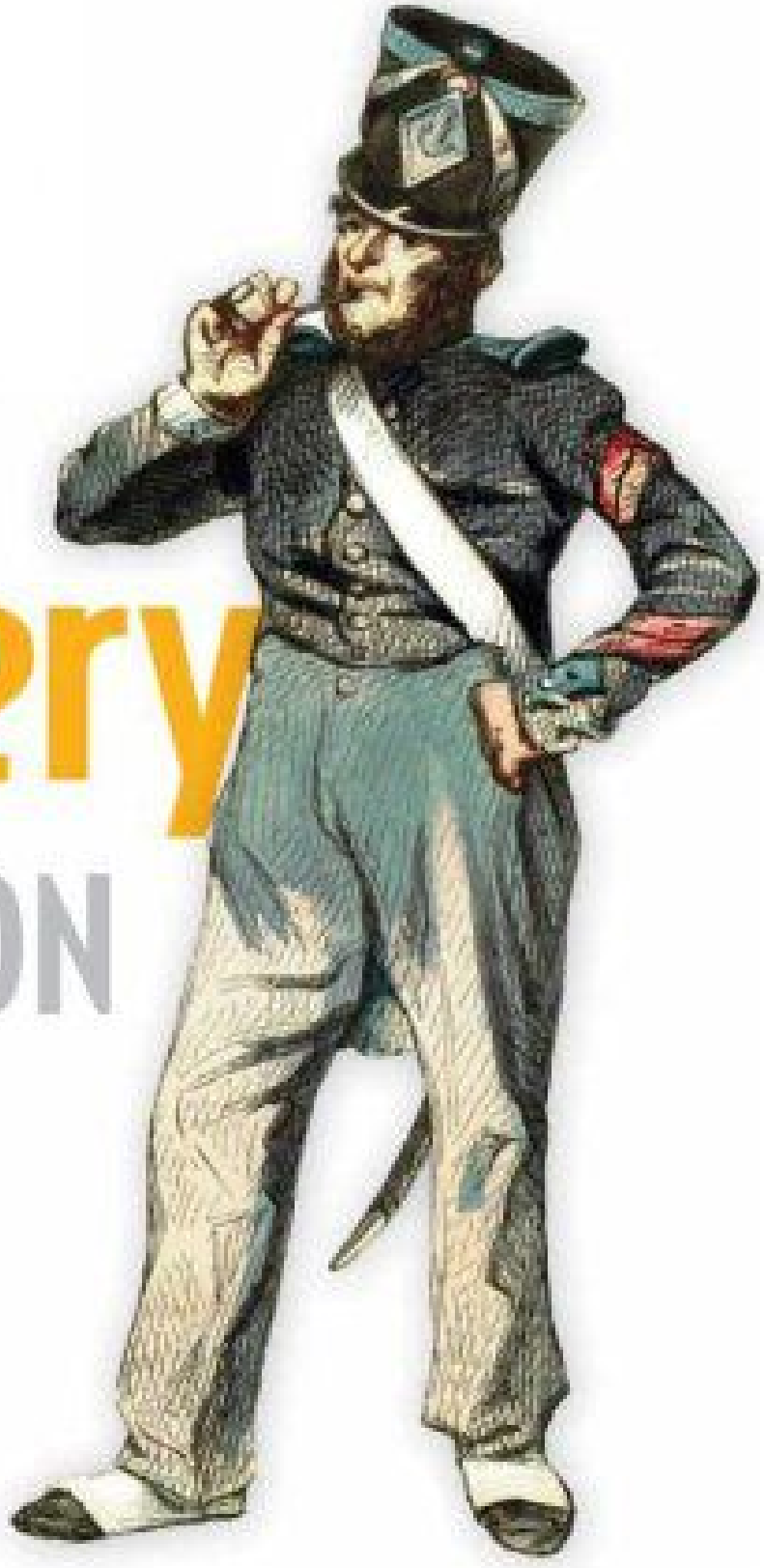
Bear Bibeault
Yehuda Katz

Covers jQuery 1.4 and jQuery UI 1.8



jQuery IN ACTION

SECOND EDITION

MANNING

Table of Contents

2 Selecting the elements upon which to act

This chapter covers

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

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

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

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

2.1 Selecting elements for manipulation

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

Fortunately, jQuery provides a robust *selector* syntax we can use to easily specify sets of elements elegantly and concisely. You probably already know a big chunk of the syntax: jQuery uses the CSS syntax you already know and love, and extends it with some custom means to perform both common and complex selections.

