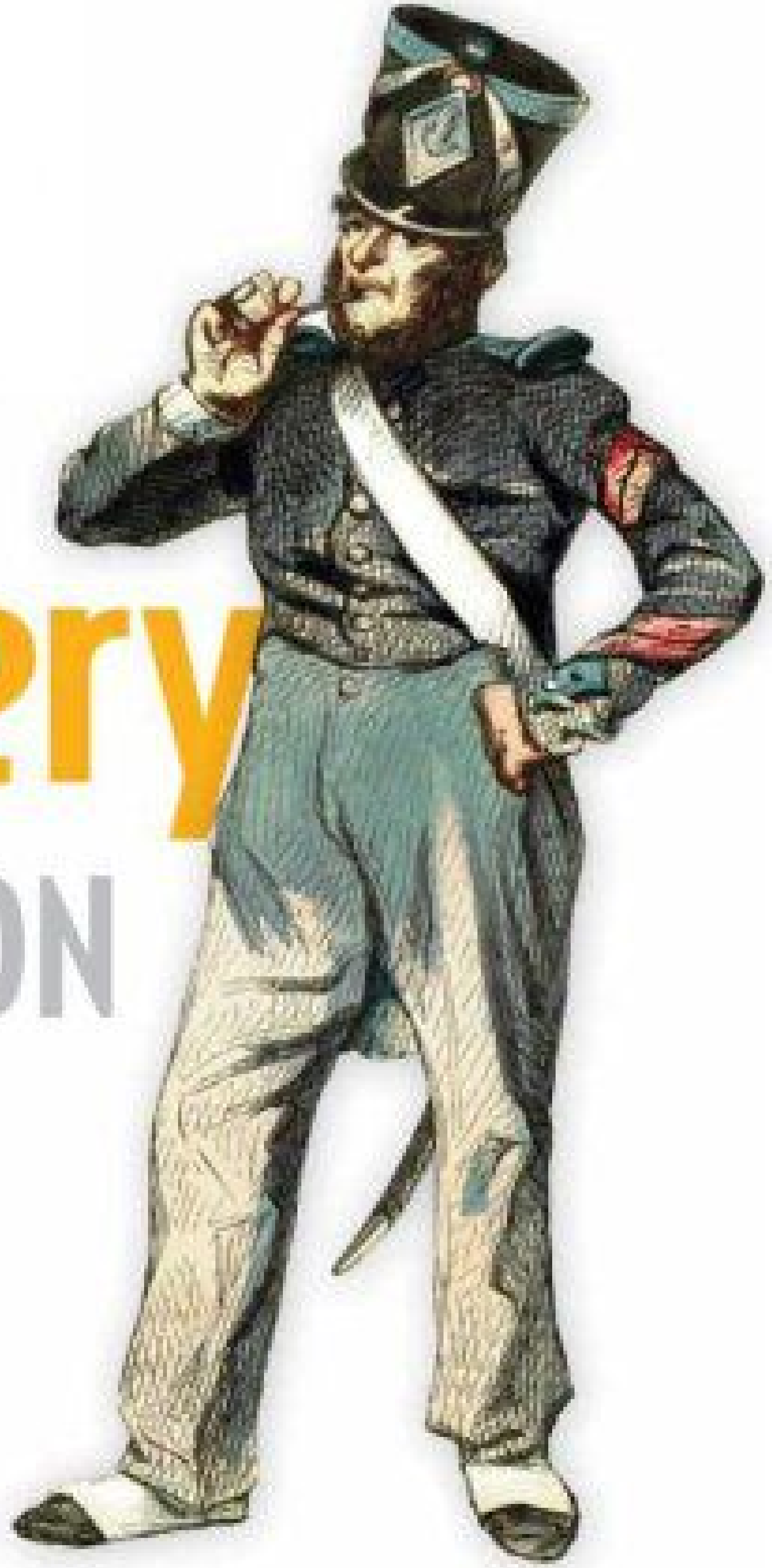
Bear Bibeault
Yehuda Katz

Covers jQuery 1.4 and jQuery UI 1.8

jQuery IN ACTION

SECOND EDITION



MANNING

Table of Contents

3 Bringing pages to life with jQuery

This chapter covers

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

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

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

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

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

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

3.1 Working with element properties and attributes

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

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

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

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

Consider the following HTML markup for an image element:

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

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

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

Figure 3.1 shows a simplified overview of this process.

There remains an active connection between the attribute values stored in the attributes list, and the corresponding properties. Changing an attribute value results in a change in the corresponding property value and vice versa. Even so, the values may not always be identical. For example, setting the src attribute of the image element to image.gif will result in the src property being set to the full absolute URL of the image.

