CHAPTER 25
Introduction to jQuery

Powerful and flexible as JavaScript is, with its plethora of built-in functions and the continuing improvements being made to it, you often still need additional layers of code for simple things that cannot be achieved natively or with CSS, such as animations, event handling, and asynchronous communication.

What's more, as a consequence of the various browser wars over the years, frustrating and annoying browser incompatibilities have come and gone, rearing their heads at different times on different platforms and programs.

As a result, ensuring your web pages look the same on all devices can sometimes be achieved only through tedious JavaScript code that accounts for all the discrepancies across the range of browsers and versions released over recent years. Yes, much has been done to resolve these differences, but even today every responsive site I create still needs exceptions to handle the different browsers, unless I use a framework, as I'm sure many would agree.

To fill these gaps, a number of libraries of functions (many of which also provide easy hooks into the DOM) have been developed to minimize the differences between browsers and to facilitate asynchronous communication and event and animation handling, such as the subject of this chapter, jQuery.

Sometimes there is a simple JavaScript-based solution to the problem you are trying to solve. One way to check whether this is the case is to make a search at youmightnotneedjquery.com, which will let you know about any alternatives to jQuery that may actually be simpler in a particular situation.

Why jQuery?

With jQuery, not only do you get a very high level of cross-browser compatibility, but you also have quick and easy access to HTML and DOM manipulation, special functions to interact directly with CSS, the ability to control events, powerful tools to create professional effects and animations, and functions for conducting asynchronous communications with the web server. jQuery is also the base for a wide range of plug-ins and other utilities.

Of course, you don't *need* to use jQuery, and some programming purists would never touch a library, preferring to create their own bespoke collections of functions (and there can be good reasons for this, such as not having to wait on others to correct bugs you find, implementing your own security features, and so on). But jQuery has definitely stood the test of time, and if you would like to take advantage of its gentle learning curve and be developing quality web pages as quickly as possible, this chapter will show how you can get started using it.

Including jQuery

There are two ways to include jQuery in your web pages. You can go to the download page, download the version you need, upload it to your web server, and reference it from a `<script>` tag in your HTML files. Or you can take advantage of a free content delivery network (CDN) and simply link to the version you require.

jQuery is released under the terms of the MIT license, which places almost no restrictions on what you can do with it. You are free to use any jQuery project in any other project (even commercial projects) as long as the copyright header is left intact.

Choosing the Right Version

Before deciding whether to download and host jQuery directly or to use a CDN, you must pick a version of jQuery. In most cases this is straightforward, because you'll simply opt for the latest release. However, if you are targeting particular browsers, or if you are maintaining a legacy website that relies on a particular version of jQuery, the latest one may not be right for you.

Unlike most software, where you simply download and install the newest available version, jQuery has evolved over time to take account of the changing dynamics in the market of differing browser versions, with different features and bugs.

At the same time, improvements to jQuery have been made that might make newer versions operate differently on sites that have been specially tailored to a particular version (and the quirks surrounding it).

Of course, each newer version is an improvement over the previous one and is more and more likely to cover all the bases. But where identical operation is critical for a website, until you have fully tested a new version, it is often best to stick with an earlier one.

Different flavors of jQuery

There are now three branches of jQuery, called 1.x, 2.x, and 3.x, each designed for different environments.

Version 1.x was the first stable release of jQuery. This release supports older web browsers that aren't even supported by their respective developers any longer. If you expect a large number of visitors with older web browsers, this is the version to go for (as I write, version 1.12 is probably the best).

Version 2.x dropped support for Internet Explorer 6–8 to increase jQuery's overall performance and reduce the library's file size. It is faster and smaller than version 1.x but doesn't support older web browsers. Since Microsoft dropped support for Windows XP, it may be safe to assume that your visitors will be using a browser compatible with version 2.x, unless you know otherwise.

If you need to support older browsers like Internet Explorer 6–8, Opera 12.1x, or Safari 5.1+, the jQuery developers recommend using version 1.12. For full details on the different versions supported, please refer to the website. In this edition of the book, I have settled on version 3.5.1.

Compressed or editable

You also must decide whether to use a version of jQuery that has been minified (compressed in size to minimize bandwidth and decrease loading time) or an uncompressed version (perhaps because you want to make an edit to it yourself—which you are fully entitled to do). Generally, a minified version is best, but most web servers support *gzip* for on-the-fly compression and decompression, so this is becoming less important (although minification does remove comments too).

Downloading

The most recent released version of jQuery is available in both uncompressed and minified forms on the download page. You can also find all of the past releases on the jQuery CDN. The slim versions of jQuery that appear on the download page omit the asynchronous communication functions to save on space, so you should avoid these if you wish to make use of any Ajax functionality using jQuery.

All you need to do is choose the version you need, right-click the link displayed alongside it, and save it to your hard disk. From there, you can upload it to your web server and then include it within `<script>` tags, like this (for the minified version of release 3.7.0):

```
<script src='http://myserver.com/jquery-3.7.0.min.js'></script>
```

If you have never used jQuery before (and have no special requirements), then just download the latest minified version, or link to it via a CDN, as described in the following section.

Using a Content Delivery Network

Several CDNs support jQuery. If you use one of them, you can save yourself the hassle of having to download new versions and then upload them to your server by simply linking directly to the URLs supplied by these networks.

Not only that, but they provide this service free of charge, and usually on high-capacity backbones that are probably the fastest in the world. Additionally, CDNs usually hold their content in a number of different geographic locations and then supply the required file from the server closest to a user, ensuring the quickest possible delivery.

Overall, if you don't need to modify the jQuery source code (requiring you to host it on your own web servers), and your users are certain to have a live internet connection, using a CDN is probably the way to go. And it's very easy. All you need to know is the filename you wish to access and the root folder the CDN is using. For example, all current and previous versions can be accessed through the CDN that jQuery uses, like this:

```
<script src='http://code.jquery.com/jquery-3.7.0.min.js'></script>
```

The base directory is at *http://code.jquery.com/*, and you simply follow this with the name of the file you need to include (in this case, *jquery-3.7.0.min.js*).

Both Microsoft and Google offer jQuery on their networks, so you can also include it in either of the following ways:

```
<script src='http://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.7.0.min.js'></script>
<script src='http://ajax.googleapis.com/ajax/libs/jquery/3.7.0/jquery.min.js'>
</script>
```

In the case of the Microsoft CDN, you should begin the URL with the base directory of *ajax.aspnetcdn.com/ajax/jQuery/* and then follow that with the filename you require.

For Google, however, you must split the filename (for example, *jquery-3.7.0.min.js*) into a folder and filename (like this: *3.7.0/jquery.min.js*). Then precede that with *ajax.googleapis.com/ajax/libs/jquery/*.

An added benefit of using a CDN is that most other websites also do this, so jQuery may well already be cached in the user's browser and might not even require redelivery. With so many websites using jQuery, this can be a lot of valuable bandwidth and time saved.

Customizing jQuery

If it's absolutely critical that you keep the amount of data downloaded by a web page to the minimum, then you may still be able to use jQuery by making a special build of it that includes only the features your website will be using. You won't be able to rely on a CDN to deliver it, but in this circumstance you probably weren't planning on using one anyway.

To create your own custom build of jQuery, try jQuery Builder. Simply check the boxes you want and uncheck those that you don't. The bespoke version of jQuery will then be loaded into a separate tab or window, from where you can copy and paste it as required.

jQuery Syntax

The most striking thing about jQuery to people who are new to it is the `$` symbol, which acts as the jQuery factory method—the main means of access into the framework. It was chosen because the symbol is legal in JavaScript, is short, and is different from customary variable, object, or function/method names.

This symbol takes the place of making a call to the `jQuery` function (which you can also do if you wish). The idea is to keep your code short and sweet and to save on unnecessary extra typing each time you access jQuery. It also immediately shows other developers new to your code that jQuery (or a similar library) is in use.

A Simple Example

At its simplest, you access jQuery by typing a `$` symbol, followed by a selector in parentheses, and then a period and a method to apply to the selected element(s).

For example, to change the font family of all paragraphs to monospace, you could use this statement:

```
$('p').css('font-family', 'monospace')
```

Or to add a border to a `<code>` element, you could use this:

```
$('code').css('border', '1px solid #aaa')
```

Let's look at that as part of a complete example (see Example 25-1, where the jQuery parts are highlighted in bold).

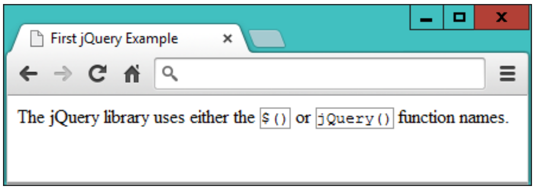
Example 25-1. A simple jQuery example

```
<!DOCTYPE html>
<html>
  <head>
    <title>First jQuery Example</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    The jQuery library uses either the <code>$()</code>
      or <code>jQuery()</code> function names.
    <script>
      $('code').css('border', '1px solid #aaa')
    </script>
  </body>
</html>
```

When you load this example into a browser, the result will be similar to Figure 25-1. Of course, this particular instruction simply replicates what you can do with normal CSS, but the idea is to illustrate jQuery syntax, so I'm keeping things simple for now.