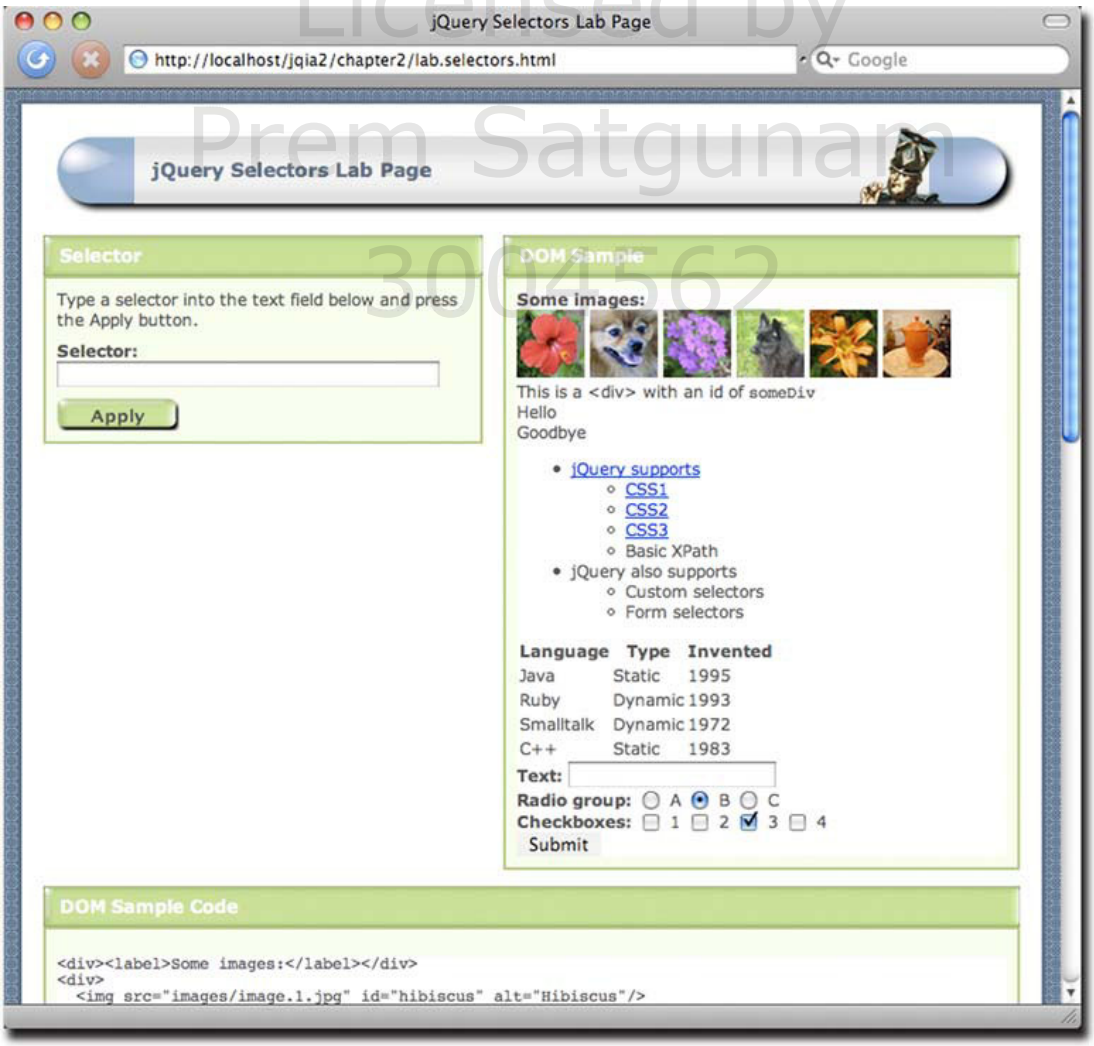


Figure 2.1 The jQuery Selectors Lab Page allows you to observe the behavior of any selector you choose in real time.

To help you learn about element selection, we've put together a jQuery Selectors Lab Page that's available within the downloadable code examples for this book (in file chapter2/lab.selectors.html). The Selectors Lab allows you to enter a jQuery selector string and see (in real time!) which DOM elements get selected. When displayed, the Lab should look as shown in figure 2.1 (if the panes don't appear correctly lined up, you may need to widen your browser window).

> **TIP** If you haven't yet downloaded the example code, you really ought to do so now—the information in this chapter will be much easier to absorb if you follow along with the Lab exercises. Visit this book's web page at http://www.manning.com/bibeault2 to find the download link.

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

The selector that you type (in this case `li`) is applied to the HTML fragment loaded into the DOM Sample pane at upper right. The Lab code that executes when Apply is clicked adds a class named `wrappedElement` to be applied to all matching elements. A CSS rule defined for the page causes all elements with that class to be highlighted with a red border and pink background. After clicking Apply, you should see the display shown in figure 2.2, in which all `<li>` elements in the DOM sample are highlighted.

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

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

We'll talk more about using this Lab as we progress through the chapter. But first, your authors must admit that they've been blatantly over-simplifying an important concept, and that's going to be rectified now.

2.1.1 Controlling the context

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

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

That's quite often exactly what we want, so it's a nice default. But there may be times when we want to limit our search to a subset of the entire DOM. In such cases, we