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

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

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

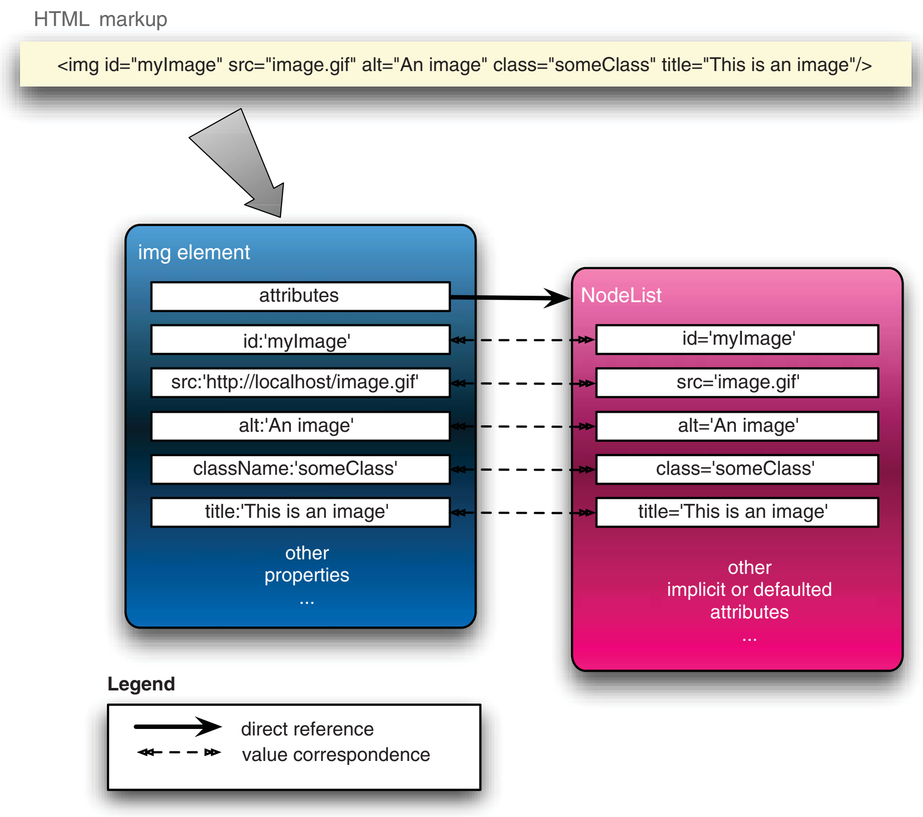


Figure 3.1 HTML markup is translated into DOM elements, including the attributes of the tag and properties created from them. The browser creates a correspondence between the attributes and properties of the elements.

3.1.1 *Manipulating element properties*

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

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

- Using array indexing on the wrapped set, as in `$(whatever)[n]`
- Using the `get()` method, which returns either an individual element by index, or `toArray()`, which returns an array of the entire set of elements
- Using the `each()` or `map()` methods, where the individual elements are made available in the callback functions
- Using the `eq()` method or `:eq()` filter
- Via callback functions to some methods (like `not()` and `filter()`, for example) that set elements as the function context of the callback

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

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

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

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

3.1.2 *Fetching attribute values*

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

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

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

Method syntax: attr

`attr(name)`

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

Parameters

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

Returns

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

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

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

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

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

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

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

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

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

Custom attributes and HTML

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

HTML 5, on the other hand, formally recognizes and allows for such custom attributes, as long as the custom attribute name is prefixed with the string `data-`. Any attributes following this naming convention will be considered valid according to HTML 5's rules; those that don't will continue to be considered invalid. (For details, see the W3C's specification for HTML 5: http://www.w3.org/TR/html5/dom.html#attr-data.)

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

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

jQuery normalized name	DOM name
`cellspacing`	`cellSpacing`
`class`	`className`
`colspan`	`colSpan`
`cssFloat`	`styleFloat` for IE, `cssFloat` for others

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

jQuery normalized name	DOM name
`float`	`styleFloat` for IE, `cssFloat` for others
`for`	`htmlFor`
`frameborder`	`frameBorder`
`maxlength`	`maxLength`
`readonly`	`readOnly`
`rowspan`	`rowSpan`
`styleFloat`	`styleFloat` for IE, `cssFloat` for others
`tabindex`	`tabIndex`
`usemap`	`useMap`

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

3.1.3 *Setting attribute values*

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

Method syntax: attr

`attr(name,value)`

Sets the named attribute to the passed value for all elements in the wrapped set.

Parameters

`name`	(String) The name of the attribute to be set.
`value`	(Any|Function) Specifies the value of the attribute. This can be any JavaScript expression that results in a value, or it can be a function. The function is invoked for each wrapped element, passing the index of the element and the current value of the named attribute. The return value of the function becomes the attribute value.

Returns

The wrapped set.

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

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

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

Consider the following statement:

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

This method will run through all elements on the page, modifying the `title` attribute of each element by appending a string (composed using the index of the element within the DOM and the `id` attribute of each specific element) to the previous value.