Another way of issuing this command is by calling the `jQuery` function (which works in the same way as `$`), like this:

```
jQuery('code').css('border', '1px solid #aaa')
```



The jQuery library uses either the `$()` or `jQuery()` function names.

Figure 25-1. Modifying elements with jQuery

Avoiding Library Conflicts

If you use other libraries alongside jQuery, you may find that they define their own `$` functions. To resolve this issue, you can call the `noConflict` method on the symbol, which releases control so that the other library can take over, like this:

```
$.noConflict()
```

Once you do this, to access jQuery thereafter, you must call the `jQuery` function. Or you can replace the `$` symbol with an object name of your choice, like this:

```
jq = $.noConflict()
```

Now you can use the keyword `jq` to call jQuery wherever you had previously used `$`.

> To distinguish and keep track of jQuery objects separately from standard element objects, some developers prefix a `$` to any object created with jQuery (so that they end up looking like PHP variables!).

Selectors

Now that you've seen how easy it is to include jQuery in a web page and access its features, let's move on to looking at its selectors, which (I'm sure you'll be pleased to learn) work in exactly the same way as in CSS. In fact, selectors are at the heart of how most of jQuery operates.

All you have to do is think about how you would style one or more elements using CSS, and then you can use the same selector(s) to apply jQuery operations on these selected elements. This means you can make use of element selectors, ID selectors, class selectors, and any combinations thereof.

The css Method

To explain jQuery's use of selectors, let's first look at one of the more fundamental jQuery methods, `css`, with which you can dynamically alter any CSS property. It takes two arguments, the property name to be accessed and a value to be applied, like this:

```
css('font-family', 'Arial')
```

As you will see in the following sections, you cannot use this method on its own; you must append it to a jQuery selector, which will select one or more elements whose properties should be changed by the method. The following, which sets the content of all `<p>` elements to display with full justification, is an example:

```
$('p').css('text-align', 'justify')
```

You can also use the `css` method to return (rather than set) a computed value by supplying only a property name (and no second argument). In this case, the value of the first element that matches the selector is returned. For example, the following will return the text color of the element with the ID of `elem` as an `rgb` method:

```
color = $('#elem').css('color')
```

Remember that the value returned is the *computed* value. In other words, jQuery will compute and return the value as used by the browser at the moment the method is called, not the original value that may have been assigned to the property via a stylesheet or in any other way.

So, if the text color is blue (for example), the value assigned to the variable `color` in the preceding statement will be `rgb(0, 0, 255)`, even if the color was originally set with the color name `blue`, or the hex strings `#00f` or `#0000ff`. This computed value, though, will always be in a form that can be assigned back to the element (or any other element) via the second argument of the `css` method.

> Be wary of any computed dimensions returned by this method because, depending on the current `box-sizing` setting (see Chapter 19), they may or may not necessarily be what you expect. When you need to get or set widths and heights without consideration for `box-sizing`, you should use the `width` and `height` methods (and their siblings), as described in the section "Modifying Dimensions" on page 663.

The Element Selector

To select an element to be manipulated by jQuery, just list its name within the parentheses following the `$` symbol (or jQuery function name). For example, if you wanted to change the background color of all `<blockquote>` elements, you could use a statement such as the following:

```
$('blockquote').css('background', 'lime')
```

The ID Selector

You can also refer to elements by their IDs if you place a `#` character in front of the ID name. So, to add a border to the element with the ID of `advert` (for example), you could use this:

```
$('#advert').css('border', '3px dashed red')
```

The Class Selector

And you can manipulate groups of elements according to the class they use. For example, to underline all elements that use the class `new`, you could use this:

```
$('.new').css('text-decoration', 'underline')
```

Combining Selectors

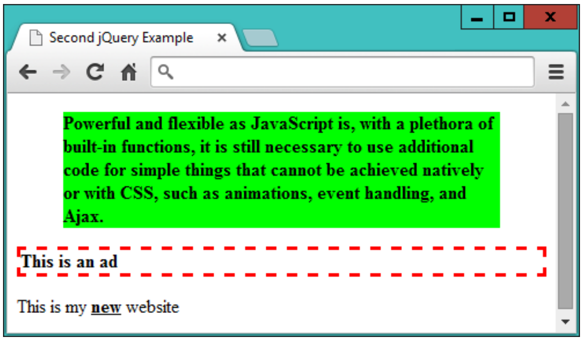
Just as with CSS, you may combine selectors into a single jQuery selection using commas, as with the following example:

```
$('blockquote, #advert, .new').css('font-weight', 'bold')
```

Example 25-2 brings all these types of selectors together into a single example (with the jQuery statements shown in bold), the result of which you can see in Figure 25-2.

Example 25-2. Using jQuery with different selectors

```
<!DOCTYPE html>
<html>
  <head>
    <title>Second jQuery Example</title>
    <script src='jquery-3.5.1.min.js'></script>
  </head>
  <body>
    <blockquote>Powerful and flexible as JavaScript is, with a plethora of
      built-in functions, it is still necessary to use additional code for
      simple things that cannot be achieved natively or with CSS, such as
      animations, event handling, and asynchronous communication.</blockquote>
    <div id='advert'>This is an ad</div>
    <p>This is my <span class='new'>new</span> website</p>
    <script>
      $('blockquote').css('background', 'lime')
      $('#advert').css('border', '3px dashed red')
      $('.new').css('text-decoration', 'underline')
      $('blockquote, #advert, .new').css('font-weight', 'bold')
    </script>
  </body>
</html>
```



Powerful and flexible as JavaScript is, with a plethora of built-in functions, it is still necessary to use additional code for simple things that cannot be achieved natively or with CSS, such as animations, event handling, and Ajax.

This is an ad

This is my **new** website

Figure 25-2. Manipulating multiple elements

Handling Events

If all jQuery could do was alter CSS styles, it wouldn't be a great deal of help—but of course, it can do far more than that. Let's further investigate by seeing how it handles events.

As you will recall, most events are triggered by user interaction: when a mouse passes over an element, the mouse button is clicked or a key is pressed. But there are also other events that can be triggered, such as when a document completes loading.

With jQuery, it's a simple matter to attach your own code to these events in a safe way that doesn't block other code from also gaining access to them. For example, here's how to get jQuery to respond to an element being clicked:

```
$('#clickme').click(function()
{
  $('#result').html('You clicked the button!')
})
```

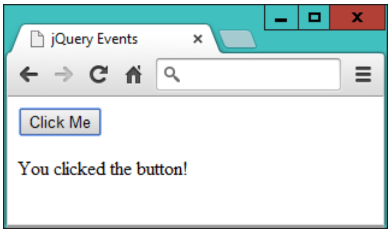
When the element with the ID of `clickme` is clicked, the `innerHTML` property of the element with the ID of `result` is updated via the jQuery `html` function.

jQuery objects (created with either the `$` or `jQuery` methods) are *not* the same as JavaScript objects created with `getElementById`. In plain JavaScript, you can use a statement like `object = document.getElementById('result')` followed by (for example) `object.innerHTML = 'something'`. But in the preceding example, `$('#result').innerHTML` would not work, because `innerHTML` is not a property of a jQuery object—hence the use of the jQuery method `html` to achieve the required result.

Example 25-3 fleshes out the idea; you can see it running in Figure 25-3.

Example 25-3. Processing an event

```
<!DOCTYPE html>
<html>
  <head>
    <title>jQuery Events</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <button id='clickme'>Click Me</button>
    <p id='result'>I am a paragraph</p>
    <script>
      $('#clickme').click(function()
      {
        $('#result').html('You clicked the button!')
      })
    </script>
  </body>
</html>
```



You clicked the button!

Figure 25-3. Processing a click event

When accessing an event with jQuery, omit the `on` prefix that you would use in standard JavaScript. So, for example, the `onmouseover` event name becomes the `mouseover` function in jQuery, `onclick` becomes `click`, and so on.

Waiting Until the Document Is Ready

Since jQuery is so closely related to the DOM in what it lets you achieve, more often than not you will need to wait until a web page has loaded before manipulating parts of it. Without jQuery, this can be achieved with the `onload` event, but there's a more efficient, cross-browser jQuery method called `ready` that you can call to enable it at the earliest possible moment (even sooner than with `onload`). This means jQuery can get working on a page that much more quickly, and user-unfriendly delays are minimized.

To make use of this feature, place your jQuery code within the following structure:

```
$('document').ready(function()
{
  // Your code goes here
})
```

The code will wait there until the document is ready, and only then will it be called by the `ready` method. In fact, there's a shorter version you can use that takes even less typing, as shown in Example 25-4.

Example 25-4. The smallest jQuery "ready" code encapsulation function

```
$(function()
{
  // Your code goes here
})
```

If you get used to encapsulating your jQuery statements in one of these two structures, you won't encounter the types of errors that can be generated by trying to access the DOM too soon.

Alternatively, another approach is to always place your JavaScript at the *end* of every HTML page so that it is executed only after the entire document has loaded. There is a secondary advantage too, in that this ensures the web page contents get priority with loading—thus, you may well see improvements in user experience.

