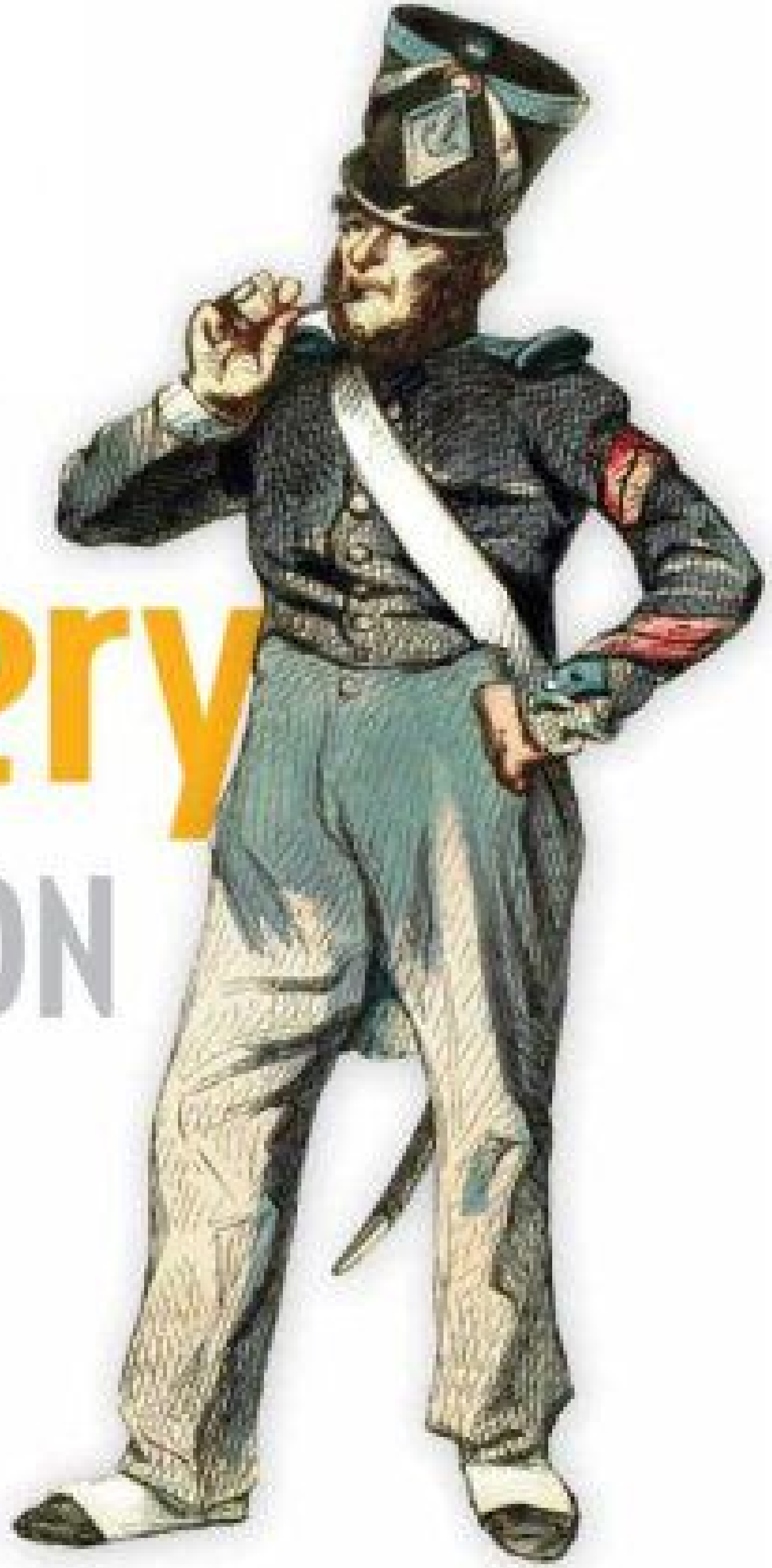
Covers jQuery 1.4 and jQuery UI 1.8

Bear Bibeault
Yehuda Katz

jQuery IN ACTION

SECOND EDITION



MANNING

Table of Contents

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'll 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 jQuery?

If you've been paying attention while reading through this book, and you've been reviewing the code examples presented within it, you undoubtedly will have noted that adopting jQuery for use in our pages has a profound effect on how script is written within a page.

jQuery promotes a certain style for a page's code: generally 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 benefits 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 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), 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 guidelines …

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 Electrical Band, 1971

Although the Five Man Electrical 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.