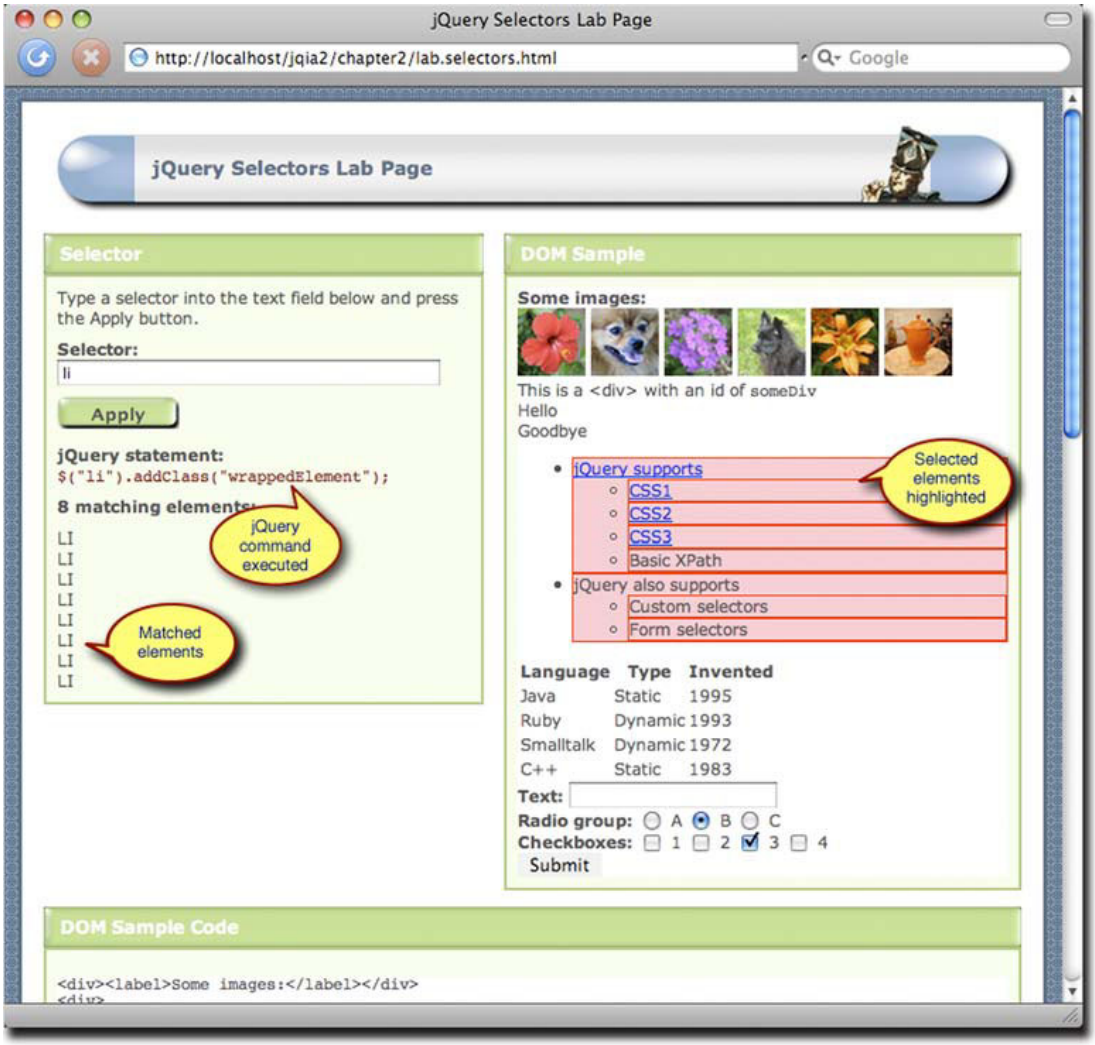


Figure 2.2 A selector value of `li` matches all `<li>` elements when applied, as shown by the displayed results.

can identify a subset of the DOM that serves as the root of the sub-tree to which the selector is applied.

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

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

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

Let's take the Lab Page as an example. We'll assume that the selector string is stored in a variable conveniently named `selector`. When we apply this submitted selector, we only want to apply it to the sample DOM, which is contained within a `<div>` element with an `id` value of `sampleDOM`.

Were we to code the call to the jQuery function like this,

```
$(selector)
```

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

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

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

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

2.1.2 Using basic CSS selectors

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

Table 2.1 provides some examples to give you a quick refresher. We can mix and match these basic selector types to identify fairly fine-grained sets of elements.

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

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

With a few exceptions, jQuery is fully CSS3 compliant, so selecting elements this way will present no surprises; the same elements that would be selected in a style sheet by a

Table 2.1 Some simple CSS selector examples

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

standards-compliant browser will be selected by jQuery's selector engine. Note that jQuery does *not* depend upon the CSS implementation of the browser it's running within. Even if the browser doesn't implement a standard CSS selector correctly, jQuery will correctly select elements according to the rules of the World Wide Web Consortium (W3C) standard.

jQuery also lets us combine multiple selectors into a single expression using the comma operator. For example, to select all `<div>` and all `<span>` elements, you could do this:

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

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

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

2.1.3 Using child, container, and attribute selectors

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

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

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

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

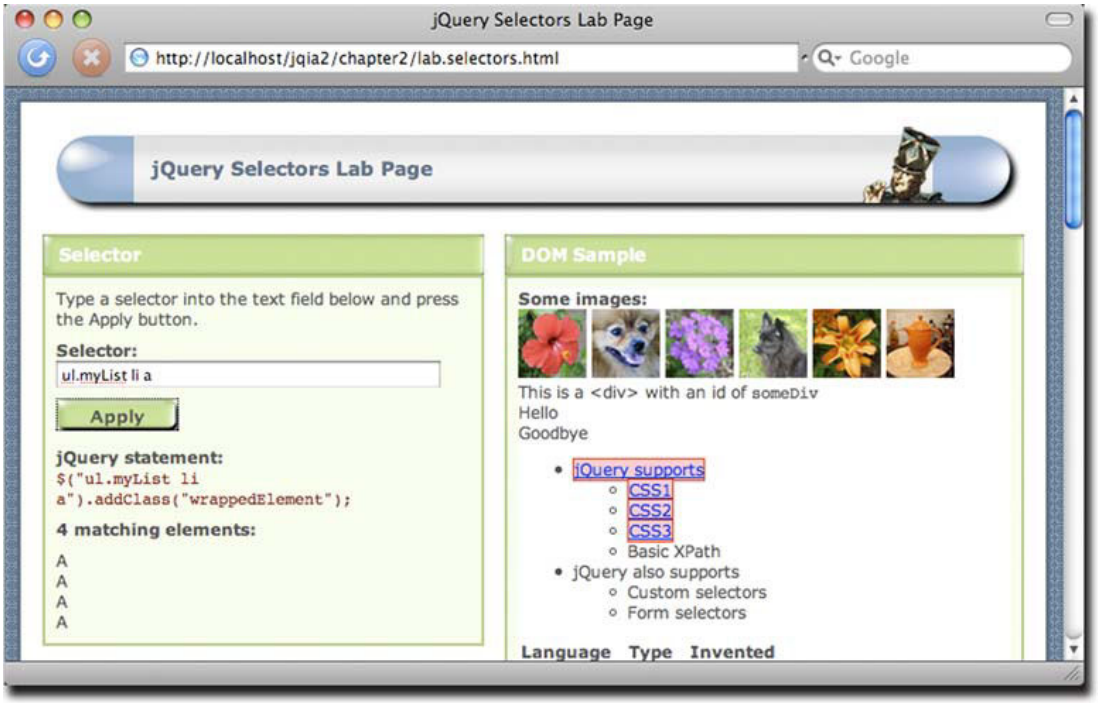


Figure 2.3 All anchor tags that are descendants, at any depth, of an `<li>` element are selected by `ul.myList li a`.

