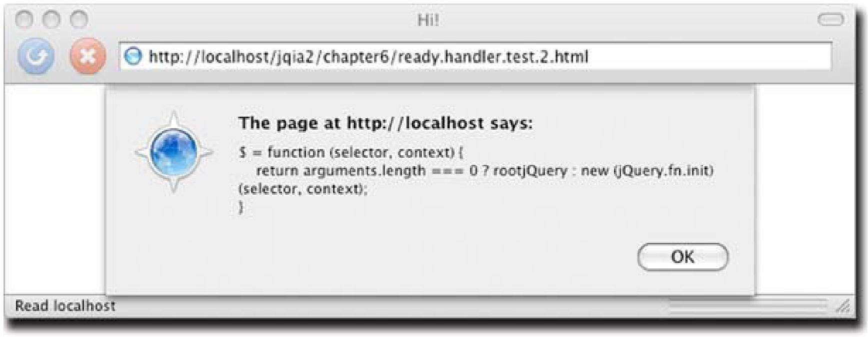
Now let's make one change to this example document. The following code shows only the portion of the document that has been modified; the minimal change is highlighted in bold. (You can get the full page in chapter6/ready.handler.test.2.html.)

```
<script type="text/javascript">
  var $ = 'Hi!';
  jQuery(function($){
    alert('$ = '+ $);
  });
</script>
```



The page at http://localhost says:
$ = Hi!

Figure 6.1 The $ says, "Hi!" as its redefinition takes effect within the ready handler.



The page at http://localhost says:

```
$ = function (selector, context) {
   return arguments.length === 0 ? rootjQuery : new (jQuery.fn.init)
(selector, context);
}
```

Figure 6.2 The alert now displays the jQuery version of $ because its definition has been enforced within the function.

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 6.2.

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 `$` within that function, the `$` identifier 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.

These functions run the gamut from simple string manipulation and type testing to complex collection filtering, serialization of form values, and even implementing a form of object inheritance through property merging.

Let's start with one that's pretty basic.

6.3.1 Trimming strings

Almost inexplicably, the JavaScript `String` type 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 characters before or after 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 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 character `\u00A0`.

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 nonscalar 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.

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.

> **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()`. Bear in mind that if you need to be concerned with performance to that level, you'll get the 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. Although 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 the data 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 `$.grep()` function might lead us to believe that the function employs the use of regular expressions like its namesake, the UNIX `grep` command. But the filtering criterion used by the `$.grep()` utility function isn't a regular expression; it's a callback function provided by the caller that defines the criteria to determine whether a data value should be included 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 isn't 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 whether 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 whether 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'd 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 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).

U.S. 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 nonconformant entries with the following:

```
var badZips = $.grep(
                  originalArray,
                  function(value) {
                    return value.match(/^\d{5}(-\d{4})?$/) != null;
                  },
                  true);
```

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 array of collected values.

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 the letter *S* instead of the expected number *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`!

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 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 isn't 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 an *array-like object* 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

`$.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 behalf. 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, isn't 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. It's 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];
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]`.

Now that we've seen how jQuery helps us to easily work with arrays, let's see how 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 copying, but as with so many other procedures, jQuery anticipates this need and provides a ready-made utility function to help us out: `$.extend()`. As we'll see in the next chapter, this function is useful for much more than extending an object. 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

`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 names exist in the sources, sources later in the argument list override those earlier in the list.