The only time end-of-page scripts may not be a good idea is if a document could appear to be ready when it isn't, or if all external stylesheets have not yet loaded (which can really be identified only by testing), causing users to think they can interact with it before your script is ready. In such cases, implement the `ready` function and all will be well. In fact, if in doubt, place your script at the page end *and* place its jQuery calls within the `ready` function, and you'll get the best of both worlds.

Event Functions and Properties

You've just seen the `ready` event method, but you can access several dozen jQuery event methods and associated properties (far too many to detail here). However, the following are some of the ones that are more commonly used, and they'll get you started for most projects. For a comprehensive summary of all available events, please see the documentation.

The blur and focus Events

The `blur` event triggers when focus is removed from an element, causing it to blur, and is a good partner for the `focus` event. The `blur` and `focus` methods can be used to add a handler to the event. They will trigger the event if you omit any arguments from the method's parentheses.

In Example 25-5, there are four input fields. The first is given immediate focus with a quick call to the `focus` method, applying it to the element with the ID of `first`. Then a pair of handlers is added to all `input` elements. The `focus` handler sets their background to yellow when focus is given, and the `blur` handler sets their background to light gray when focus is removed (or blurred).

Example 25-5. Using the `focus` and `blur` events

```
<!DOCTYPE html>
<html>
  <head>
```

```
    <title>Events: blur</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <h2>Click in and out of these fields</h2>
    <input id='first'> <input> <input> <input>
    <script>
      $('#first').focus()
      $('input').focus(function() { $(this).css('background', '#ff0') } )
      $('input') .blur(function() { $(this).css('background', '#aaa') } )
    </script>
  </body>
</html>
```

You are allowed to include whitespace characters between the closing parenthesis of a method and the period operator used to attach another method to it (and after the period too, if you like), as in the previous example where I have right-aligned the `blur` event name under `focus`, to help the rest of the statements also line up in columns.

In Figure 25-4, you can see how this code gives any input fields that have ever had focus a light-gray background color. If one currently has focus, its background color is set to yellow, while unvisited fields remain with a white background color.

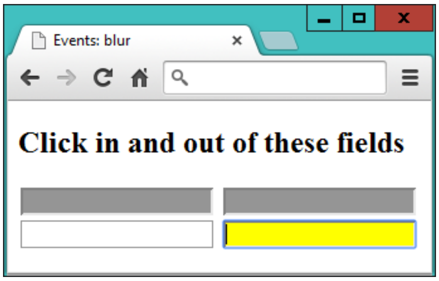


Figure 25-4. Attaching event handlers to the `blur` and `focus` events

The this Keyword

This example also serves to illustrate the use of the `this` keyword. When an event is called, the element on which it was triggered is passed in the object `this`, which can then be given to the `$` method for processing. Or, since `this` is a standard JavaScript object (and not a jQuery object), it can be used as such. So, if you prefer, you could replace this:

```
    $(this).css('background', '#ff0')
```

with this:

```
this.style.background = '#ff0'
```

The click and dblclick Events

You saw the `click` event a little earlier, but there's an event for handling double-clicks too. To use either, attach the event's method to a jQuery selection, and for its argument provide a jQuery method to invoke when the event triggers, like this:

```
$('.myclass')   .click( function() { $(this).slideUp() })
$('.myclass').dblclick( function() { $(this).hide()    })
```

Here I have opted to use inline anonymous functions, but you can use named ones instead if you like (just remember to supply only the name of the function, without parentheses, or it will be called at the wrong time). The `this` object will pass through as expected and be made available to the named function, like this:

```
$('.myclass').click(doslide)

function doslide()
{
  $(this).slideUp()
}
```

The `slideUp` and `hide` methods are detailed in the section "Special Effects" on page 649. For now, though, just try running Example 25-6 and either clicking or double-clicking the buttons to see how they sometimes disappear with an animation (using `slideUp`) and sometimes just vanish (using `hide`), as shown in Figure 25-5.

Example 25-6. Attaching to the `click` and `dblclick` events

```
<!DOCTYPE html>
<html>
  <head>
    <title>Events: click & dblclick</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <h2>Click and double click the buttons</h2>
    <button class='myclass'>Button 1</button>
    <button class='myclass'>Button 2</button>
    <button class='myclass'>Button 3</button>
    <button class='myclass'>Button 4</button>
    <button class='myclass'>Button 5</button>
    <script>
      $('.myclass').click(    function() { $(this).slideUp() })
      $('.myclass').dblclick( function() { $(this).hide()    })
    </script>
  </body>
</html>
```

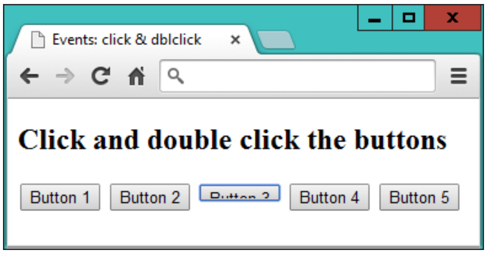


Figure 25-5. Button 3 has been clicked once and is sliding up

The keypress Event

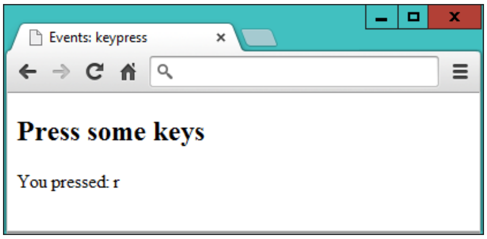
From time to time, you'll need better control over user keyboard interaction, particularly when processing complex forms or when writing games. For cases such as these, you can use the `keypress` method, which can be attached to anything that accepts keyboard input, such as an input field or even the document itself.

In Example 25-7, the method has been attached to the document in order to intercept all key presses. The result of running it can be seen in Figure 25-6.

Example 25-7. Intercepting key presses

```
<!DOCTYPE html>
<html>
  <head>
    <title>Events: keypress</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <h2>Press some keys</h2>
    <div id='result'></div>
    <script>
      $(document).keypress(function(event)
      {
        key = String.fromCharCode(event.which)

        if (key >= 'a' && key <= 'z' ||
            key >= 'A' && key <= 'Z' ||
            key >= '0' && key <= '9')
        {
          $('#result').html('You pressed: ' + key)
          event.preventDefault()
        }
      })
    </script>
  </body>
</html>
```



Press some keys

You pressed: r

Figure 25-6. Processing key presses from the keyboard

There are a few things of note in this example that you will need to bear in mind when writing your own keyboard handlers. For example, because browsers return differing values for this event, the `which` property of the `event` object is normalized by jQuery to return the same character codes across all browsers. So, this is where to look for which key was pressed.

Being a character code, though, the value in `which` is a number. You can turn it into a single-letter string by passing it through `String.fromCharCode`. You don't have to do this, because you can easily respond to ASCII values in your code, but this method is useful when you need to work with characters.

Within the `if` block, when a key press is recognized, the example code inserts a simple statement to that effect into the `innerHTML` property of the `<div>` element with the ID of `result`.

This is a good example of where the `document.write` function should not be used, because the document will have already fully loaded by the time the user presses a key. If `document.write` were called to display the information at this time, it would erase the current document. Writing into the HTML of an element instead is a perfect, nondestructive means of supplying user feedback, as explained in "About document.write" on page 345 in Chapter 14.

Considerate Programming

When you are anticipating user input, you should decide which values you will respond to and then ignore all others, just in case another event handler needs access to them. This is considerate practice for any other utilities (and the main browser itself) that may also be running. For example, in the preceding example I have chosen to accept only the characters a–z, A–Z, and 0–9, ignoring all others.

There are two ways you can pass keyboard interrupts on to (or deny them from) other handlers. First, do nothing; when your code exits, other handlers will also see and be able to react to the same key presses. This can cause confusion, though, if multiple actions occur from a single key press.

Alternatively, when you don't want the event to trigger any other handlers, you can make a call to the `preventDefault` method of `event`, which then keeps the event from "bubbling up" to other handlers.

> Be careful where you place your call to `preventDefault`—if it's outside the part of the code in which you process the key presses, it will prevent all other keyboard events from bubbling up, and you may lock the user out of the browser (or at least out of using certain features).

The mousemove Event

Some of the most commonly intercepted events are for mouse handling. I've already covered mouse button clicks, but now let's take a look at attaching to mouse movement events.

It's time, I think, for a slightly more interesting example, so in Example 25-8 I have put together a rudimentary drawing program using jQuery in conjunction with an HTML5 canvas. Although the canvas is not fully explained until Chapter 28, don't worry because the code is very simple.