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

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

```
p > a
```

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

Going back to our example, consider a selector such as

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

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

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

```
<li><a href="http://jquery.com">jQuery supports</a>
    <ul>
      <li><a href="css1">CSS1</a></li>
```

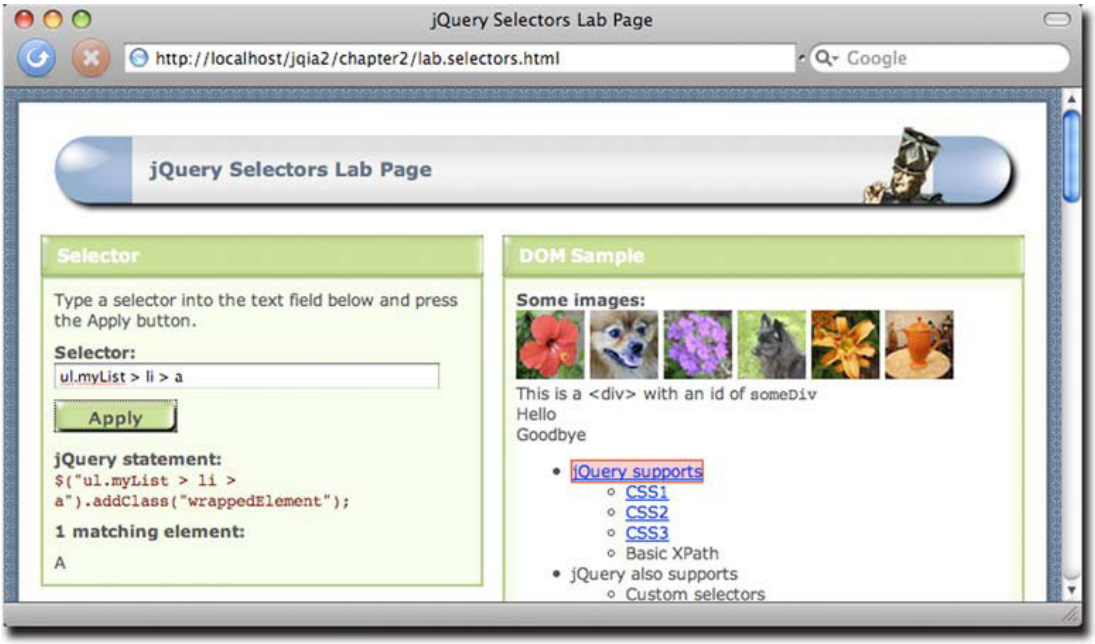


Figure 2.4 **With the selector `ul.myList > li > a`, only the direct children of parent nodes are matched.**

```
      <li><a href="css2">CSS2</a></li>
      <li><a href="css3">CSS3</a></li>
      <li>Basic XPath</li>
    </ul>
</li>
```

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

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

This matches all links with an `href` value beginning with the exact string `http://`. The caret character (`^`) is used to specify that the match is to occur at the beginning of a value. As this is the same character used by most regular expression processors to signify matching at the beginning of a candidate string, it should be easy to remember.

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

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

```
form[method]
```

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

To match a specific attribute value, we use something like

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 2.2 The basic CSS selectors supported by jQuery

Selector	Description
`*`	Matches any element.
`E`	Matches all elements with tag name `E`.
`E F`	Matches all elements with tag name `F` that are descendants of `E`.
`E>F`	Matches all elements with tag name `F` that are direct children of `E`.
`E+F`	Matches all elements with tag name `F` that are immediately preceded by sibling `E`.
`E~F`	Matches all elements with tag name `F` preceded by any sibling `E`.
`E.C`	Matches all elements with tag name `E` with class name `C`. Omitting `E` is the same as `*.C`.
`E#I`	Matches all elements with tag name `E` with the id of `I`. Omitting `E` is the same as `*#I`.
`E[A]`	Matches all elements with tag name `E` that have attribute `A` of any value.
`E[A=V]`	Matches all elements with tag name `E` that have attribute `A` whose value is exactly `V`.
`E[A^=V]`	Matches all elements with tag name `E` that have attribute `A` whose value starts with `V`.
`E[A$=V]`	Matches all elements with tag name `E` that have attribute `A` whose value ends with `V`.
`E[A!=V]`	Matches all elements with tag name `E` that have attribute `A` whose value doesn't match the value `V`, or that lack attribute `A` completely.
`E[A*=V]`	Matches all elements with tag name `E` that have attribute `A` whose value contains `V`.

2.1.4 Selecting by position

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

For example, consider

```
a:first
```

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

What about picking every other element?

```
p:odd
```

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

```
p:even
```

Another form,

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

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

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

See table 2.3 for a list of these positional filters (which the jQuery documentation terms the *basic* and *child* filters).

Table 2.3 The positional filter selectors supported by jQuery

Selector	Description
`:first`	Matches the first match within the context. `li a:first` returns the first link that's a descendant of a list item.
`:last`	Matches the last match within the context. `li a:last` returns the last link that's a descendant of a list item.
`:first-child`	Matches the first child element within the context. `li:first-child` returns the first list item of each list.
`:last-child`	Matches the last child element within the context. `li:last-child` returns the last list item of each list.
`:only-child`	Returns all elements that have no siblings.

Table 2.3 The positional filter selectors supported by jQuery *(continued)*

Selector	Description
`:nth-child(n)`	Matches the *n*th child element within the context. `li:nth-child(2)` returns the second list item of each list.
`:nth-child(even|odd)`	Matches even or odd children within the context. `li:nth-child(even)` returns the even list items of each list.
`:nth-child(Xn+Y)`	Matches the *n*th child element computed by the supplied formula. If `Y` is 0, it may be omitted. `li:nth-child(3n)` returns every third list item, whereas `li:nth-child(5n+1)` returns the item after every fifth element.
`:even`	Matches even elements within the context. `li:even` returns every even list item.
`:odd`	Matches odd elements within the context. `li:odd` returns every odd list item.
`:eq(n)`	Matches the *n*th matching element.
`:gt(n)`	Matches matching elements after and excluding the *n*th matching element.
`:lt(n)`	Matches matching elements before and excluding the *n*th matching element.

There is one quick gotcha (isn't there always?). The `:nth-child` filter starts counting from 1 (for CSS compatibility), whereas the other selectors start counting from 0 (following the more common programming convention). This becomes second nature with practice, but it may be a bit confusing at first.

Let's dig in some more.