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

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

Method syntax: attr

`attr(attributes)`

Uses the properties and values specified by the passed object to set corresponding attributes onto all elements of the matched set.

Parameters

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

Returns

The wrapped set.

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

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

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

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

> **WARNING** Internet Explorer won't allow the `name` or `type` attributes of `<input>` elements to be changed. If you want to change the name or type of `<input>` elements in Internet Explorer, you must replace the element with a new element possessing the desired name or type. This also applies to the `value` attribute of `file` and `password` types of `<input>` elements.

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

3.1.4 Removing attributes

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

Method syntax: removeAttr

`removeAttr(name)`
Removes the specified attribute from every matched element.

Parameters

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

Returns
The wrapped set.

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

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

3.1.5 Fun with attributes

Let's see how these methods can be used to fiddle with the element attributes in various ways.

EXAMPLE #1—FORCING LINKS TO OPEN IN A NEW WINDOW

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

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

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

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

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

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

EXAMPLE #2—SOLVING THE DREADED DOUBLE-SUBMIT PROBLEM

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

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

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

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

When is "enabled" not enabling?

Don't be fooled into thinking that you can substitute the value `enabled` for `disabled` as follows:

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

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

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

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

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

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

3.1.6 Storing custom data on elements

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

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

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

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

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

jQuery comes to our rescue by providing a means to tack data onto any DOM element that we choose, in a controlled fashion, without relying upon potentially problematic expandos. We can place any arbitrary JavaScript value, even arrays and objects, onto DOM elements by use of the cleverly named `data()` method. This is the syntax:

Method syntax: data

`data(name,value)`

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

Parameters

`name`	(String) The name of the data to be stored.
`value`	(Object|Function) The value to be stored. If a function, the function is invoked for each wrapped element, passing that element as the function context. The function's returned value is used as the data value.

Returns

The wrapped set.

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

Method syntax: data

`data(name)`

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

Parameters

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

Returns

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

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

Method syntax: removeData

`removeData(name)`

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

Parameters

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

Returns

The wrapped set.

Note that it's not necessary to remove data "by hand" when removing an element from the DOM with jQuery methods. jQuery will smartly handle that for us.

The capability to tack data onto DOM elements is one that we'll see exploited to our advantage in many of the examples in upcoming chapters, but for those who have run into the usual headaches that global variables can cause, it's easy to see how storing data in-context within the element hierarchy opens up a whole new world of possibilities. In essence, the DOM tree has become a complete "namespace" hierarchy for us to employ; we're no longer limited to a single global space.

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

3.2 Changing element styling

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

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

3.2.1 *Adding and removing class names*

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

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

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

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

> **NOTE** The list of class names is considered *unordered*; that is, the order of the names within the space-delimited list has no semantic meaning.

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

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

Method syntax: addClass

`addClass(names)`
Adds the specified class name or class names to all elements in the wrapped set.

Parameters

`names`	(String|Function) Specifies the class name, or a space-delimited string of names, to be added. If a function, the function is invoked for each wrapped element, with that element set as the function context, and passing two parameters: the element index and the current class value. The function's returned value is used as the class name or names.

Returns
The wrapped set.

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

Method syntax: removeClass

`removeClass(names)`

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

Parameters

`names`	(String|Function) Specifies the class name, or a space-delimited string of names, to be removed. If a function, the function is invoked for each wrapped element, setting that element as the function context, and passing two parameters: the element index, and the class value prior to any removal. The function's returned value is used as the class name or names to be removed.

Returns

The wrapped set.

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

Method syntax: toggleClass

`toggleClass(names)`

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

Parameters

`names`	(String|Function) Specifies the class name, or a space-delimited string of names, to be toggled. If a function, the function is invoked for each wrapped element, passing that element as the function context. The function's returned value is used as the class name or names.

Returns

The wrapped set.