So it is with the rules—which are more like common-sense guidelines—governing how to successfully extend jQuery with our own plugin code. These guidelines help us 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 (what we've been calling 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 1950s, in which multiple contestants claimed to be the same person with the same name, and a panel of celebrities was tasked with determining which of the contestants was in reality the person they all claimed to be. Although fun for a television audience, name collisions aren't fun at all when it comes to programming.

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 don't 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.
- Optionally, include the major and minor version numbers 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 could be jquery.fred-1.0.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.

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://plugins.jquery.com/, 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 subprefix 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, which 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 `$`?

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 `$` alias in place of `jQuery`. But when writing plugins that may end up in other people's pages, we can't be quite so cavalier. As plugin authors, we have no way of knowing whether a web developer intends to use the `$.noConflict()` function to allow the `$` alias to be used 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 web developers and ourselves as plugin authors) when 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) {
```

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 `null`s. 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. Web developers using this function are forced to carefully keep track of counting `null`s and the order of the parameters; plus, the code is difficult to read and understand.

But short of not allowing the caller so many options, 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});
```

The second would be as follows:

```
complex(valueA, {
  p3: valueC,
  p4: valueD,
  p7: valueB
});
```

Much better!

We don't have to account for omitted parameters with placeholder `null`s, and we also don't need to count parameters; each optional parameter is conveniently labeled so that it's clear exactly what it represents (when we use better parameter names than `p1` through `p7`, that is).

> **NOTE** Some APIs follow this convention of bundling optional parameters into a single `options` parameter (leaving required parameters as standalone parameters), while others bundle the complete set of parameters, required and optional alike, into a single object. Neither approach is deemed more correct than the other, so choose whichever best suits your code.

Although this is obviously a great advantage to the caller of our complex functions, what about the ramifications for *us* as the plugin 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 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 by the web developer in the `options` parameter with an object containing all the available options with their default values, the `settings` variable ends up with the default values superseded by any explicit values specified by the web developer.

> **TIP** Rather than creating a new `settings` variable, we could also just use the `options` reference itself to accumulate the values. That would cut down on one reference on the stack, but let's keep on the side of clarity for the moment.

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 look at how we extend jQuery with our own utility functions.

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 `$`). These functions aren't intended to operate on DOM elements—that's the job of methods defined to operate on a jQuery wrapped set—but to either operate on non-element JavaScript objects or perform some other operation that doesn't specifically operate on any objects. Some examples we've seen of each of these types of function are `$.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 `$`? What if some developer includes this function on a page that uses Prototype and has called `$.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 reassign the `$`? It's a good idea to err on the side of caution.

One way to ensure that someone stomping on `$` doesn't also stomp on us is to avoid using `$` 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 it 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 `$` when declaring and defining the function. Should the function ever need to become more complex, we could extend and modify it without wondering whether it's safe to use the `$` or not.

With this pattern fresh in our minds, let's implement a nontrivial 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 the 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 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.
`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();
   fill = fill || '0';                              ❶ Assigns default value
   var padding = length - result.length;            ❷ Computes padding
   if (padding < 0) {
     result = result.substr(-padding);              ❸ Truncates if necessary
   }
   else {
     for (var n = 0; n < padding; n++)
       result = fill + result;                      ❹ Pads result
   }
   return result;                    ❺ Returns final result
 };
})(jQuery);
```

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` ❶. Then, we compute the amount of padding needed ❷.

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 ❸; otherwise, we pad the beginning of the result with the appropriate number of fill characters ❹ prior to returning it as the result of the function ❺.

Namespacing utility functions

If you want to make sure that your 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'd do the following:

```
$.jQiaDateFormatter = {};
$.jQiaDateFormatter.toFixedWidth = function(value,length,fill) {...};
```

This ensures 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 because we're focusing 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?
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 of the month as a 0-filled, 2-character field
d: the day of the month
EEEE: the full name of the day of the week
EEE: the abbreviated name of the day of the week
a: the meridian (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'll use this implementation to point out some interesting tactics that we can use when creating a somewhat complex utility function.

Listing 7.2 Implementation of the `$.formatDate()` custom utility function

```
(function($){
  $.formatDate = function(date,pattern) {
    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);
      }
```

❶ **Implements main body of the function**

Chapter 7. Expand your reach by extending jQuery. jQuery in Action, Second Edition, ISBN: 9781935182320
Prepared for Prem.Satgunam@flagstar.com, Prem Satgunam

```
    }
    return result.join('');
  };
```

2 Defines the regular expression

```
  $.formatDate.patternParts =
    /^(yy(yy)?|M(M(M(M)?)?)?|d(d)?|EEE(E)?|a|H(H)?|h(h)?|m(m)?|s(s)?|S)/;
```

3 Provides name of the months