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

```
<table id="languages">
  <thead>
    <tr>
      <th>Language</th>
      <th>Type</th>
      <th>Invented</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Java</td>
      <td>Static</td>
      <td>1995</td>
    </tr>
    <tr>
      <td>Ruby</td>
      <td>Dynamic</td>
      <td>1993</td>
```

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

Let's say that we wanted to get all of the table cells that contain the names of programming languages. Because they're all the first cells in their rows, we could use

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

We could also easily use

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

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

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

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

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

2.1.5 Using CSS and custom jQuery filter selectors

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

As an example, we might want to match all checkboxes that are in checked state. You might be tempted to try something along these lines:

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

But trying to match by attribute will only check the *initial* state of the control as specified in the HTML markup. What we really want to check is the real-time state of the controls. CSS offers a pseudo-class, `:checked`, that matches elements that are in a checked state. For example, whereas the `input` selector selects all `<input>` elements,

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

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

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

Table 2.4 The CSS and custom jQuery filter selectors

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

Table 2.4 The CSS and custom jQuery filter selectors *(continued)*

Selector	Description	In CSS?
`:selected`	Selects only `<option>` elements that are in selected state.	
`:submit`	Selects only submit buttons (`button[type=submit]` or `input[type=submit]`).	
`:text`	Selects only text elements (`input[type=text]`).	
`:visible`	Selects only elements that are visible.	

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

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

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

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

USING THE :NOT FILTER

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

For example, to select non-checkbox `<input>` elements, you could use

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

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

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

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

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

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

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

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

jQuery also adds a custom filter that helps when making selections using parent-child relationships.

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

USING THE :HAS FILTER

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

```
div span
```

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

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

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

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

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

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

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

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

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

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

2.2 Generating new HTML

Sometimes, we'll want to generate new fragments of HTML to insert into the page. Such dynamic elements could be as simple as extra text we want to display under certain conditions, or something as complicated as creating a table of database results we've obtained from a server.

With jQuery, creating dynamic elements is a simple matter, because, as we saw in chapter 1, the `$()` function can create elements from HTML strings in addition to selecting existing page elements. Consider this line:

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

This expression creates a new `<div>` element ready to be added to the page. Any jQuery methods that we could run on wrapped element sets of existing elements can

be run on the newly created fragment. This may not seem impressive on first glance, but when we throw event handlers, Ajax, and effects into the mix (as we will in the upcoming chapters), you'll see that it could come in mighty handy.

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

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

This is identical to `$("<div></div>")` and `$("<div/>")`, though it is recommended that you use well-formed markup and include the opening and closing tags for any element types that can contain other elements.

It's almost embarrassingly easy to create such simple HTML elements, and thanks to the chainability of jQuery methods, creating more complex elements isn't much harder. We can apply any jQuery method to the wrapped set containing the newly created element. For example, we can apply styles to the element with the `css()` method. We could also create attributes on the element with the `attr()` method, but jQuery provides an even better means to do so.

We can pass a second parameter to the element-creating `$()` method that specifies the attributes and their values. This parameter takes the form of a JavaScript object whose properties serve as the name and value of the attributes to be applied to the element.

Let's say that we want to create an image element complete with multiple attributes, some styling, and let's make it clickable to boot! Take a look at the code in listing 2.1.

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

```
$('<img>',                          <-- 1 Creates the basic <img> element
  {
    src: 'images/little.bear.png',
    alt: 'Little Bear',                          <-- 2 Assigns various attributes
    title:'I woof in your general direction',
    click: function(){                           <-- 3 Establishes click handler
      alert($(this).attr('title'));
    }
  })
  .css({                            <-- 4 Styles the image
    cursor: 'pointer',
    border: '1px solid black',
    padding: '12px 12px 20px 12px',
    backgroundColor: 'white'
  })
  .appendTo('body');                <-- 5 Attaches the element to the document
```

The single jQuery statement in listing 2.1 creates the basic `<img>` element ❶, gives it important attributes, such as its source, alternate text, and flyout title ❷, styles it to look like a printed photograph ❹, and attaches it to the DOM tree ❺.

We also threw a bit of a curve ball at you here. We used the `attribute` object to establish an event handler that issues an alert (garnered from the image's `title`) when the image is clicked (3).

jQuery not only lets us specify attributes in the `attribute` parameter; we can also establish handlers for all the event types (which we'll be exploring in depth in chapter 4), as well as supply values for handful of jQuery methods whose purpose is to set various facets of the element. We haven't examined these methods yet, but we can set values for the following methods (which we'll mostly discuss in the next chapter): `val`, `css`, `html`, `text`, `data`, `width`, `height`, and `offset`.

So, in listing 2.1 we could omit the call to the chained `css()` method, replacing it with the following property in the `attribute` parameter:

```
css: {
  cursor: 'pointer',
  border: '1px solid black',
  padding: '12px 12px 20px 12px',
  backgroundColor: 'white'
}
```

Regardless of how we arrange the code, that's a pretty hefty statement—which we spread across multiple lines, and with logical indentation, for readability—but it also does a heck of a lot. Such statements aren't uncommon in jQuery-enabled pages, and if you find it a bit overwhelming, don't worry, we'll be covering every method used in this statement over the next few chapters. Writing such compound statements will be second nature before much longer.

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

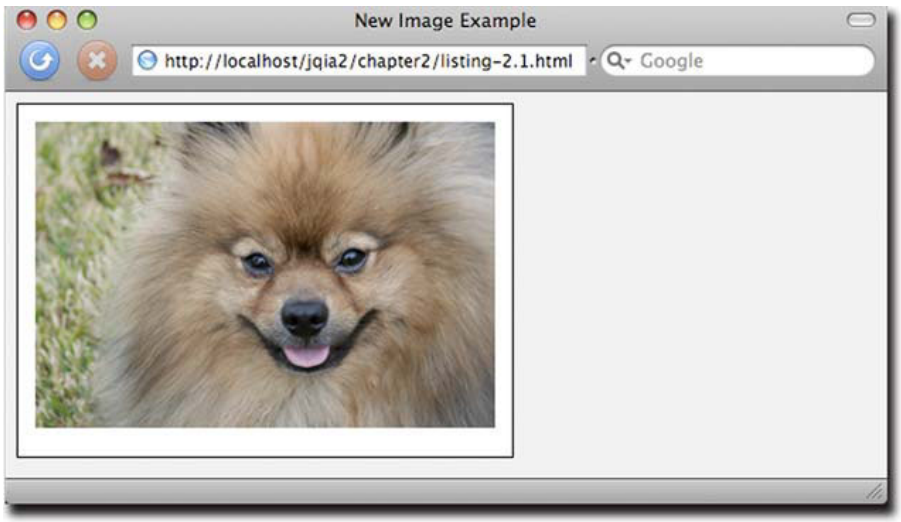


Figure 2.5a Creating complex elements on the fly, including this image, which generates an alert when it's clicked upon, is easy as pie.