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

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

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

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

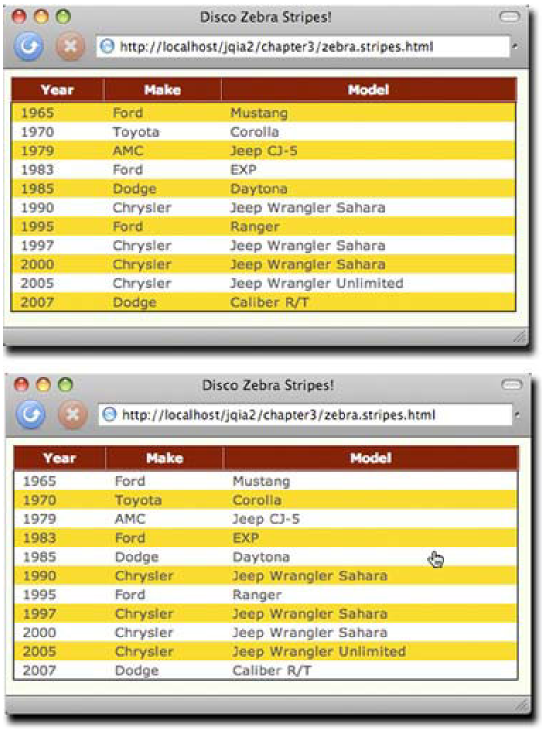
```
$(function(){/

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

The first statement in the body of this handler applies the class `striped` to every other row of the table using the `nth-child` selector that we learned about in the previous chapter. The second statement establishes event handlers for mouseover and mouseout events that both call the same `swapThem` function. We'll be learning all about event handling in the next chapter, but for now the important point is that whenever the mouse enters or leaves the table, the following line of code ends up being executed:

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

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


Disco Zebra Stripes!
http://localhost/jqia2/chapter3/zebra.stripes.html

Year	Make	Model
1965	Ford	Mustang
1970	Toyota	Corolla
1979	AMC	Jeep CJ-5
1983	Ford	EXP
1985	Dodge	Daytona
1990	Chrysler	Jeep Wrangler Sahara
1995	Ford	Ranger
1997	Chrysler	Jeep Wrangler Sahara
2000	Chrysler	Jeep Wrangler Sahara
2005	Chrysler	Jeep Wrangler Unlimited
2007	Dodge	Caliber R/T

Disco Zebra Stripes!
http://localhost/jqia2/chapter3/zebra.stripes.html

Year	Make	Model
1965	Ford	Mustang
1970	Toyota	Corolla
1979	AMC	Jeep CJ-5
1983	Ford	EXP
1985	Dodge	Daytona
1990	Chrysler	Jeep Wrangler Sahara
1995	Ford	Ranger
1997	Chrysler	Jeep Wrangler Sahara
2000	Chrysler	Jeep Wrangler Sahara
2005	Chrysler	Jeep Wrangler Unlimited
2007	Dodge	Caliber R/T

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

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

Method syntax: toggleClass

`toggleClass(names,switch)`

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

Parameters

`names`	(String|Function) Specifies the class name, or a space-delimited string of names, to be toggled. If a function, the function is invoked for each wrapped element, setting that element as the function context, and passing two parameters: the element index and the current class value. The function's returned value is used as the class name or names.
`switch`	(Boolean) A control expression whose value determines if the class will be added to the elements (if `true`) or removed (if `false`).

Returns

The wrapped set.

It's extremely common to need to determine whether an element has a particular class. We may want to conditionalize our activity based upon whether an element has a certain class or not, or we may just be using it to identify a certain type of element by class.

With jQuery, we can do that by calling the `hasClass()` method:

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

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

Method syntax: hasClass

`hasClass(name)`
Determines if any element of the matched set possesses the passed class name.

Parameters

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

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

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

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

But arguably, the `hasClass()` method makes for more readable code, and internally, `hasClass()` is more efficient.

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

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

Recall that the `attr()` method will return `undefined` if the attribute in question doesn't exist, so this statement will throw an error if the `<p>` element doesn't possess any class names.

We could solve this by first checking for the attribute, and if we wanted to wrap the entire thing in a repeatable, useful jQuery extension, we could write

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

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

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

3.2.2 *Getting and setting styles*