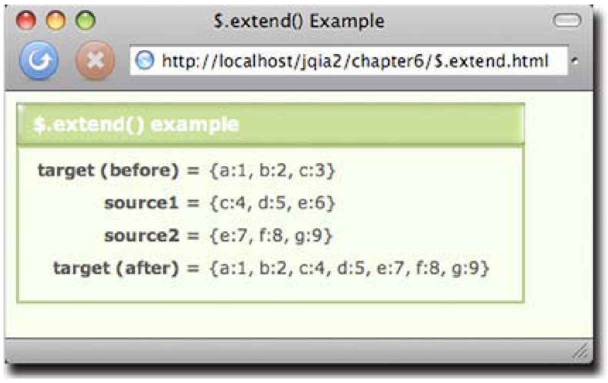
Returns

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 };
```



```
target (before) = {a:1, b:2, c:3}
        source1 = {c:4, d:5, e:6}
        source2 = {e:7, f:8, g:9}
 target (after) = {a:1, b:2, c:4, d:5, e:7, f:8, g:9}
```

Figure 6.3 The `$.extend()` function merges properties from multiple source objects without duplicates, and gives precedence to instances in reverse order of specification.

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 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.

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` overrides 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, highly interactive application, submitting 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 formats the request body containing the request parameters on our behalf. 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 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 the 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. Although JavaScript provides a handy function for that (`encodeURIComponent()`), the formatting of the query string falls squarely into our laps.

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,traditional)`

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.
`traditional`	(Boolean) An optional flag that forces this function to perform the serialization using the same algorithm used prior to jQuery 1.4. This generally only affects source objects with nested objects. See the sections that follow for more details. If omitted, defaults to `false`.

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 contains elements other than those representing form values, you'll end up with a bunch of entries such as

```
&undefined=undefined
```

in the resulting string, because this function doesn't 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 on to your hat. In chapter 8, when we start talking about Ajax, we'll see that form elements aren't always submitted by their forms!

But that's not going to be an issue, because we'll also see later on that jQuery provides a higher-level means (that internally uses this very utility function) to handle this sort of thing in a more sophisticated fashion.

SERIALIZING NESTED PARAMETERS

Trained by years of dealing with the limitations of HTTP and HTML form controls, web developers are conditioned to think of serialized parameters, aka query strings, as a flat list of name/value pairs.

For example, imagine a form in which we collect someone's name and address. The query parameters for such a form might contain names such as `firstName`, `lastName` and `city`. The serialized version of the query string might be:

```
firstName=Yogi&lastName=Bear&streetAddress=123+Anywhere+Lane
  ➥ &city=Austin&state=TX&postalCode=78701
```

The pre-serialized version of this construct would be:

```
{
  firstName: 'Yogi',
  lastName: 'Bear',
  streetAddress: '123 Anywhere Lane',
  city: 'Austin',
  state: 'TX',
  postalCode : '78701'
}
```

As an object, that doesn't really represent the way that we'd think about such data. From a data organization point of view, we might think of this data as two major elements, a name and an address, each with their own properties. Perhaps something along the lines of:

```
{
  name: {
    first: 'Yogi',
    last: 'Bear'
  },
  address: {
    street: '123 Anywhere Lane',
    city: 'Austin',
    state: 'TX',
    postalCode : '78701'
  }
}
```

But this nested version of the element, though more logically structured than the flat version, doesn't easily lend itself to conversion to a query string.

Or does it?

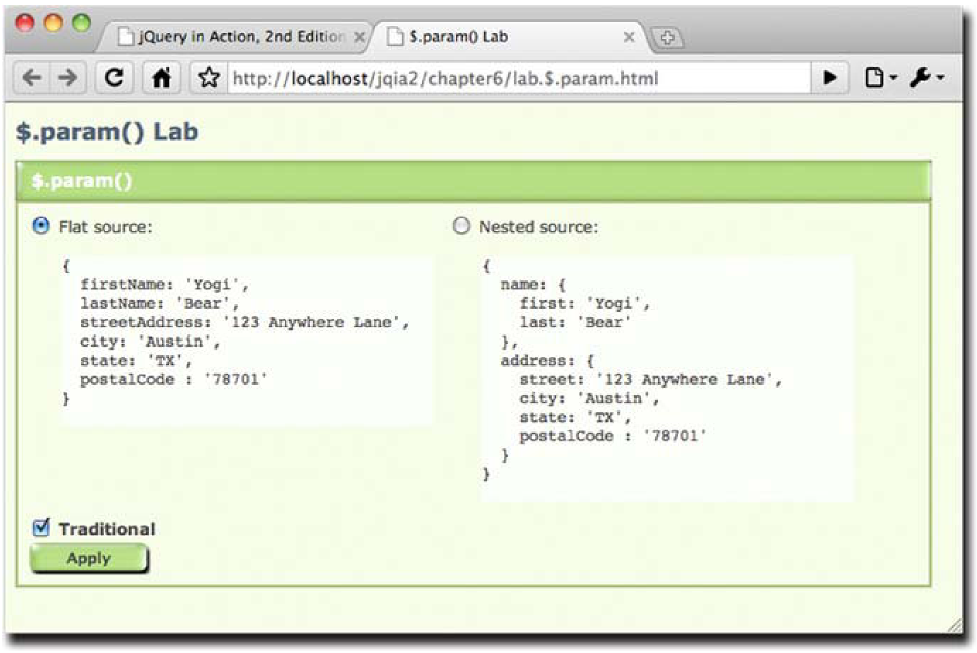
By using a conventional notation employing square brackets, such a construct could be expressed as the following:

```
name[first]=Yogi&name[last]=Bear&address[street]=123+Anywhere+Lane
  ➥ &address[city]=Austin&address[state]=TX&address[postalCode]=78701
```

In this notation, sub-properties are expressed using square brackets to keep track of the structure. Many server-side frameworks such as RoR (Ruby on Rails) and PHP can handily decode these strings. Java doesn't have any native facility to reconstruct a nested object from such notation, but such a processor would be pretty easy to build.

This is new behavior in jQuery 1.4—older versions of jQuery's `$.param()` function did not produce anything meaningful when passed a nested construct. If we want to cause `$.param()` to exhibit the older behavior, the `traditional` parameter should be passed as `true`.

We can prove this to ourselves with the `$.param()` Lab Page provided in file chapter6/lab.$.param.html, and shown in figure 6.4.


jQuery in Action, 2nd Edition
$.param() Lab
http://localhost/jqia2/chapter6/lab.$.param.html
$.param() Lab
$.param()
Flat source:
Nested source:
Traditional
Apply

```
{
  firstName: 'Yogi',
  lastName: 'Bear',
  streetAddress: '123 Anywhere Lane',
  city: 'Austin',
  state: 'TX',
  postalCode : '78701'
}
```

```
{
  name: {
    first: 'Yogi',
    last: 'Bear'
  },
  address: {
    street: '123 Anywhere Lane',
    city: 'Austin',
    state: 'TX',
    postalCode : '78701'
  }
}
```

Figure 6.4 The `$.param()` Lab lets us see how flat and nested objects are serialized using the new and traditional algorithms.

This Lab lets us see how `$.param()` will serialize flat and nested objects, using its new algorithm, as well as the traditional algorithm.

Go ahead and play around with this Lab until you feel comfortable with the action of the function. We even urge you to make a copy of the page and play around with different object structures that you might want to serialize.

6.3.8 *Testing objects*

You may have noticed that many of the jQuery wrapper methods and utility functions have rather malleable parameter lists; optional parameters can be omitted without the need to include `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 simply 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 there are only two parameters, and that a function is passed as the second parameter, it interprets that as the handler rather than as a data argument.

Testing parameters for various types, including whether they are functions or not, will certainly come in handy if we want to create our own functions and methods that are similarly friendly and versatile, so jQuery exposes a number of testing utility functions, as outlined in table 6.4.

Table 6.4 jQuery offers utility functions for testing objects

Function	Description
`$.isArray(o)`	Returns `true` if `o` is a JavaScript array (but not if `o` is any other array-like object like a jQuery wrapped set).
`$.isEmptyObject(o)`	Returns `true` if `o` is a JavaScript object with no properties, including any inherited from `prototype`.
`$.isFunction(o)`	Returns `true` if `o` is a JavaScript function. Warning: in Internet Explorer, built-in functions such as `alert()` and `confirm()`, as well as element methods are not correctly reported as functions.
`$.isPlainObject(o)`	Returns `true` if `o` is a JavaScript object created via `{}` or `new Object()`.
`$.isXMLDoc(node)`	Returns `true` if `node` is an XML document, or a node within an XML document.

Now let's look at a handful of miscellaneous utility functions that don't really fit into any one category.

6.4 Miscellaneous utility functions

This section will explore the set of utility functions that each pretty much define their own category. We'll start with one that doesn't seem to do very much at all.

6.4.1 Doing nothing

jQuery defines a utility function that does nothing at all. This function could have been named `$.wastingAwayAgainInMargaritaville()`, but that's a tad long so it's named `$.noop()`. It's defined with the following syntax:

Function syntax: $.noop

`$.noop()`
Does nothing.

Parameters
`none`

Returns
Nothing.

Hmmm, a function that is passed nothing, does nothing, and returns nothing. What's the point?

Recall how many jQuery methods are passed parameters, or option values, that are optional function callbacks? `$.noop()` serves as a handy default for those callbacks when the user does not supply one.