Example 25-8. Intercepting mouse movement and mouse key events

```
<!DOCTYPE html>
<html>
  <head>
    <title>Events: Mouse Handling</title>
    <script src='jquery-3.7.0.min.js'></script>
    <style>
      #pad {
        background:#def;
        border    :1px solid #aaa;
      }
    </style>
  </head>
  <body>
    <canvas id='pad' width='480' height='320'></canvas>
    <script>
      canvas  = $('#pad')[0]
      context = canvas.getContext("2d")
      pendown = false
      $('#pad').mousemove(function(event)
      {
        var xpos = event.pageX - canvas.offsetLeft
        var ypos = event.pageY - canvas.offsetTop

        if (pendown) context.lineTo(xpos, ypos)
        else         context.moveTo(xpos, ypos)

        context.stroke()
      })

      $('#pad').mousedown(function() { pendown = true  } )
      $('#pad')  .mouseup(function() { pendown = false } )
    </script>
  </body>
</html>
```

In Figure 25-7, you can see how this fairly simple set of instructions can be used to create line drawings (well, if you have artistic ability, that is ☺). Here's how it works. First, the program creates a `canvas` object by referencing the first (or zeroth) element of the jQuery selector, like this:

```
    canvas = $('#pad')[0]
```

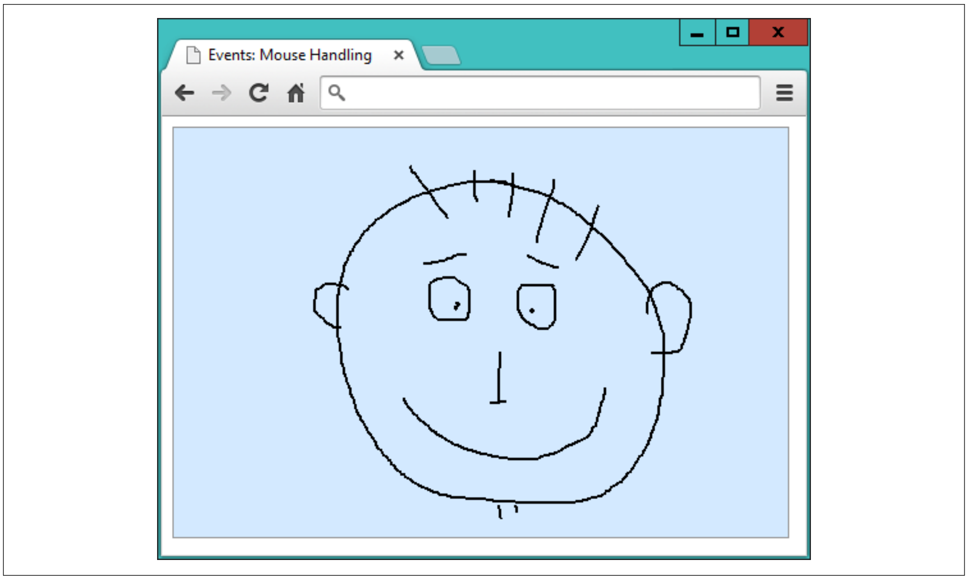


Figure 25-7. Capturing mouse movement and mouse key events

This is one way you can quickly take a jQuery object and extract a standard JavaScript element object. Another would be to use the `get` method, like this:

```
    canvas  = $('#pad').get(0)
```

The two are interchangeable, but `get` has the edge because, without any argument passed, it will return all the element node objects from a jQuery object as an array.

Anyway, as you'll learn in Chapter 28, the canvas will be written using a special `context` object, which is now created:

```
context = canvas.getContext("2d")
```

There's one more thing to initialize, which is a Boolean variable called `pendown` to track the status of the mouse button (initially `false` because the pen is up):

```
pendown = false
```

After this, the canvas (with the ID of `pad`) has its `mousemove` event intercepted by the anonymous function that follows, within which three sets of things happen:

```
$('#pad').mousemove(function(event)
{
  ...
})
```

First, the `xpos` and `ypos` local variables (local because of the `var` keywords, although better practice these days is to use `let` in place of `var`) are assigned values representing the position of the mouse within the canvas area.

These values are taken from the jQuery `pageX` and `pageY` properties, which refer to the mouse pointer's offset from the top-left-hand corner of the containing document. Therefore, since the canvas is itself offset slightly from that location, the `canvas` offset values (in `offsetLeft` and `offsetTop`) are subtracted from `pageX` and `pageY`:

```
var xpos = event.pageX - canvas.offsetLeft
var ypos = event.pageY - canvas.offsetTop
```

Now that we know where the mouse pointer is in relation to the canvas, the next pair of lines tests the value of `pendown`. If it is `true`, we know that the mouse button is being pressed, so a `lineTo` call is made to draw a line to the current location. Otherwise, the pen is up, and so `moveTo` is called to simply update the current location:

```
if (pendown) context.lineTo(xpos, ypos)
else         context.moveTo(xpos, ypos)
```

Then the `stroke` method is called to apply whichever drawing command was just made to the canvas. These five lines are all that is needed to handle drawing, but it's still necessary to track the mouse button state, so the final two lines of code intercept the `mousedown` and `mouseup` events, setting `pendown` to `true` when the mouse button is pressed and `false` when it's released:

```
$('#pad').mousedown(function() { pendown = true  } )
$('#pad')  .mouseup(function() { pendown = false } )
```

In this example, you see the combination of three different event handlers working together to create a simple utility, using both local variables for internal expressions and global variables where an object or the state of something must be made available across multiple functions.

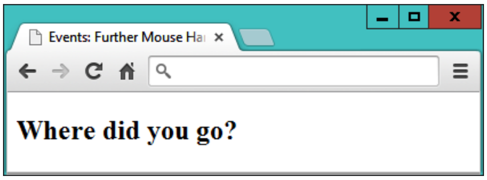
Other Mouse Events

The `mouseenter` and `mouseleave` events trigger whenever the mouse passes into an element or leaves it. No position values are supplied because the functions assume you simply want to make a Boolean decision about what to do when one of these events is triggered.

In Example 25-9, a pair of anonymous functions is attached to these events, altering the HTML of an element accordingly, as shown in Figure 25-8.

Example 25-9. Detecting the mouse entering and leaving an element

```
<!DOCTYPE html>
<html>
  <head>
    <title>Events: Further Mouse Handling</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <h2 id='test'>Pass the mouse over me</h2>
    <script>
      $('#test').mouseenter(function() { $(this).html('Hey, stop tickling!') } )
      $('#test').mouseleave(function() { $(this).html('Where did you go?')   } )
    </script>
  </body>
</html>
```



Where did you go?

Figure 25-8. Detecting when the mouse enters and leaves an element

When the mouse enters the bounds of the selected element, that element's `innerHTML` property is updated (with a call to `html`). Then, when the mouse leaves again, a further update is made to the element's HTML.

Alternative Mouse Methods

A number of other jQuery mouse event functions are available to cover a wide range of circumstances, all of which are detailed in the mouse events documentation. For example, you can use the following alternative `mouseover` and `mouseout` methods to achieve similar results to the code in the preceding section:

```
$('#test').mouseover(function() { $(this).html('Cut it out!')         } )
$('#test') .mouseout(function() { $(this).html('Try it this time...') } )
```

Or you could use the `hover` method to bind two handlers with a single function call, like this:

```
$('#test').hover(function() { $(this).html('Cut it out!')         },
                 function() { $(this).html('Try it this time...') } )
```

If you are planning on creating `mouseover` and `mouseout` combined effects, clearly the `hover` method is the logical function to choose—but there's also another way you can achieve the same result, which is *chaining* (explained in the section "Method chaining" on page 657), using code like this:

```
$('#test').mouseover(function() { $(this).html('Cut it out!')         } )
           .mouseout(function() { $(this).html('Try it this time...') } )
```

Here the period operator at the start of the second statement attaches it to the first, creating a chain of methods.

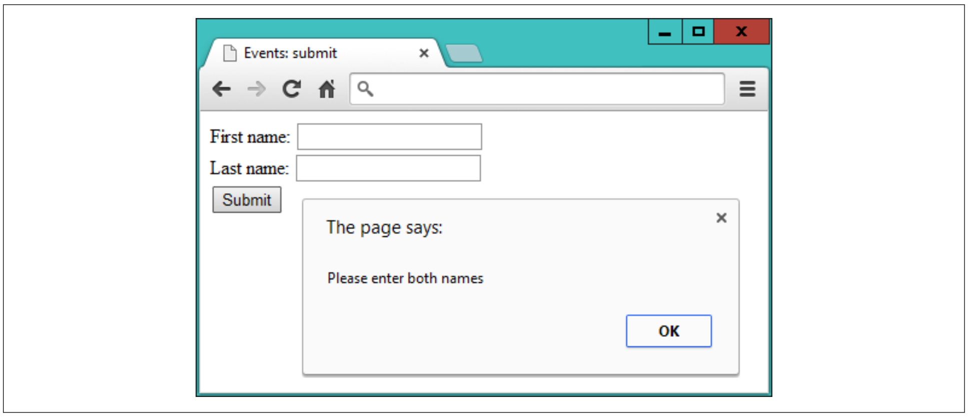
The preceding examples show how to capture mouse click, mouse movement, and keyboard events, and therefore they are mostly suitable for desktop environments—which is what jQuery is primarily targeted at. However, there is a version of jQuery for mobile devices that provides all the touch-handling event control you could wish for (and much more), called jQuery Mobile. We'll take a closer look at this in the next chapter.

The submit Event

When a form is submitted, you may want to perform some error checking on the data entered before it gets sent to the server. One way to do this is to intercept the `submit` event of the form, as in Example 25-10. Figure 25-9 shows the result of loading this document and then submitting the form with one or more fields left empty.