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

The jQuery `css()` method allows us to manipulate these styles, working in a similar fashion to the `attr()` method. We can set an individual CSS style by specifying its name and value, or a series of styles by passing in an object. First, let's look at specifying a single style name and value.

Method syntax: css

`css(name,value)`

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

Parameters

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

`value` (String|Number|Function) A string, number, or function containing the property value. If a function is passed as this parameter, it will be invoked for each element of the wrapped set, setting the element as the function context, and passing two parameters: the element index and the current value. The returned value serves as the new value for the CSS property.

Returns

The wrapped set.

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

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

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

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

Method syntax: css

`css(properties)`

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

Parameters

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

Returns

The wrapped set.

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

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

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

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

Method syntax: css

`css(name)`

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

Parameters

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

Returns

The computed value as a string.

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

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

GETTING AND SETTING DIMENSIONS

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

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

Method syntax: width and height

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

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

Parameters

`value`	(Number|String|Function) The value to be set. This can be a number of pixels, or a string specifying a value in units (such as `px`, `em`, or `%`). If no unit is specified, `px` is the default. If a function, the function is invoked for each wrapped element, passing that element as the function context. The function's returned value is used as the value.

Returns

The wrapped set.

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

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

is identical to

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

We can retrieve the width or height as follows:

Method syntax: width and height

`width()`
`height()`

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

Parameters

`none`

Returns

The computed width or height as a number in pixels.

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

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

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

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

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

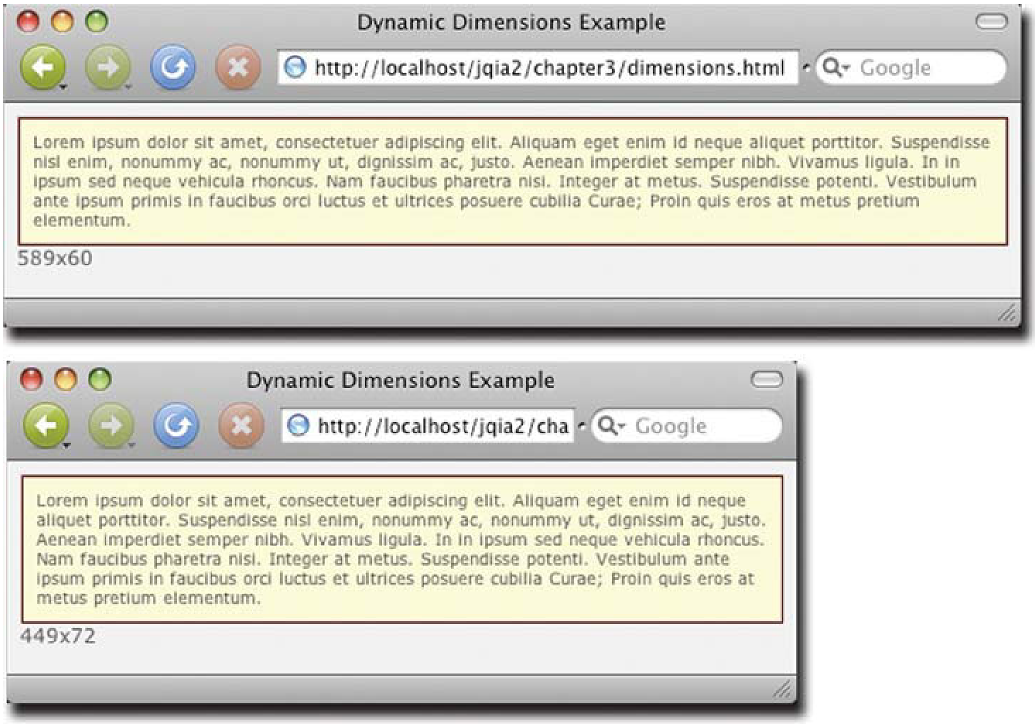
```
function displayDimensions() {
  $('#display').html(
    $('#testSubject').width()+'x'+$('#testSubject').height()
  );
```

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

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

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

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



Dynamic Dimensions Example

http://localhost/jqia2/chapter3/dimensions.html

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

589x60

Dynamic Dimensions Example

http://localhost/jqia2/cha

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

449x72

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

Listing 3.1 Dynamically tracking and displaying the dimensions of an element