6.4.2 Testing for containment

When we want to test one element for containment within another, jQuery provides the `$.contains()` utility function:

Function syntax: $.contains

`$.contains(container,containee)`
Tests if one element is contained within another with the DOM hierarchy.

Parameters

`container`	(Element) The DOM element being tested as containing another element.
`containee`	(Element) The DOM element being tested for containment.

Returns
`true` if the `containee` is contained within the `container`; `false` otherwise.

Hey, wait a minute! Doesn't this sound familiar? Indeed, we discussed the `has()` method back in chapter 2, to which this function bears a striking resemblance.

This function, used frequently internally to jQuery, is most useful when we already have references to the DOM elements to be tested, and there's no need to take on the overhead of creating a wrapped set.

Let's look at another function that closely resembles its wrapper-method equivalent.

6.4.3 Tacking data onto elements

Back in chapter 3, we examined the `data()` method, which allows us to assign data to DOM elements. For those cases where we already have a DOM element reference, we can use the low-level utility function `$.data()` to perform the same action:

Function syntax: $.data

`$.data(element,name,value)`
Stores or retrieves data on the passed element using the specified name.

Parameters

`element`	(Element) The DOM element upon which the data is to be established, or from which the data is to be retrieved.
`name`	(String) The name with which the data is associated.
`value`	(Object) The data to be assigned to the element with the given name. If omitted, the named data is retrieved.

Returns
The data value that was stored or retrieved.

As might be expected, we can also remove the data via a utility function:

Function syntax: $.removeData

`$.removeData(element,name)`
Removes data stored on the passed element.

Parameters

`element`	(Element) The DOM element from which the data is to be removed.
`name`	(String) The name of the data item to be removed. If omitted, all stored data is removed.

Returns
Nothing.

Now let's turn our attention to one of the more esoteric utility functions—one that lets us have a pronounced effect on how event listeners are called.

6.4.4 Prebinding function contexts

As we've seen throughout our examination of jQuery, functions and their contexts play an important role in jQuery-using code. In the coming chapters on Ajax (chapter 8) and jQuery UI (chapters 9 through 11), we'll see an even stronger emphasis on functions, particularly when used as callbacks.

The contexts of functions—what's pointed to by `this`—is determined by how the function is invoked (see the appendix if you want to review this concept). When we want to call a particular function and explicitly control what the function context will be, we can use the `Function.call()` method to invoke the function.

But what if we're not the ones calling the function? What if, for example, the function is a callback? In that case, we're not the ones invoking the function so we can't use `Function.call()` to affect the setting of the function context.

jQuery gives us a utility function by which we can prebind an object to a function such that when the function is invoked, the bound object will become the function context. This utility function is named `$.proxy()`, and its syntax is as follows:

Function syntax: $.proxy

`$.proxy(function,proxy)`
`$.proxy(proxy,property)`

Creates a copy of a function with a prebound proxy object to serve as the function context when the function is invoked as a callback.

Parameters

`function`	(Function) The function to be prebound with the proxy object.
`proxy`	(Object) The object to be bound as the proxy function context.
`property`	(String) The name of the property within the object passed as `proxy` that contains the function to be bound.

Returns

The new function prebound with the proxy object.

Bring up the example in file chapter6/$.proxy.html. You'll see a display as shown in figure 6.5.

In this example page, a Test button is created within a `<div>` element with an `id` value of `buttonContainer`. When the Normal radio button is clicked, a click handler is established on the button and its container as follows:

```
$('#testButton,#buttonContainer').click(
  function(){ say(this.id); }
);
```

When the button is clicked, we'd expect the established handler to be invoked on the button and, because of event bubbling, on its parent container. In each case, the function context of the invocation should be the element upon which the handler was established.

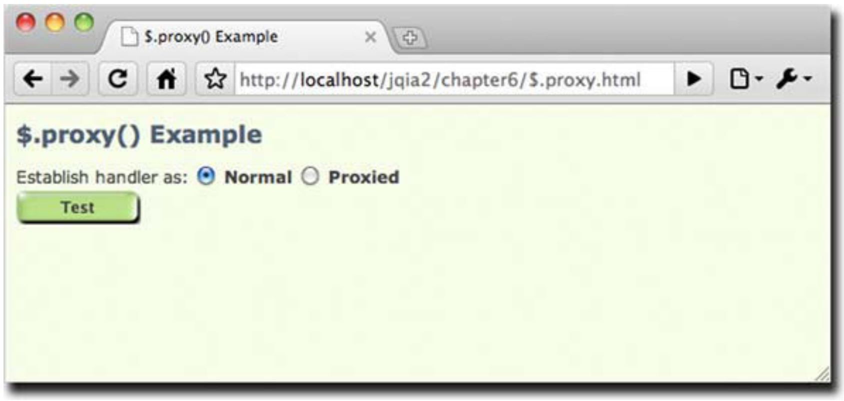

$.proxy() Example
http://localhost/jqia2/chapter6/$.proxy.html
$.proxy() Example
Establish handler as: Normal Proxied
Test

Figure 6.5 **The `$.proxy` example page will help us see the difference between normal and proxied callbacks.**

The results of the call to `say(this.id)` within the handler (which reports the `id` property of the function context) show that all is as expected—see the top portion of figure 6.6. The handler is invoked twice: first on the button and then on the container, with each element respectively set as the function context.

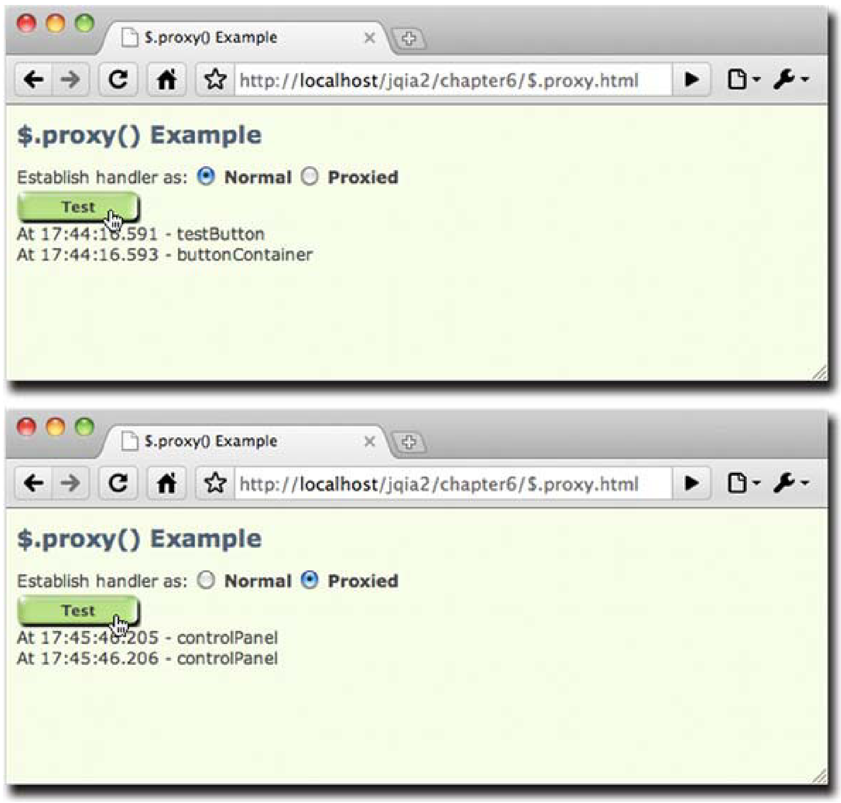
However, when the Proxy radio button is checked, the handler is established as follows:

```
$('#testButton,#buttonContainer').click(
  $.proxy(function(){ say(this.id); }, $('#controlPanel')[0])
);
```

This establishes the same handler as before, *except* that the handler function has been passed through the `$.proxy()` utility function, prebinding an object to the handler.

In this case, we bound the element with the `id` of `controlPanel`. The bound object does not *have* to be an element—in fact, most often it won't be. We just chose it for this example because it makes the object easy to identify via its `id` value.

Now when we click the Test button, we see the display at the bottom of figure 6.6, showing that the function context has been forced to be the object that we bound to the handler with `$.proxy()`.



Establish handler as: [✓ Normal] Proxied

```
At 17:44:16.591 - testButton
At 17:44:16.593 - buttonContainer
```

Establish handler as: Normal [✓ Proxied]

```
At 17:45:46.205 - controlPanel
At 17:45:46.206 - controlPanel
```

Figure 6.6 This example shows the effects of prebinding an object to the click handler for the Test button.

This ability is really useful for providing data to a callback that it might not normally have access to via closures or other means.

The most common use case for `$.proxy()` is when we want to bind a method of an object as a handler, and have the method's owning object established as the handler's function context exactly as if we had called the method directly. Consider an object such as this:

```
var o = {
  id: 'o',
  hello: function() { alert("Hi there! I'm " + this.id); }
};
```

If we were to call the `hello()` method via `o.hello()`, the function context (`this`) would be `o`. But if we establish the function as a handler like so,

```
$(whatever).click(o.hello);
```

we find that the function context is the current bubbling element, not `o`. And if our handler relies upon `o`, we're rather screwed.

We can unwedge ourselves by using `$.proxy()` to force the function context to be `o` with one of these two statements:

```
$(whatever).click($.proxy(o.hello,o));
```
or
```
$(whatever).click($.proxy(o,'hello'));
```

Be aware that going this route means that you will not have any way of knowing the current bubble element of the event propagation—the value normally established as the function context.

6.4.5 *Parsing JSON*

JSON has fast become an Internet darling child, threatening to push the more stodgy-seeming XML off the interchange-format pedestal. As most JSON is also valid JavaScript expression syntax, the JavaScript `eval()` function has long been used to convert a JSON string to its JavaScript equivalent.

Modern browsers provide `JSON.parse()` to parse JSON, but not everyone has the luxury of assuming that all of their users will be running the latest and greatest. Understanding this, jQuery provides the `$.parseJSON()` utility function.

Function syntax: $.parseJSON

`$.parseJSON(json)`
Parses the passed JSON string, returning its evaluation.

Parameters

`json`	(String) The JSON string to be parsed.

Returns
The evaluation of the JSON string.

When the browser supports `JSON.parse()`, jQuery will use it. Otherwise, it will use a JavaScript trick to perform the evaluation.

Bear in mind that the JSON string must be completely *well-formed*, and that the rules for well-formed JSON are much more strict than JavaScript expression notation. For example, all property names must be delimited by double-quote characters, even if they form valid identifiers. And that's *double*-quote characters—single-quote characters won't cut it. Invalid JSON will result in an error being thrown. See http://www.json.org/ for the nitty-gritty on well-formed JSON.

Speaking of evaluations ...

6.4.6 *Evaluating expressions*

While the use of `eval()` is derided by some Internet illuminati, there are times when it's quite useful.

But `eval()` executes in the current context. When writing plugins and other reusable scripts, we might want to ensure that the evaluation always takes place in the global context. Enter the `$.globalEval()` utility function.

Function syntax: $.globalEval

`$.globalEval(code)`

Evaluates the passed JavaScript code in the global context.

Parameters

`code`	(String) The JavaScript code to be evaluated.

Returns

The evaluation of the JavaScript code.

Let's wrap up our investigation of the utility functions with one that we can use to dynamically load new scripts into our pages.

6.4.7 *Dynamically loading scripts*

Most of the time, we'll load the external scripts that 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 a 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, and we don't want to include the script unless it's 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 scripts 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.

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-1.4.js"></script>
    <script type="text/javascript"
            src="../scripts/jqia2.support.js"></script>
    <script type="text/javascript">
      $(function(){
        $('#loadButton').click(function(){
          $.getScript(
            'new.stuff.js'
            //,function(){$('#inspectButton').click()}
          );
        });
        $('#inspectButton').click(function(){
          someFunction(someVariable);
        });
      });
    </script>
  </head>

  <body>
    <button type="button" id="loadButton">Load</button>
    <button type="button" id="inspectButton">Inspect</button>
  </body>
</html>
```