```
  $.formatDate.monthNames = [
    'January','February','March','April','May','June','July',
    'August','September','October','November','December'
  ];
```

4 Provides name of the days

```
  $.formatDate.dayNames = [
    'Sunday','Monday','Tuesday','Wednesday','Thursday','Friday',
    'Saturday'
  ];
```

5 Collects token-to-value translation functions

```
  $.formatDate.patternValue = {
    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();
      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);
```

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 ❶ needs some ancillary data to do its job. In particular,

- A regular expression used to match tokens in the pattern ❷
- A list of the English names of the months ❸
- A list of the English names of the days ❹
- A set of subfunctions designed to provide the value for each token type, given a source date ❺

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 JavaScript 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.
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 function 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; 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 the `each()` jQuery method 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()`.

In the previous example, we applied 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
      // DOM 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.

Iteration via methods that accept functions for values

Note that the "blue or red" example is a tad contrived to show how `each()` can be used to traverse the individual elements in the wrapped set. Because the `css()` method accepts a function for its value (which automatically iterates over the elements), the astute among you might have noted that this custom method could also have been written without `each()` as follows:

```
(function($){
  $.fn.makeItBlueOrRed = function() {
    return this.css('color' function() {
      return $(this).is('[id]') ? 'blue' : 'red';
    });
  };
})(jQuery);
```

This is a common idiom across the jQuery API; when a function can be passed in place of a value, the function is invoked in an iterative fashion over the elements of the wrapped set.

The variant of the example using `each()` is illustrative of cases where there's no such automatic iteration of elements.

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 percent if not read-only, 50 percent 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.

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 it exhibits the following key concepts:

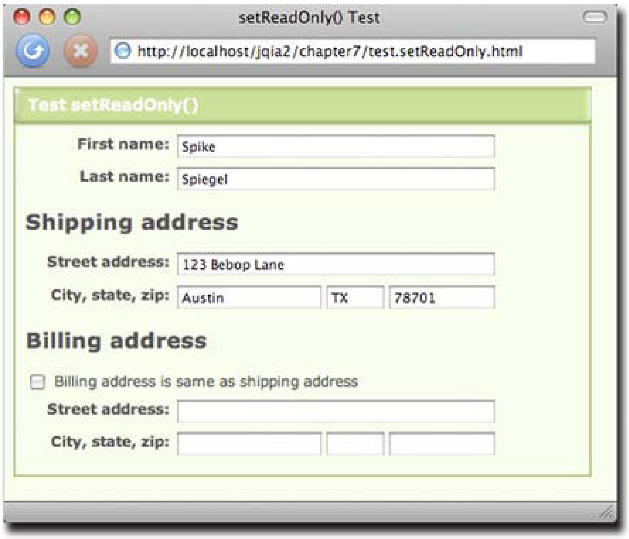
- A parameter is passed that affects how the method operates.
- Four jQuery methods are applied to the wrapped set by use of jQuery chaining.
- The new method can participate in a jQuery chain because it returns the wrapped set as its value.
- The `filter()` method is used to ensure that, no matter what set of wrapped elements the web developer applies 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, and 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 checkbox to the billing address that indicates whether the billing address is the same as the shipping address. When this box is checked, the



setReadOnly() Test

http://localhost/jqia2/chapter7/test.setReadOnly.html

Test setReadOnly()

First name: Spike

Last name: Spiegel

Shipping address

Street address: 123 Bebop Lane

City, state, zip: Austin TX 78701

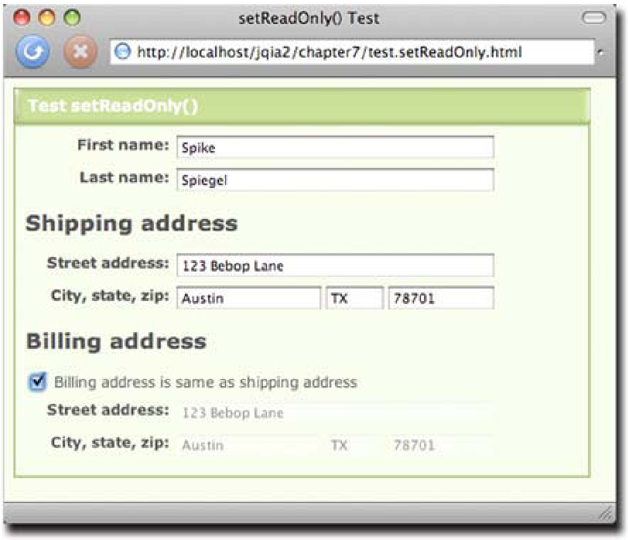
Billing address

□ Billing address is same as shipping address

Street address:

City, state, zip:

Figure 7.1a Our form for testing the `setReadOnly()` custom wrapper method before checking the checkbox



setReadOnly() Test

http://localhost/jqia2/chapter7/test.setReadOnly.html

Test setReadOnly()

First name: Spike

Last name: Spiegel

Shipping address

Street address: 123 Bebop Lane

City, state, zip: Austin TX 78701

Billing address

☑ Billing address is same as shipping address

Street address: 123 Bebop Lane

City, state, zip: Austin TX 78701

Figure 7.1b Our form for testing the `setReadOnly()` custom wrapper method after checking the checkbox, showing the results of applying the custom method

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 shows the after state.

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 test page for the `setReadOnly()` wrapper method