```
<!DOCTYPE html>
<html>
  <head>
    <title>Dynamic Dimensions Example</title>
    <link rel="stylesheet" type="text/css" href="../styles/core.css"/>
    <style type="text/css">
      body {
        background-color: #eeeeee;
      }
      #testSubject {
        background-color: #ffffcc;
        border: 2px ridge maroon;
        padding: 8px;
        font-size: .85em;
      }
    </style>
    <script type="text/javascript"
            src="../scripts/jquery-1.4.js"></script>
    <script type="text/javascript">
      $(function(){
        $(window).resize(displayDimensions);
        displayDimensions();
      });
```

Establishes resize handler that invokes display function

Invokes reporting function in document ready handler

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

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

Displays width and height of test subject

Declares test subject with dummy text

Displays dimensions in this area

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

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

Table 3.2 Additional jQuery dimension-related methods

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

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

POSITIONS AND SCROLLING

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

The two methods use different origins from which their relative computed values are measured. One of these methods, `offset()`, returns the position relative to the document:

Method syntax: offset

`offset()`

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

Parameters

`none`

Returns

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

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

Method syntax: position

`position()`

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

Parameters

`none`

Returns

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

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

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

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

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

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

Table 3.3 The jQuery scroll control methods

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

3.3 Setting element content

When it comes to modifying the contents of elements, there's an ongoing debate regarding which technique is better: using DOM API methods or changing the elements' inner HTML.

Although use of the DOM API methods is certainly exact, it's also fairly "wordy" and results in a lot of code, much of which can be difficult to visually inspect. In most cases, modifying an element's HTML is easier and more effective, so jQuery gives us a number of methods to do so.

3.3.1 Replacing HTML or text content

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

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

Method syntax: html

`html()`
Obtains the HTML content of the first element in the matched set.

Parameters

`none`

Returns

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

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

Method syntax: html

`html(content)`
Sets the passed HTML fragment as the content of all matched elements.

Parameters

`content` (String|Function) The HTML fragment to be set as the element content. If a function, the function is invoked for each wrapped element, setting that element as the function context, and passing two parameters: the element index and the existing content. The function's returned value is used as the new content.

Returns

The wrapped set.

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

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

The statement

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

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

Method syntax: text

`text()`

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

Parameters

`none`

Returns

The concatenated string.

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

Method syntax: text

`text(content)`

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

Parameters

`content`	(String|Function) The text content to be set into the wrapped elements. Any angle bracket characters are escaped as HTML entities. If a function, the function is invoked for each wrapped element, setting that element as the function context, and passing two parameters: the element index and the existing text. The function's returned value is used as the new content.

Returns

The wrapped set.

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

3.3.2 *Moving and copying elements*

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

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

Method syntax: append

`append(content)`

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

Parameters

`content`	(String|Element|jQuery|Function) A string, element, wrapped set, or function specifying the elements of the wrapped set. If a function, the function is invoked for each wrapped element, setting that element as the function context, and passing two parameters: the element index and the previous contents. The function's returned value is used as the content.

Returns

The wrapped set.

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

Consider the following simple case:

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

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

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

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

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

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

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

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

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