Example 25-10. Intercepting the `submit` event of a form

```
<!DOCTYPE html>
<html>
  <head>
    <title>Events: submit</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <form id='form'>
      First name: <input id='fname' type='text' name='fname'><br>
      Last name:  <input id='lname' type='text' name='lname'><br>
      <input type='submit'>
    </form>
    <script>
      $('#form').submit(function()
      {
        if ($('#fname').val() == '' ||
            $('#lname').val() == '')
        {
          alert('Please enter both names')
          return false
        }
      })
    </script>
  </body>
</html>
```



The page says:

Please enter both names

Figure 25-9. Checking user input upon submission

The key parts of this example are where the event is attached to an anonymous function, like this:

```
    $('#form').submit(function()
```

and where the values of the two input fields are tested for being empty:

```
    if ($('#fname').val() == '' ||
        $('#lname').val() == '')
```

Here the jQuery `val` method is used to retrieve the value in the `value` property of each field. This is neater than using `$('#fname')[0]` (as in Example 25-8) to get access to the DOM object and then appending `value` to it to read the field's value, like this: `$('#fname')[0].value`.

In this example, by returning the value `false` if one or more fields are empty, the `if` test cancels the normal submission process. To enable the submission to proceed, you can return `true`, or just not return anything.

Special Effects

jQuery really begins to come into its own when processing special effects. Although you can use CSS3 transitions, they are not always easy to manage dynamically from JavaScript—but with jQuery it really is as simple as selecting one or more elements and then applying one or more effects to them.

The core effects available are hiding and showing, fading in and out, sliding, and animations. These can be used singly, together in synchronization, or in sequence. They also support the use of callbacks, which are functions or methods that get called only once an operation has completed.

The following sections describe some of the more useful jQuery effects, each of which supports up to three arguments, as follows:

Duration
: When a duration value is supplied, the effect will take place over the assigned time, which can be a value in milliseconds or the strings `fast` or `slow`.

Easing
: There are only two easing options in the jQuery library, `swing` and `linear`. The default is `swing`, which gives a more natural effect than `linear`. For more easing options, you can check out plug-ins like jQuery UI.

Callback
: If you supply a callback function, it will be called after the effect method completes.

This means that when no arguments are given, the method is called immediately without being placed in the animation queue.

So, for example, you can call the `hide` method in a variety of ways, such as these:

```
$('#object').hide()
$('#object').hide(1000)
$('#object').hide('fast')
$('#object').hide('linear')
$('#object').hide('slow', 'linear')
$('#object').hide(myfunction)
$('#object').hide(333, myfunction)
$('#object').hide(200, 'linear', function() { alert('Finished!') } )
```

As you'll see in the section "Method chaining" on page 657, you can attach function calls (with arguments) to each other and they will then be animated in turn, like in the following example, which will hide and then reveal an element:

```
$('#object').hide(1000).show(1000)
```

Other less commonly used arguments are also supported by many of these methods, and you can get full details on them (and all the other supported effects methods) by visiting the documentation on effects.

Hiding and Showing

Probably the simplest effect is to hide and show elements in response to user interaction. As described in the previous section, you can provide no arguments or a variety of arguments to the `hide` and `show` methods. By default, when none are supplied, the result is to instantly hide or reveal an element.

When arguments are supplied, these two methods modify the `width`, `height`, and `opacity` properties of an element simultaneously, until they reach `0` for `hide` or their original values for `show`. The `hide` function sets the `display` property of the element to `none` when it is fully hidden, and the `show` function reassigns its previous value to it once the element is fully restored.

Example 25-11 lets you try `hide` and `show` for yourself (as shown in Figure 25-10).

Example 25-11. Hiding and showing an element

```
<!DOCTYPE html>
<html>
  <head>
    <title>Effects: hide &amp; show</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <button id='hide'>Hide</button>
    <button id='show'>Show</button>
    <p id='text'>Click the Hide and Show buttons</p>
    <script>
      $('#hide').click(function() { $('#text').hide('slow', 'linear') })
      $('#show').click(function() { $('#text').show('slow', 'linear') })
    </script>
  </body>
</html>
```

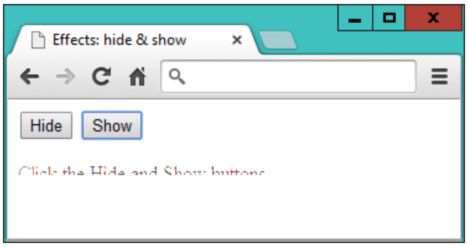


Figure 25-10. The element is in the process of being revealed

The toggle Method

As an alternative to calling both the `hide` and `show` methods, you can use the `toggle` method, which enables you to replace the previous example with Example 25-12.

Example 25-12. Using the `toggle` method

```
<!DOCTYPE html>
<html>
  <head>
    <title>Effects: toggle</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <button id='toggle'>Toggle</button>
    <p id='text'>Click the Toggle button</p>
    <script>
      $('#toggle').click(function() { $('#text').toggle('slow', 'linear') })
    </script>
  </body>
</html>
```

The `toggle` method takes the same arguments as `hide` and `show` but keeps track of the state of the element internally so that it knows whether to hide or show it.

There are four main jQuery methods that set either one state or another and that offer toggling versions to simplify coding. In addition to `toggle`, there's `fadeToggle`, `slideToggle`, and `toggleClass`, all described in this chapter.

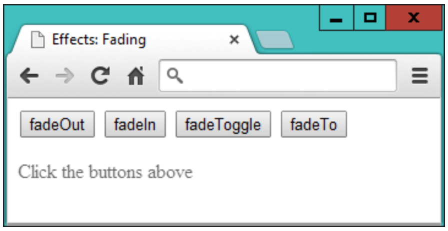
Fading In and Out

Four methods manage fades: `fadeIn`, `fadeOut`, `fadeToggle`, and `fadeTo`. You should be getting the idea of how jQuery works by now and realize that the first three are similar to `show`, `hide`, and `toggle`, respectively. The last one, though, is a little different in that it lets you specify an opacity value to which an element (or elements) should be faded, between `0` and `1`.

Example 25-13 provides four buttons with which to try out each of these methods, as shown in Figure 25-11.

Example 25-13. The four fade methods

```
<!DOCTYPE html>
<html>
  <head>
    <title>Effects: Fading</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <button id='fadeout'>fadeOut</button>
    <button id='fadein'>fadeIn</button>
    <button id='fadetoggle'>fadeToggle</button>
    <button id='fadeto'>fadeTo</button>
    <p id='text'>Click the buttons above</p>
    <script>
      $('#fadeout')   .click(function() { $('#text').fadeOut(   'slow'     ) })
      $('#fadein')    .click(function() { $('#text').fadeIn(    'slow'     ) })
      $('#fadetoggle').click(function() { $('#text').fadeToggle('slow'     ) })
      $('#fadeto')    .click(function() { $('#text').fadeTo(    'slow', 0.5) })
    </script>
  </body>
</html>
```



Click the buttons above

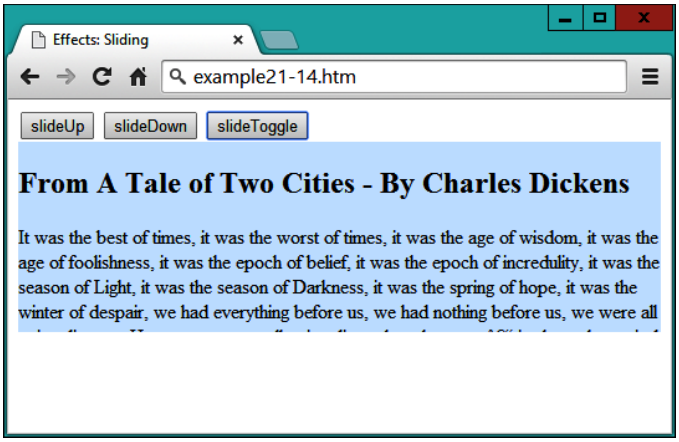
Figure 25-11. The text has been faded to 50% opacity

Sliding Elements Up and Down

Another way to make elements disappear and reappear is to alter their height over time to make them appear to slide up and down. There are three jQuery methods to do this: `slideDown`, `slideUp`, and `slideToggle`. They work in a similar way to the previous functions, as coded in Example 25-14 and shown in Figure 25-12.

Example 25-14. Using the slide methods

```
<!DOCTYPE html>
<html>
  <head>
    <title>Effects: Sliding</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <button id='slideup'>slideUp</button>
    <button id='slidedown'>slideDown</button>
    <button id='slidetoggle'>slideToggle</button>
    <div id='para' style='background:#def'>
      <h2>From A Tale of Two Cities - By Charles Dickens</h2>
      <p>It was the best of times, it was the worst of times, it was the age of
      wisdom, it was the age of foolishness, it was the epoch of belief, it was
      the epoch of incredulity, it was the season of Light, it was the season of
      Darkness, it was the spring of hope, it was the winter of despair, we had
      everything before us, we had nothing before us, we were all going direct to
      Heaven, we were all going direct the other way - in short, the period was so
      far like the present period, that some of its noisiest authorities insisted
      on its being received, for good or for evil, in the superlative degree of
      comparison only</p>
    </div>
    <script>
      $('#slideup')    .click(function() { $('#para').slideUp(    'slow') })
      $('#slidedown')  .click(function() { $('#para').slideDown(  'slow') })
      $('#slidetoggle').click(function() { $('#para').slideToggle('slow') })
    </script>
  </body>
</html>
```



From A Tale of Two Cities - By Charles Dickens

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all

Figure 25-12. The paragraph is sliding up

These methods work well when you have menus and submenus that you want to open up or close dynamically, according to the sections clicked by the user.