```
<!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-1.4.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>

<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 checkbox in the ready handler. When the state of the checkbox is changed by a click, we do three things:

1. Copy the checked state into the variable `same` for easy reference in the remainder of the listener.
2. Set the values of the billing address fields. If they're 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.

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 checkbox too. The checkbox 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 web developer to accept the ability to affect only the opacity? How would you modify the method to allow the developer 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. 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 web developer to whip up a quick slideshow page. We'll create a jQuery plugin, which we'll name *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.

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 for moving through the slideshow manually, and for starting and stopping the automatic slideshow

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

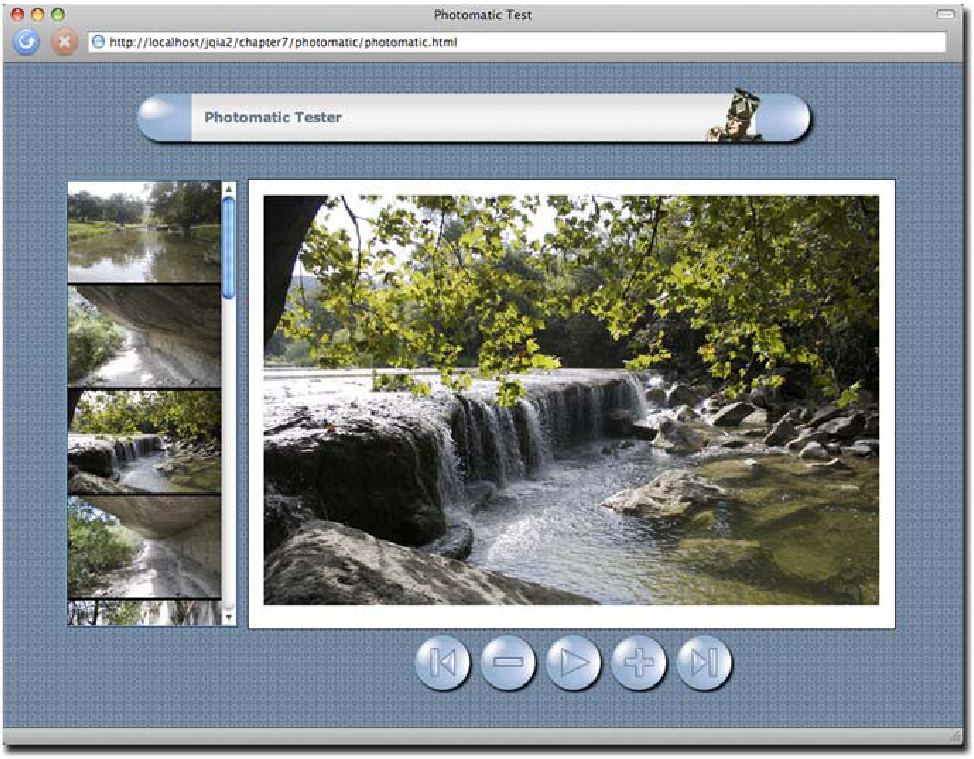


Figure 7.2 The test page that we'll use to put our Photomatic plugin through its paces

  - 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 past 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 web developers as much freedom for layout and styling as possible; we'll define our plugin so that developers can set up the elements in any manner they like and then tell us which page element should be used for each purpose. Furthermore, in order to give web developers as much leeway as possible, we'll define our plugin so that they can provide any wrapped set of images to serve as thumbnails. Usually, thumbnails will be gathered together as in our test page, but developers 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 `options` hash, to operate as Photomatic controls.

Parameters

`options`	(Object) An object hash that specifies the options for Photomatic. See table 7.1 for details.

Returns

The wrapped set.