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

Method syntax: prepend

`prepend(content)`

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

Parameters

`content` (String|Element|jQuery|Function) A string, element, wrapped set, or function specifying the content to append to the elements of the wrapped set. If a function, the function is invoked for each wrapped element, setting that element as the function context, and passing two parameters: the element index and the previous contents. The function's returned value is used as the content.

Returns

The wrapped set.

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

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

Method syntax: before

`before(content)`

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

Parameters

`content` (String|Element|jQuery|Function) A string, element, wrapped set, or function specifying the content to insert into the DOM before the elements of the wrapped set. If a function, the function is invoked for each wrapped element, passing that element as the function context. The function's returned value is used as the content.

Returns

The wrapped set.

Method syntax: after

`after(content)`

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

Parameters

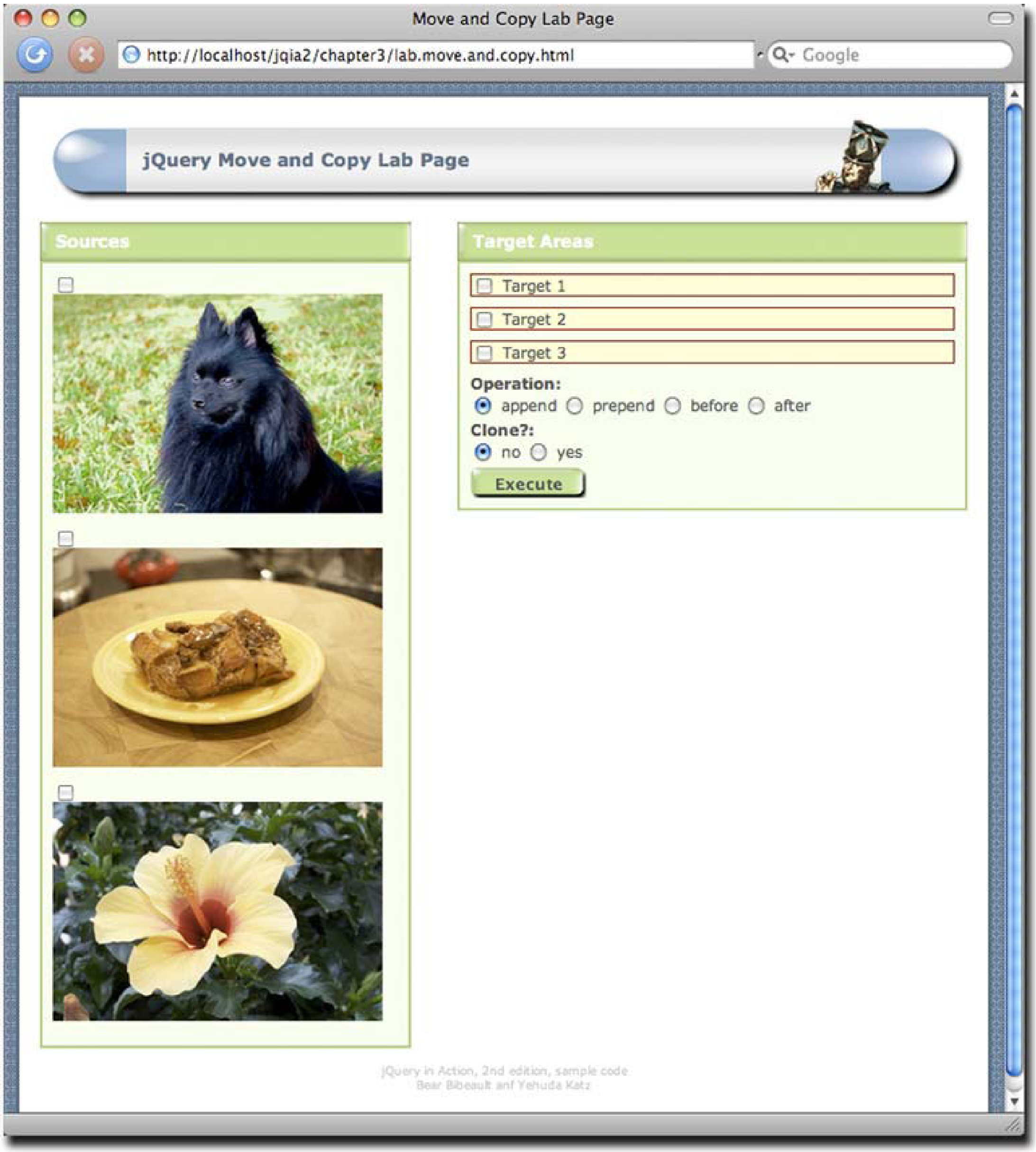
`content` (String|Element|jQuery|Function) A string, element, wrapped set, or function specifying the content to insert into the DOM after the elements of the wrapped set. If a function, the function is invoked for each wrapped element, passing that element as the function context. The function's returned value is used as the content.

Returns

The wrapped set.

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

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



jQuery Move and Copy Lab Page

Sources

□

□

□

Target Areas

□ Target 1

□ Target 2

□ Target 3

Operation:
● append ○ prepend ○ before ○ after

Clone?:
● no ○ yes

jQuery in Action, 2nd edition, sample code
Bear Bibeault anf Yehuda Katz

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

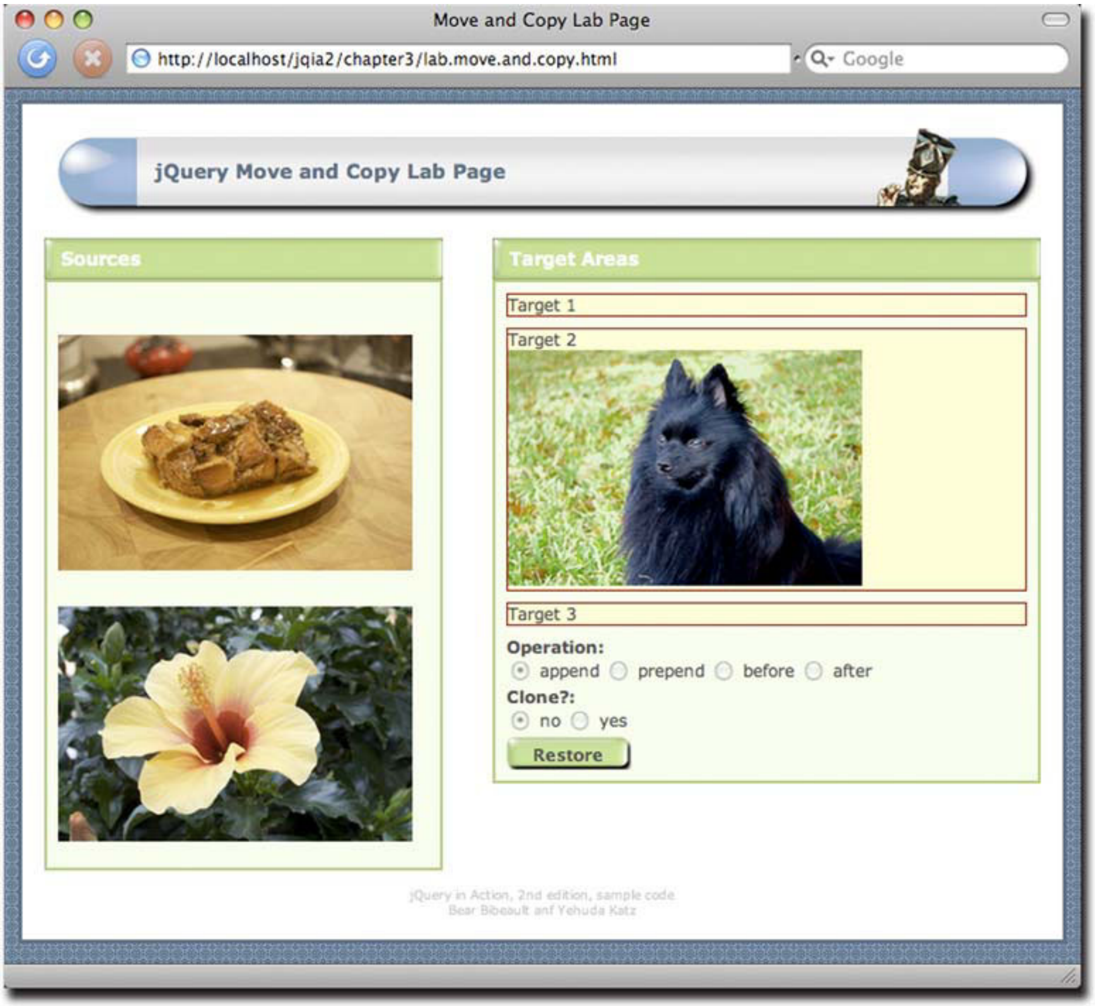
Targets for the move/copy operations are in the right pane and are also selected via checkboxes. Controls at the bottom of the pane allow us to select one of the four operations to apply: append, prepend, before, or after. (Ignore "clone" for now; we'll attend to that later.)

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

Let's run an "append" experiment.

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

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


Move and Copy Lab Page
http://localhost/jqia2/chapter3/lab.move.and.copy.html
Google
jQuery Move and Copy Lab Page
Sources
Target Areas
Target 1
Target 2
Target 3
Restore
jQuery in Action, 2nd edition, sample code
Bear Bibeault anf Yehuda Katz

Operation:
● append ○ prepend ○ before ○ after

Clone?:
● no ○ yes

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

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

Method syntax: appendTo

`appendTo(targets)`

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

Parameters

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

Returns

The wrapped set.

Method syntax: prependTo

`prependTo(targets)`

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

Parameters

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

Returns

The wrapped set.

Method syntax: insertBefore

`insertBefore(targets)`

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

Parameters

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

Returns

The wrapped set.

Method syntax: insertAfter

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

Parameters

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

Returns
The wrapped set.

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

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

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

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

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

3.3.3 Wrapping and unwrapping elements

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

Method syntax: wrap

wrap(wrapper)
Wraps the elements of the matched set with the passed HTML tags or a clone of the passed element.

Parameters

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

Returns
The wrapped set.

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

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

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

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

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

Method syntax: wrapAll

`wrapAll(wrapper)`

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

Parameters

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

Returns

The wrapped set.

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

Method syntax: wrapInner

`wrapInner(wrapper)`

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

Parameters

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

Returns

The wrapped set.

The converse operation, removing the parent of a child element, is also possible with the `unwrap()` method: :

Method syntax: unwrap

`unwrap()`

Removes the parent element of the wrapped elements. The child element, along with any siblings, replaces the parent element in the DOM.

Parameters

`none`

Returns

The wrapped set.

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

3.3.4 Removing elements

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

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

Method syntax: remove

`remove(selector)`
Removes all elements in the wrapped set from the page DOM.

Parameters

`selector`	(String) An optional selector that further filters which elements of the wrapped set are to be removed.