Animations

Now we can really start to have some fun by actually moving elements around in the browser. To do this, though, because the default value of `static` will not allow them to move, you must remember to first give your elements' `position` properties the values of `relative`, `fixed`, or `absolute`.

To animate an element, all you do is supply a list of CSS properties (excluding colors) to the `animate` method. Unlike the previous effects methods, `animate` requires this list of properties first, and then you can supply any duration, easing, and callback arguments you need.

So, for example, to animate a bouncing ball, you could use code such as that in Example 25-15 (which displays like Figure 25-13).

Example 25-15. Creating a bouncing ball animation

```
<!DOCTYPE html>
<html>
  <head>
    <title>Effects: Animation</title>
    <script src='jquery-3.7.0.min.js'></script>
    <style>
      #ball {
        position  :relative;
      }
      #box {
        width     :640px;
        height    :480px;
        background:green;
        border    :1px solid #444;
      }
    </style>
  </head>
  <body>
    <div id='box'>
      <img id='ball' src='ball.png'>
    </div>
    <script>
      bounce()

      function bounce()
      {
        $('#ball')
          .animate( { left:'270px', top :'380px' }, 'slow', 'linear')
          .animate( { left:'540px', top :'190px' }, 'slow', 'linear')
          .animate( { left:'270px', top :'0px'   }, 'slow', 'linear')
          .animate( { left:'0px',   top :'190px' }, 'slow', 'linear')
      }
    </script>
  </body>
</html>
```

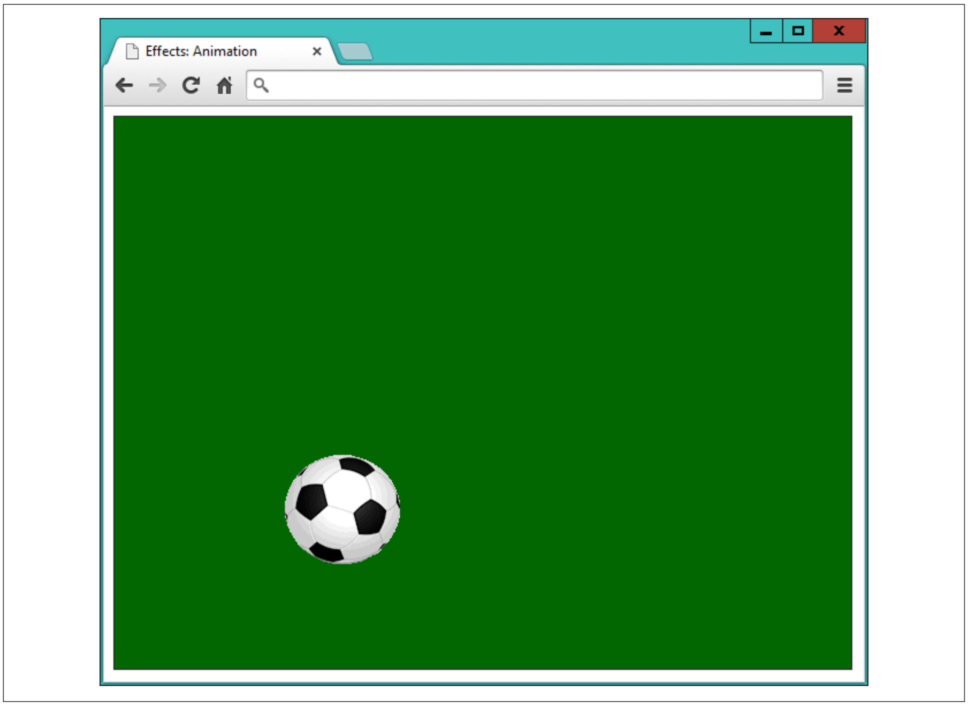


Figure 25-13. The ball is bouncing around in the browser

In the `<style>` section of this example, the `position` property of the ball is set relative to its container, which is a `<div>` element that has been given a border and a green background.

Then the `<script>` section features a function called `bounce`, which concatenates four calls to `animate`. Notice how the names of the properties to animate (`left` and `top` in this example) are supplied without quotation marks and are separated from the values to which they should be changed (such as `'270px'`) with colons—in other words, in the form of associative arrays.

You can also supply relative values in place of absolute ones by using the `+=` and `-=` operators. So, for example, the following will animate the ball to the right and up by 50 pixels relative to its current position:

```
.animate( { left:'+=50px', top:'-=50px' }, 'slow', 'linear')
```

And you can even use the string values of `hide`, `show`, and `toggle` to update a property, like this:

```
.animate( { height:'hide', width:'toggle' }, 'slow', 'linear')
```

If you wish to modify any hyphenated CSS properties and they're not being passed within quotation marks (as with `height` and `width` in the previous example), you must convert their names to camelCase by removing the hyphens and capitalizing the letter following. For example, to animate the `left-margin` property of an element, you would supply the name of `leftMargin`. However, when supplying a hyphenated property name within a string—for example, `css('font-weight', 'bold')`, you *shouldn't* convert it to camelCase.

Method chaining

Because of the way that method chaining works, when jQuery methods have been given arguments, they will run sequentially. So, each of these methods is called only after the previous one has finished animating. Any methods you call without arguments, however, will run immediately and quickly, without animation.

When you load Example 25-15 into a web browser, the animation is kicked off (so to speak) with a single call to `bounce`, causing the ball to bounce off the bottom, right, and top edges of its container and then come to rest back at the middle of the left-hand edge. Looking again at the `bounce` function in this example, you can see there are four chained calls to the `animate` function.

Using callbacks

As it stands, the preceding example stops after four animations, but you can use a callback function to make the animation start over again every time it completes. This is why I chose to place the animation in a named function.

With the animation in the `bounce` function, it is necessary only to add that name as a callback to the fourth animation in the group in order to make the animation repeat forever, as shown in bold here:

```
.animate( { left:'0px', top :'190px' }, 'slow', 'linear', bounce)
```

Using the `animate` method, you can animate many CSS properties, with the notable exception of colors. However, even color animation is possible with the addition of the jQuery UI add-in, which provides the facility to create very eye-pleasing color-changing effects (plus many more goodies). Please refer to the jQuery UI page for details.

Stopping Animations

There are several methods available for cutting off animations midway through, or ending a chain of animations. For example, `clearQueue` can empty all stored animations from the queue, `stop` can immediately stop any animation currently in progress, and `finish` will stop the currently running animation and remove any that are queued.

Let's turn the previous example into a sort of game by making the ball clickable, such that when the click event triggers, the animation will cease. To do this, all that's required is to add the following single line of code underneath the `bounce` function:

```
$('#ball').click(function() { $(this).finish() })
```

If you successfully manage to click the ball, the `finish` method will stop the current animation, empty the queue, and ignore any callbacks—in other words, the ball will come to rest.

For more information on managing jQuery queues, check out the documentation for the `queue` method, where you'll also learn how to directly manipulate the contents of queues to get exactly the effects you need.

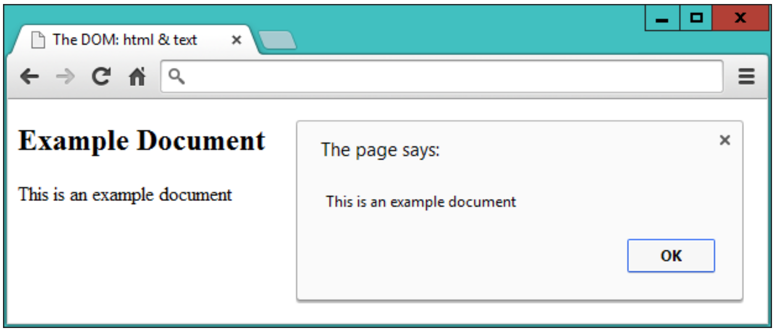
Manipulating the DOM

Because jQuery is so tightly linked with the DOM, of necessity the examples in this chapter have already utilized some of its DOM-accessing methods, such as `html` and `val`. But let's now look at all the DOM methods in detail to discover exactly what you can get access to with jQuery, and how.

In Example 25-3, you saw how to use the `html` method to change the `innerHTML` property of an element. This method can be used either to set the HTML or retrieve it from an HTML document. Example 25-16 (with the jQuery highlighted in bold) shows how to retrieve the HTML content of an element (as shown in Figure 25-14).

Example 25-16. Displaying the HTML of an element using an alert window

```
<!DOCTYPE html>
<html>
  <head>
    <title>The DOM: html & text</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <h2>Example Document</h2>
    <p id='intro'>This is an example document</p>
    <script>
      alert($('#intro').html())
    </script>
  </body>
</html>
```



Example Document

This is an example document

The page says:

This is an example document

Figure 25-14. Retrieving and displaying the HTML of an element

When you issue this method without arguments, the result is to read rather than set the HTML of the element.