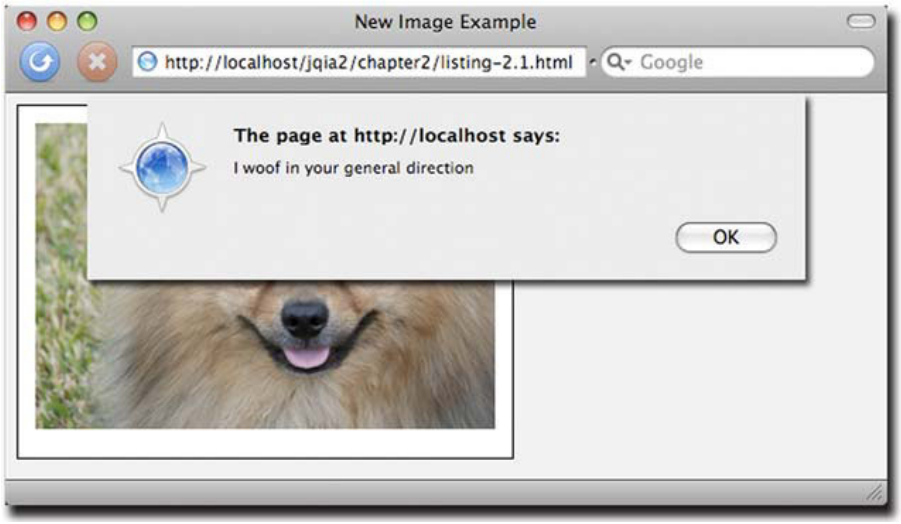


Figure 2.5b The dynaimcally-generated image possesses all expected styles and attributes, including the mouse click behavior of issuing an alert

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

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

2.3 Managing the wrapped element set

Once we've got a set of wrapped elements, whether identified from existing DOM elements with selectors, or created as new elements using HTML snippets (or a combination of both), we're ready to manipulate those elements using the powerful set of jQuery methods. We'll start looking at those methods in the next chapter, but what if we want to further refine the set of elements wrapped by the jQuery function? In this section, we'll explore the many ways that we can refine, extend, or subset the set of wrapped elements that we wish to operate upon.

In order to help you in this endeavor, we've included another Lab in the downloadable project code for this chapter: the jQuery Operations Lab Page (chapter2/lab.operations.html). This page, which looks a lot like the Selectors Lab we employed earlier in this chapter, is shown in figure 2.6.

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

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

Figure 2.6 The jQuery Operations Lab Page lets us compose wrapped sets in real time to help us see how wrapped sets can be created and managed.

way jQuery chaining works, this expression can also include wrapper methods, making this a powerful Lab for examining the operations of jQuery.

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

To get a feel for the Lab, display it in your browser and enter this text into the Operation field:

```
$('img').hide()
```

Then click the Execute button.

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

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

2.3.1 Determining the size of a wrapped set

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

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

Consider the following statement:

```
$('#someDiv')
  .html('There are '+$('a').size()+' link(s) on this page.');
```

The jQuery expression embedded in the statement matches all elements of type `<a>` and returns the number of matched elements using the `size()` method. This is used to construct a text string, which is set as the content of an element with `id` of `someDiv` using the `html()` method (which we'll see in the next chapter).

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

Method syntax: size

`size()`

Returns the count of elements in the wrapped set.

Parameters

`none`

Returns

The element count.

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

2.3.2 Obtaining elements from a wrapped set

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

Let's look at some of the ways that jQuery allows us to do just that.

FETCHING ELEMENTS BY INDEX

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

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

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

Method syntax: get

`get(index)`

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

Parameters

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

Returns

A DOM element or an array of DOM elements.

The fragment

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

is equivalent to the previous example that used array indexing.

The `get()` method will also accept a negative index value as a parameter. In this case, it fetches the element relative to the end of the wrapped set. For example `.get(-1)` will retrieve the last element in the wrapped set, `.get(-2)` the second to last, and so on.

In addition to obtaining a single element, `get()` can also return an array.

Although the `toArray()` method (discussed in the next section) is the preferred way to obtain a JavaScript array of the elements within a wrapped set, the `get()` method can also be used to obtain a plain JavaScript array of all the wrapped elements.

This method of obtaining an array is provided for backward compatibility with previous versions of jQuery.

The `get()` method returns a DOM element, but sometimes we'll want a wrapped set containing a specific element rather than the element itself. It would look really weird to write something like this:

```
$($('p').get(23))
```

So jQuery provides the `eq()` method, that mimics the action of the `:eq` selector filter:

Method syntax: eq

`eq(index)`
Obtains the indexed element in the wrapped set and returns a new wrapped set containing just that element.

Parameters

`index` (Number) The index of the single element to return. As with `get()`, a negative index can be specified to index from the end of the set.

Returns
A wrapped set containing one or zero elements.

Obtaining the first element of a wrapped set is such a common operation that there's a convenience method that makes it even easier: the `first()` method.

Method syntax: first

`first()`
Obtains the first element in the wrapped set and returns a new wrapped set containing just that element. If the original set is empty, so is the returned set.

Parameters

`none`

Returns
A wrapped set containing one or zero elements.

As you might expect, there's a corresponding method to obtain the last element in a wrapped set as well.

Method syntax: last

`last()`
Obtains the last element in the wrapped set and returns a new wrapped set containing just that element. If the original set is empty, so is the returned set.

Parameters

`none`

Returns
A wrapped set containing one or zero elements.

Now let's examine the preferred method of obtaining an array of wrapped elements.