Because we have a nontrivial number of parameters for controlling the operation of Photomatic (many of which can be omitted), we utilize an object hash to pass them, as was discussed in section 7.2.3. The possible options that we can specify are shown in table 7.1.

Table 7.1 The options for the Photomatic custom plugin method

Option name	Description
`firstControl`	(Selector|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`	(Selector|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`	(Selector|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.

Table 7.1 The options for the Photomatic custom plugin method *(continued)*

Option name	Description
`photoElement`	(Selector|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 `img.photomaticPhoto`.
`playControl`	(Selector|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`	(Selector| 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.
`delay`	(Number) The interval between transitions for an automatic slideshow, in milliseconds. Defaults to 3000.

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 in 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-1.4.js"></script>
    <script type="text/javascript"
            src="jquery.jqia.photomatic.js"></script>
    <script type="text/javascript">
      $(function(){
        $('#thumbnailsPane img').photomatic({      ❶ Invokes the Photomatic plugin
          photoElement: '#photoDisplay',
          previousControl: '#previousButton',
          nextControl: '#nextButton',
          firstControl: '#firstButton',
          lastControl: '#lastButton',
          playControl: '#playButton',
          delay: 1000
        });
      });
    </script>
  </head>
  <body class="fancy">

    <div id="pageContainer">
      <div id="pageContent">

        <h1>Photomatic Tester</h1>

        <div id="thumbnailsPane">
          <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=""/>
        </div>

        <div id="buttonBar">
          <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>
```

2 Contains thumbnail images

3 Defines image element for full-sized photos

4 Contains elements to serve as controls

If that looks simpler than you thought it would, you shouldn't be surprised. 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 sourceless) 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.

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 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 described in 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 (which 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 defaults supplied by the inline hash object overridden with any values supplied by the caller. Although 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 for our plugin to know what concepts like *next* relative image and *previous* relative image mean, we need not only an ordered 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 developers collected in the wrapped set, so we want to filter it down to only image elements, which we can easily do 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), which we store in a property of settings that we name thumbnails$ (the trailing dollar sign being a convention that indicates a stored reference to a wrapped set).

Another piece of state that we need to keep track of is the *current* image. We'll do that by maintaining an index into the list of thumbnails by adding another property to settings named current:

```
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 (which we'll be examining shortly) 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.

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 a list of click event listeners that we'll 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 even 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 (our other goal). What could be simpler?

So we define a function named `showPhoto()`, which accepts a single parameter indicating the index of the thumbnail that's 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 full-size photo to be displayed. We chain a call to the click() method to the previous statement that references settings.thumbnails$, as follows:

```
.click(function(){
  showPhoto($(this).data('photomatic-index'));
});
```

In this handler, we 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 the 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:

```
$(settings.photoElement)
  .attr('title', 'Click for next photo')
  .css('cursor','pointer')
  .click(function(){
    showPhoto((settings.current+1) % settings.thumbnails$.length);
  });
```

We add a thoughtful title attribute to the photo so users know that clicking on the photo will progress to the next one, and we set the cursor to indicate that the element is clickable.

We then establish a click handler, in which 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.

The handlers for the First, Previous, Next, and Last controls all follow a similar pattern: figure out 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 use to accomplish that:

```
$(settings.playControl).toggle(
  function(event){
    settings.tick = window.setInterval(
      function(){
        $(settings.nextControl).triggerHandler('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 web developer 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 for 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.

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){ $(this).data('photomatic-index',n); }
      )
      .click(function(){
        showPhoto($(this).data('photomatic-index'));
      });
    $(settings.photoElement)
      .attr('title','Click for next photo')
      .css('cursor','pointer')
      .click(function(){
        showPhoto((settings.current+1) % settings.thumbnails$.length);
      });
    $(settings.nextControl).click(function(){
      showPhoto((settings.current+1) % settings.thumbnails$.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).triggerHandler('click'); },
          settings.delay);
        $(event.target).addClass('photomatic-playing');
        $(settings.nextControl).click();
      },
      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're free to add as many Photomatic widgets to a page as we like, without fear that they will interfere with one another (taking care, of course, to make sure we don't use duplicate id values in the markup).

But is it complete? You be the judge and consider the following exercises:

- Again, error checking and handling has been glossed over. How would you go about making the plugin as bulletproof as possible?
- 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.
- Going one step further, how would you go about allowing the developer to use a custom animation of his or her choice?
- 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://plugins.jquery.com/ 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 filenames, 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 `$`, and new wrapper methods are easily created as properties of `$.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 you'll 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 interactive applications practically a no-brainer.