The Difference Between the text and html Methods

When you are working with XML documents, you cannot use the `html` method because it simply won't work (it's designed for use only with HTML). But you can use the `text` method to achieve a similar result (in XML or HTML documents), like this:

```
text = $('#intro').text()
```

The difference between the two is quite simply that `html` treats the content as HTML and `text` treats it as text. So, for example, let's assume you wish to assign the following string to an element:

```
<a href='http://google.com'>Visit Google</a>
```

If you assign it to an HTML element using the `html` method, the DOM will be updated with the new `<a>` element and the link will become clickable. But if you do so to either an XML or HTML document using the `text` method, that string will first be escaped into text (for example, by converting HTML characters such as `<` into the `&lt;` entity, and so on) and then inserted into the element—no element is added to the DOM.

The val and attr Methods

There are a couple more methods for interacting with the content of elements. First, you can set and get the value of an input element with the `val` method, as illustrated in Example 25-10, in which the first and last name fields are read. To set a value, simply provide it as an argument to the method, like this:

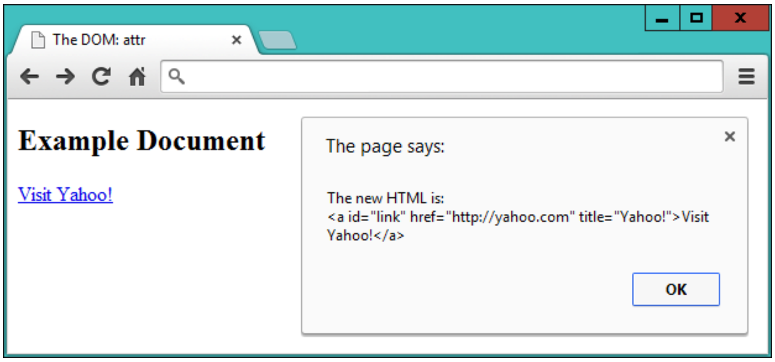
```
$('#password').val('mypass123')
```

With the `attr` method, you can get and set the attributes of elements, as shown in Example 25-17, in which a link to the Google website has been completely replaced with one to Yahoo.

Example 25-17. Modifying attributes with the `attr` method

```
<!DOCTYPE html>
<html>
  <head>
    <title>The DOM: attr</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <h2>Example Document</h2>
    <p><a id='link' href='http://google.com' title='Google'>Visit Google</a></p>
    <script>
      $('#link').text('Visit Yahoo!')
      $('#link').attr( { href :'http://yahoo.com', title:'Yahoo!' } )
      alert('The new HTML is:\n' + $('p').html())
    </script>
  </body>
</html>
```

The first jQuery statement uses the `text` method to change the text inside the `<a>` element, and the second one changes the `href` and `title` attribute values to match by supplying the data in the form of an associative array. The third statement displays the changed element's contents in an alert window by first retrieving it with the `html` method, as shown in Figure 25-15.



```
The new HTML is:
<a id="link" href="http://yahoo.com" title="Yahoo!">Visit Yahoo!</a>
```

Figure 25-15. The link has now been completely modified

You can also read the value of an attribute like this:

```
url = $('#link').attr('href')
```

Adding and Removing Elements

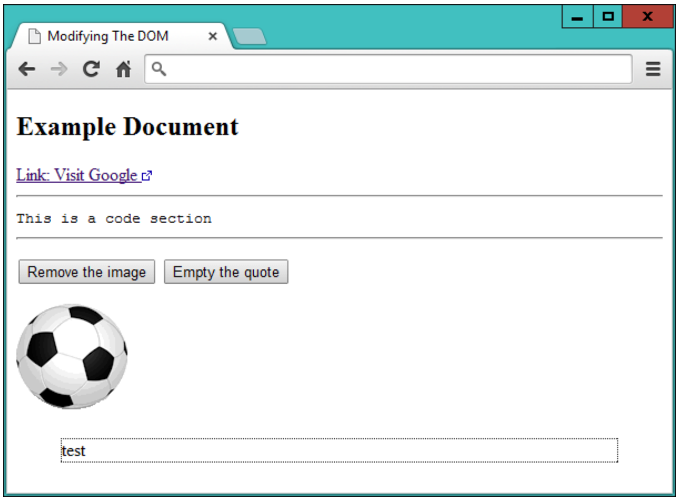
While it is possible to insert elements into the DOM using the `html` method, this is suitable only for creating child elements of a particular element. Therefore, jQuery provides a number of methods for manipulating any part of the DOM.

These methods are `append`, `prepend`, `after`, `before`, `remove`, and `empty`. An example of each appears in Example 25-18.

Example 25-18. Adding and removing elements

```
<!DOCTYPE html>
<html>
  <head>
    <title>Modifying The DOM</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <h2>Example Document</h2>
    <a href='http://google.com' title='Google'>Visit Google</a>
    <code>
      This is a code section
    </code>
    <p>
      <button id='a'>Remove the image</button>
      <button id='b'>Empty the quote</button>
    </p>
    <img id='ball' src='ball.png'>
    <blockquote id='quote' style='border:1px dotted #444; height:20px;'>
      test
    </blockquote>
    <script>
      $('a').prepend('Link: ')
      $("[href^='http']").append(" <img src='link.png'>")
      $('code').before('<hr>').after('<hr>')
      $('#a').click(function() { $('#ball').remove() } )
      $('#b').click(function() { $('#quote').empty() } )
    </script>
  </body>
</html>
```

In Figure 25-16, you can see the result of applying the `prepend`, `append`, `before`, and `after` methods to some elements.



Example Document

Link: Visit Google

This is a code section

test

Figure 25-16. A document with a variety of elements

The `prepend` method has been used to insert the string `Link:` before the inner text or HTML of all `<a>` elements, like this:

```
$('a').prepend('Link: ')
```

Then an attribute selector is used to select all elements that have an `href` attribute starting with `http`. The string `http` appearing at the beginning of the URL (because of the `^=` operator) denotes links that are not relative and therefore are absolute. In these cases, an external link icon is appended to the end of the inner text or HTML of all matching elements, like this:

```
$("[href^='http']").append(" <img src='link.png'>")
```

The `^=` operator is how only the start of the string is matched. If just the `=` operator were used, only entire strings that matched would be selected. CSS selectors are covered in detail in Chapters 24 and 19.

Next, using chained methods, the `before` and `after` methods are employed to place sibling elements either before or after another one. In this instance, I have chosen to place an `<hr>` element both before and after `<code>` elements, like this:

```
$('code').before('<hr>').after('<hr>')
```

Then I added a little user interaction with a couple of buttons. When clicked, using the `remove` method, the first button removes the `<img>` element containing the ball, like this:

```
$('#a').click(function() { $('#ball').remove() } )
```

The image is now no longer in the DOM, which you can verify if you right-click and choose Inspect Element in most major desktop browsers.

Finally, the `empty` method is applied to the `<blockquote>` element when the second button is clicked, like this:

```
$('#b').click(function() { $('#quote').empty() } )
```

This empties out the element's contents but leaves the element in the DOM.

Dynamically Applying Classes

Sometimes it can be convenient to change the class applied to an element, or maybe just add a class to an element or remove it from one. For example, suppose you have a class called `read` that you use to style blog posts that have been read. Using the `addClass` method, it's a simple matter to add a class to that post, like this:

```
$('#post23').addClass('read')
```

You can add more than one class at a time by separating them with spaces, like this:

```
$('#post23').addClass('read liked')
```

But what if a reader chooses to mark a post as unread again, perhaps to be reminded to read it again later? In this case, all you need to do is use `removeClass`, like this:

```
$('#post23').removeClass('read')
```

All other classes that the post uses remain unaffected when you do this.

Where you are supporting the ability of a class to be continuously added or removed, however, you might find it simpler to use the `toggleClass` method, like this:

```
$('#post23').toggleClass('read')
```

Now, if the post doesn't already use the class, it is added; otherwise, it is removed.

Modifying Dimensions

Working with dimensions is always a tricky web development task because different browsers tend to utilize slightly differing values. One of jQuery's big strengths, therefore, is that it does a great job of normalizing these types of values so that your pages will look how you intend them to appear in all major browsers.

There are three types of dimensions: element width and height, inner width and height, and outer width and height. Let's look at these in turn.

The width and height Methods

Both the `width` and `height` methods can get the width or height of the first element that matches a selector, or set the width or height of all matching elements. For example, to get the width of an element with the ID of `elem`, you could use this statement:

```
width = $('#elem').width()
```

The value returned to the variable `width` is a plain numeric value, which is different from returning the CSS value from a call to the `css` method, as in the following example, which would return (for example) `230px` rather than just the number `230`:

```
width = $('#elem').css('width')
```

You can also obtain the width of either the current window or the document, like this:

```
width = $(window).width()
width = $(document).width()
```

When you pass the `window` or `document` objects to jQuery, you cannot fetch their width or height with the `css` method. Instead, you must use the `width` or `height` methods.