FETCHING ALL THE ELEMENTS AS AN ARRAY

If we wish to obtain all of the elements in a wrapped set as a JavaScript array of DOM elements, jQuery provides the `toArray()` method:

Method syntax: toArray

`toArray()`

Returns the elements in the wrapped set as an array of DOM elements.

Parameters

`none`

Returns

A JavaScript array of the DOM elements within the wrapped set.

Consider this example:

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

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

FINDING THE INDEX OF AN ELEMENT

While `get()` finds an element given an index, we can use an inverse operation, `index()`, to find the index of a particular element in the wrapped set. The syntax of the `index()` method is as follows:

Method syntax: index

`index(element)`

Finds the passed element in the wrapped set and returns its ordinal index within the set, or finds the ordinal index of the first element of the wrapped set within its siblings. If the element isn't found, the value -1 is returned.

Parameters

`element` (Element|Selector) A reference to the element whose ordinal value is to be determined, or a selector that identifies the element. If omitted, the first element of the wrapped set is located within its list of siblings.

Returns

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

Let's say that for some reason we want to know the ordinal index of an image with the `id` of `findMe` within the entire set of images in a page. We can obtain this value with this statement:

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

We can also shorten this:

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

The `index()` method can also be used to find the index of an element within its parent (that is, among its siblings). For example,

```
var n = $('img').index();
```

This will set `n` to the ordinal index of the first `<img>` element within its parent.

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

2.3.3 Slicing and dicing a wrapped element set

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

ADDING MORE ELEMENTS TO A WRAPPED SET

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

We'll look at some concrete examples of such situations in a moment, but first, let's start with a simpler scenario. Let's say that we want to match all `<img>` elements that have either an `alt` or a `title` attribute. The powerful jQuery selectors allow us to express this as a single selector, such as

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

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

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

Using the `add()` method in this fashion allows us to chain a bunch of selectors together, creating a union of the elements that satisfy either of the selectors.

Methods such as `add()` are also significant (and more flexible than aggregate selectors) within jQuery method chains because they don't augment the original wrapped set, but create a *new* wrapped set with the result. We'll see in just a bit how this can be extremely useful in conjunction with methods such as `end()` (which we'll examine in section 2.3.6) that can be used to "back out" operations that augment original wrapped sets.

This is the syntax of the `add()` method:

Method syntax: add

`add(expression,context)`

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

Parameters

`expression`	(Selector|Element|Array) Specifies what is to be added to the matched set. This parameter can be a jQuery selector, in which case any matched elements are added to the set. If the parameter is an HTML fragment, the appropriate elements are created and added to the set. If it is a DOM element or an array of DOM elements, they're added to the set.
`context`	(Selector|Element|Array|jQuery) Specifies a context to limit the search for elements that match the first parameter. This is the same context that can be passed to the `jQuery()` function. See section 2.1.1 for a description of this parameter.

Returns

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

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

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

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

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

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

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

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

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

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

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

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

The `add()` method can also be used to add elements to an existing wrapped set, given direct references to those elements. Passing an element reference, or an array of

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

element references, to the `add()` method adds the elements to the wrapped set. If we had an element reference in a variable named `someElement`, it could be added to the set of all images containing an `alt` property with this statement:

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

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

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

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

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

HONING THE CONTENTS OF A WRAPPED SET

We saw that it's a simple matter to augment wrapped sets from multiple selectors chained together with the `add()` method. It's also possible to chain selectors together to form an *except* relationship by employing the `not()` method. This is similar to the `:not` filter selector we discussed earlier, but it can be employed in a similar fashion to the `add()` method to remove elements from the wrapped set anywhere within a jQuery chain of methods.

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

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

Type this expression into the jQuery Operations Lab Page, and execute it. You'll see that only the tan puppy image has the highlight applied. The black puppy, which is

included in the original wrapped set because it possesses a `title` attribute, is removed by the `not()` invocation because its `title` contains the text "puppy".

Method syntax: not

`not(expression)`

Creates a copy of the wrapped set and removes elements from the new set that match criteria specified by the value of the `expression` parameter.

Parameters

`expression`	(Selector|Element|Array|Function) Specifies which items are to be removed. If the parameter is a jQuery selector, any matching elements are removed. If an element reference or array of elements is passed, those elements are removed from the set. If a function is passed, the function is invoked for each item in the wrapped set (with `this` set to the item), and returning `true` from the invocation causes the item to be removed from the wrapped set.

Returns

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

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

When maximum flexibility is needed, a function can be passed to `not()`, and a determination of whether to keep or remove the element can be made on an element-by-element basis. Consider this example:

```
$('img').not(function(){ return !$(this).hasClass('keepMe'); })
```

This expression will select all `<img>` elements and then remove any that don't have the class "keepMe".

This method allows us to filter the wrapped set in ways that are difficult or impossible to express with a selector expression by resorting to programmatic filtering of the wrapped set items.

For those times when the test applied within the function passed to `not()` seems to be the opposite of what we want to express, `not()` has an inverse method, `filter()`, that works in a similar fashion, except that it removes elements when the function returns `false`.

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

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

This jQuery expression creates a wrapped set of all `<td>` elements and then invokes the function passed to the `filter()` method for each, with the current matched

elements as the `this` value for the invocation. The function uses a regular expression to determine whether the element content matches the described pattern (a sequence of one or more digits), returning `false` if not. Elements whose filter function invocation returns `false` aren't included in the returned wrapped set.

Method syntax: filter

`filter(expression)`

Creates a copy of the wrapped set and removes elements from the new set that don't match criteria specified by the value of the `expression` parameter.

Parameters

`expression`	(Selector|Element|Array|Function) Specifies which items are to be removed. If the parameter is a jQuery selector, any elements that don't match are removed. If an element reference or array of elements is passed, all but those elements are removed from the set. If a function is passed, the function is invoked for each element in the wrapped set (with `this` referencing the element), and returning `false` from the invocation causes the element to be removed from the wrapped set.

Returns

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

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

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

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

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

The `not()` and `filter()` methods give us powerful means to adjust a set of wrapped elements on the fly, based on just about any criteria concerning the wrapped elements. But we can also subset the wrapped set, based on the *position* of the elements within the set. Let's look at those methods next.