1 Fetches script on clicking Load button

2 Displays result on clicking Inspect button

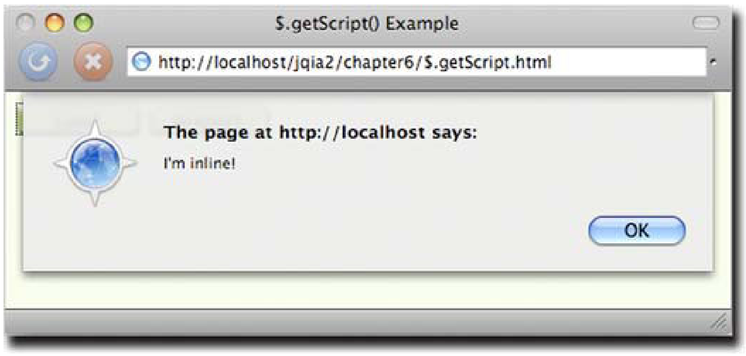
3 Contains test buttons

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 6.7.

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 6.8, we know that both the variable and function are loaded correctly.

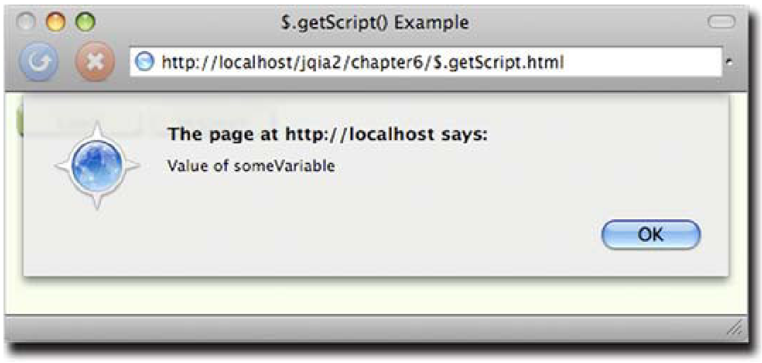
If you're still running Safari 2 or older (which is very out-of-date at this point) and would like to observe the behavior of older versions of Safari that we warned you about earlier, make a copy of the HTML file for the page shown in figure 6.8, and



The page at http://localhost says:

I'm inline!

Figure 6.7 The dynamic loading and evaluation of the script file results in the inline alert statement being executed.



The page at http://localhost says:
Value of someVariable

Figure 6.8 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.

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 2, the function and variable loaded dynamically from the script are available within the callback function. But when executed on Safari 2, 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 about the capabilities of the containing browser using the various flags in the `$.support` object. When we need to resort to browser detection to account for differences in browser capabilities and operations beyond what `$.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 the data of a source array using whatever filtering criteria we'd 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()`.

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.

We also saw a bevy of functions for testing objects to see if they're functions, JavaScript objects, or even empty objects—useful for many situations, but particularly when inspecting variable argument lists.

The `$.proxy()` method can be used to prebind an object to later be used as the function context for an event handler invocation, and the `$.noop()` function can be used to do nothing at all!

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 adding our own extensions to jQuery. Let's get to it in the next chapter.