Returns
The wrapped set.

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

Note, however, that any jQuery data or events that were bound to the elements are removed when the elements are removed from the DOM using `remove()`. A similar method, `detach()`, also removes the elements from the DOM, but retains any bound events and data.

Method syntax: detach

`detach(selector)`
Removes all elements in the wrapped set from the page DOM, retaining any bound events and jQuery data.

Parameters

`selector`	(Selector) An optional selector string that further filters which elements of the wrapped set are to be detached.

Returns
The wrapped set.

The `detach()` method is the preferred means of removing an element that we'll want to put back into the DOM at a later time with its events and data intact.

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

Method syntax: empty

`empty()`

Removes the content of all DOM elements in the matched set.

Parameters

`none`

Returns

The wrapped set.

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

3.3.5 *Cloning elements*

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

Method syntax: clone

`clone(copyHandlers)`

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

Parameters

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

Returns

The newly created wrapped set.

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

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

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

A slightly more complex example is as follows:

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

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

One last example:

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

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

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

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

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

3.3.6 *Replacing elements*

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

Method syntax: replaceWith

`replaceWith(content)`

Replaces each matched element with the specific content.

Parameters

`content` (String|Element|Function) A string containing an HTML fragment to become the replaced content, or an element reference to be moved to replace the existing elements. If a function, the function is invoked for each wrapped element, setting that element as the function context and passing no parameters. The function's returned value is used as the new content.

Returns

A jQuery wrapped set containing the replaced elements.

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

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

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

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

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

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

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

Method syntax: replaceAll

`replaceAll(selector)`

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

Parameters

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

Returns

A jQuery wrapped set containing the inserted elements.

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

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

3.4 Dealing with form element values

Because form elements have special properties, jQuery's core contains a number of convenience functions for activities such as

- Getting and setting their values
- Serializing them
- Selecting elements based on form-specific properties

These functions will serve us well in most cases, but the Form Plugin—an officially sanctioned plugin developed by members of the jQuery Core Team—provides even more form-related functionality. Learn more about this plugin at http://jquery.malsup.com/form/.

So what's a form element?

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

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

Method syntax: val

`val()`

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

Parameters

`none`

Returns

The fetched value or values.

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

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

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

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

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

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

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

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

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

Method syntax: val

`val(value)`

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

Parameters

`value`	(String|Function) Specifies the value that is to be set as the `value` property of each form element in the wrapped set. If a function, the function is invoked for each element in the wrapped set, with that element passed as the function context, and two parameters: the element index and the current value of the element. The value returned from the function is taken as the value to be set.

Returns

The wrapped set.

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

Method syntax: val

`val(values)`

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

Parameters

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

Returns

The wrapped set.

Consider the following statement:

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

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

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

3.5 Summary

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

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

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