OBTAINING SUBSETS OF A WRAPPED SET

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

Method syntax: slice

`slice(begin,end)`

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

Parameters

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

Returns

The newly created wrapped set.

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

For example, to obtain the third element, we could write

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

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

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

Therefore, a statement such as

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

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

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

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

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

Another method we can use to obtain a subset of a wrapped set is the `has()` method. Like the `:has` filter, this method tests the children of the wrapped elements, using this check to choose the elements to become part of the subset.

Method syntax: has

`has(test)`
Creates and returns a new wrapped set containing only elements from the original wrapped set that contain descendents that match the passed `test` expression.

Parameters

`test`	(Selector|Element) A selector to be applied to all descendents of the wrapped elements, or an element to be tested. Only elements possessing an element that matches the selector, or the passed element, are included in the returned wrapped set.

Returns
The resulting wrapped set.

For example, consider this line:

```
$('div').has('img[alt]')
```

This expression will create a wrapped set of all `<div>` elements, and then create and return a second set that contains only those `<div>` elements that contain at least one descendent `<img>` element that possess an `alt` attribute.

TRANSLATING ELEMENTS OF A WRAPPED SET

We'll often want to perform transformations on the elements of a wrapped set. For example, what if we wanted to collect all the `id` values of the wrapped elements, or perhaps collect the values of a wrapped set of form elements in order to create a query string from them? The `map()` method comes in mighty handy for such occasions.

Method syntax: map

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

Parameters

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

Returns
The wrapped set of translated values.

For example, the following code will collect all the `id` values of all images on the page into a JavaScript array:

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

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

TRAVERSING A WRAPPED SET'S ELEMENTS

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

Method syntax: each

`each(iterator)`
Traverses all elements in the matched set, invoking the passed iterator function for each.

Parameters

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

Returns
The wrapped set.

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

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

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

As a convenience, the `each()` method will also iterate over arrays of JavaScript objects and even individual objects (not that the latter has a lot of utility). Consider this example:

```
$([1,2,3]).each(function(){ alert(this); });
```

This statement will invoke the iterator function for each element of the array that's passed to `$()`, with the individual array items passed to the function as `this`.

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

2.3.4 Getting wrapped sets using relationships

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

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

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

Table 2.5 Methods for obtaining a new wrapped set based on relationships to other HTML DOM elements

Method	Description
`children([selector])`	Returns a wrapped set consisting of all unique children of the wrapped elements.
`closest([selector])`	Returns a wrapped set containing the single nearest ancestor that matches the passed expression, if any.
`contents()`	Returns a wrapped set of the contents of the elements, which may include text nodes, in the wrapped set. (This is frequently used to obtain the contents of `<iframe>` elements.)
`next([selector])`	Returns a wrapped set consisting of all unique next siblings of the wrapped elements.
`nextAll([selector])`	Returns a wrapped set containing all the following siblings of the wrapped elements.
`nextUntil([selector])`	Returns a wrapped set of all the following siblings of the elements of the wrapped elements until, but not including, the element matched by the selector. If no matches are made to the selector, or if the selector is omitted, all following siblings are selected.
`offsetParent()`	Returns a wrapped set containing the closest relatively or absolutely positioned parent of the first element in the wrapped set.
`parent([selector])`	Returns a wrapped set consisting of the unique direct parents of all wrapped elements.
`parents([selector])`	Returns a wrapped set consisting of the unique ancestors of all wrapped elements. This includes the direct parents as well as the remaining ancestors all the way up to, but not including, the document root.
`parentsUntil([selector])`	Returns a wrapped set of all ancestors of the elements of the wrapped elements up until, but not including, the element matched by the selector. If no matches are made to the selector, or if the selector is omitted, all ancestors are selected.
`prev([selector])`	Returns a wrapped set consisting of all unique previous siblings of the wrapped elements.
`prevAll([selector])`	Returns a wrapped set containing all the previous siblings of the wrapped elements.
`prevUntil([selector])`	Returns a wrapped set of all preceding siblings of the elements of the wrapped elements until, but not including, the element matched by the selector. If no matches are made to the selector, or if the selector is omitted, all previous siblings are selected.
`siblings([selector])`	Returns a wrapped set consisting of all unique siblings of the wrapped elements.

Consider a situation where a button's event handler (which we'll be exploring in great detail in chapter 4) is triggered with the button element referenced by the `this` keyword within the handler. Further, let's say that we want to find the `<div>` block within which the button is defined. The `closest()` method makes it a breeze:

```
$(this).closest('div')
```

But this would only find the most immediate ancestor `<div>`; what if the `<div>` we seek is higher in the ancestor tree? No problem. We can refine the selector we pass to `closest()` to discriminate which elements are selected:

```
$(this).closest('div.myButtonContainer')
```

Now the ancestor `<div>` with the class `myButtonContainer` will be selected.

The remainder of these methods work in a similar fashion. Take, for example, a situation in which we want to find a sibling button with a particular `title` attribute:

```
$(this).siblings('button[title="Close"]')
```

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

2.3.5 Even more ways to use a wrapped set

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

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

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

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

Method syntax: find

`find(selector)`

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

Parameters

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

Returns

The newly created wrapped set.

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

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

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

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

Method syntax: is

`is(selector)`

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

Parameters

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

Returns

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

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

2.3.6 *Managing jQuery chains*

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

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

Consider the following statement:

```
$('img').filter('[title]').hide();
```

Two wrapped sets are generated within this statement: the original wrapped set of all the `<img>` elements in the DOM, and a second wrapped set consisting of only those wrapped elements which also possess `title` attributes. (Yes, we could have done this

with a single selector, but bear with us for illustration of the concept. Imagine that we do something important in the chain before the call to `filter()`.)

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

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

Consider

```
$('img').filter('[title]').hide().end().addClass('anImage');
```

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

Method syntax: end

`end()`

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

Parameters

`none`

Returns

The previous wrapped set.

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

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

Method syntax: andSelf

`andSelf()`

Merges the two previous wrapped sets in a method chain.

Parameters

`none`

Returns

The merged wrapped set.

Consider

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

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

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

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

2.4 Summary

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

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

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

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

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

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