The value returned is independent of the `box-sizing` setting (see Chapter 19). If you need to take `box-sizing` into account, use the `css` method with an argument of `width` instead, like this (but remember to remove from the returned value the `px` that will be added after the numeric part if you intend to work with the values returned):

```
width = $('#elem').css('width')
```

Setting values is just as easy. For example, to set all elements that use the class `box` to 100 × 100 pixels, you could use this statement:

```
$('.box').width(100).height(100)
```

Example 25-19 combines these actions into a single program that displays as Figure 25-17.

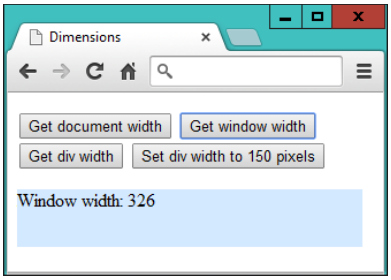
Example 25-19. Getting and setting element dimensions

```
<!DOCTYPE html>
<html>
  <head>
    <title>Dimensions</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <p>
      <button id='getdoc'>Get document width</button>
      <button id='getwin'>Get window width</button>
      <button id='getdiv'>Get div width</button>
      <button id='setdiv'>Set div width to 150 pixels</button>
    </p>
    <div id='result' style='width:300px; height:50px; background:#def;'></div>
    <script>
      $('#getdoc').click(function()
      {
        $('#result').html('Document width: ' + $(document).width())
      } )

      $('#getwin').click(function()
      {
        $('#result').html('Window width: ' + $(window).width())
      } )

      $('#getdiv').click(function()
      {
        $('#result').html('Div width: ' + $('#result').width())
      } )

      $('#setdiv').click(function()
      {
        $('#result').width(150)
        $('#result').html('Div width: ' + $('#result').width())
      } )
    </script>
  </body>
</html>
```



Window width: 326

Figure 25-17. Getting and setting element dimensions

At the start of the `<body>`, there are four buttons: three for reporting the widths of the document, window, and a `<div>` element that appears just below the buttons; and one to set the `<div>`'s width to a new value. In the `<script>` section, there are four jQuery statements, the first three of which simply fetch the widths of the objects given and then report these values by writing into the HTML of the `<div>`.

The final statement has two parts: the first reduces the width of the `<div>` element to 150 pixels, and the second displays the new width value within the `<div>` by fetching it using the `width` method, to ensure the computed value is displayed.

When the page has been zoomed (either in or out) by the user, this event is not noted in any major browser in any way that JavaScript can reliably detect. Therefore, jQuery cannot take zooming into account when it applies or returns dimensional values, so it is possible to get unexpected results in this circumstance.

The innerWidth and innerHeight Methods

It's often necessary to also take borders, padding, and other properties into account when working with dimensions. For these purposes, you can use the `innerWidth` and `innerHeight` methods to return the width and height of the first element that matches the selector, *including* the padding but *not including* any border.

For example, the following returns the `innerWidth` of the element with an ID of `elem`, including padding:

```
iwidth = $('#elem').innerWidth()
```

The outerWidth and outerHeight Methods

To return the dimensions of an element including *both* padding and border, you can call the `outerWidth` and `outerHeight` methods, like this:

```
owidth = $('#elem').outerWidth()
```

If you wish to *also* include any margin in the returned value, you can pass the value of `true` when you call either of these methods, like this:

```
owidth = $('#elem').outerWidth(true)
```

The values returned for any of the `inner...` or `outer...` methods are not necessarily integers and may be fractional in some cases. User page zooming is not detected by these methods, and you cannot use these methods on `window` or `document` objects—for those, use the `width` or `height` methods instead.

DOM Traversal

If you refer back to the section on the Document Object Model in Chapter 14, you'll recall that all web pages are constructed in much the same way as extended families. There are parent and child objects, siblings, grandparents, grandchildren, and even element relationships that could be referred to as cousins, aunts, and so forth. For example, in the following snippet, the `<li>` elements are children of the `<ul>` element, which, in turn, is parent to the `<li>` elements:

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
```

And, like in families, there are multiple ways you can refer to HTML elements, such as with absolute specifications or by starting at the window level and moving on down (also known as *traversing the DOM*). You can also use the relationship between one element and another to refer to elements. Really, it's a matter of what makes sense for your particular project—for example, if you want a web page to be as self-contained as possible so that you have a better chance of cutting and pasting elements into other web documents without having to change the pasted HTML to match that of the destination. Whatever you choose, though, jQuery offers a wide range of functions to help you accurately address elements.

Parent Elements

To refer to an element's direct parent, use the `parent` method like this:

```
my_parent = $('#elem').parent()
```

Whatever type of element `elem` may be, the `my_parent` object now contains a jQuery object that refers to its parent element. In fact, since selectors can refer to multiple elements, this call actually returns an object that refers to a list of parent elements (although the list can have only one item), one for each matching element.

Since a parent may have many children, you might wonder whether more elements can be returned by this method than there are parents. Take the preceding snippet with three `<li>` elements. If we do this:

```
my_parent = $('li').parent()
```

will three parent elements be returned (because three matches will be made), even though there is just a single `<ul>` parent? The answer is no, because jQuery is smart enough to recognize all duplicates and filter them out. To verify this, if you ask for the number of elements returned like this, the result will be `1`:

```
alert($('li').parent().length)
```

Let's now make something happen when the selector matches, such as changing the `font-weight` property of the parent element in the previous snippet to `bold`, like this:

```
$('li').parent().css('font-weight', 'bold')
```

Using a filter

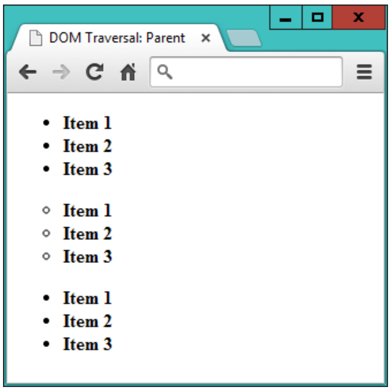
Optionally, a selector can be passed to `parent` to filter out which of the parents should reflect the desired changes. To illustrate, Example 25-20 has three small lists and a couple of jQuery statements.

Example 25-20. Accessing parent elements

```
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Traversal: Parent</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </ul>
    <ul class='memo'>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </ul>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </ul>
    <script>
      $('li').parent()       .css('font-weight',     'bold')
      $('li').parent('.memo').css('list-style-type', 'circle')
    </script>
  </body>
</html>
```

The three lists are all the same, except that the middle one's `<ul>` element uses a class of `memo`. In the `<script>` section, the first statement applies a value of `bold` to the `font-weight` property of all parents of `<li>` elements. In this instance, it causes all the `<ul>` elements to display in bold.

The second statement is similar but also passes the class name `memo` to the parent method so that only that parent will be selected. Then the `css` method is called to set the `list-style-type` property of the selected list to `circle`. Figure 25-18 shows the effect of these two statements.



- Item 1
- Item 2
- Item 3

- Item 1
- Item 2
- Item 3

- Item 1
- Item 2
- Item 3

Figure 25-18. Accessing parent elements with and without filters

Selecting all ancestor elements

We've just seen how to select direct parents of elements, but you can also select ancestors, all the way back to the `<html>` root element, by using the `parents` method. But why might you want to do this? Well, how about wanting to access the first `<div>` element up the ancestry chain in order to style it according to something dynamic that has gone on further down the chain?

This type of selection may be slightly too advanced for any purpose you can think of right now, but you'll be pleased it's there when you need it, and here's how you might proceed:

```
$('#elem').parents('div').css('background', 'yellow')
```

Actually, that may not be exactly what you want, because it will select all `<div>` elements in the ancestry chain, and there may be others further up that you don't want to style. So, for this type of eventuality, you can further filter the selection by using the `parentsUntil` method instead.

The `parentsUntil` method traverses up the ancestry chain in the same way as `parents` but stops at the first element that matches the selection filter (in this case, it's a `<div>` element), so you can use it in the same way as the preceding statement, certain in the knowledge that you will select only the closest matching element you want:

```
$('#elem').parentsUntil('div').css('background', 'yellow')
```

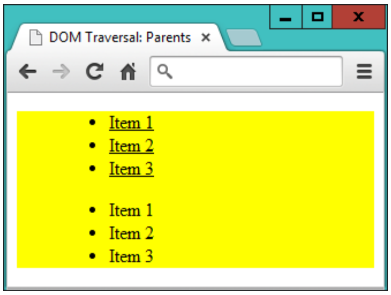
To illustrate the difference between these two methods, take a look at Example 25-21, which contains two sets of nested elements, both of which are within one parent

<div> element. The <script> section calls one example each of the parents and parentsUntil methods.

Example 25-21. Using the parents and parentsUntil methods

```
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Traversal: Parents</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <div>
      <div>
        <section>
          <blockquote>
            <ul>
              <li>Item 1</li>
              <li id='elem'>Item 2</li>
              <li>Item 3</li>
            </ul>
          </blockquote>
        </section>
      </div>
      <div>
        <section>
          <blockquote>
            <ul>
              <li>Item 1</li>
              <li>Item 2</li>
              <li>Item 3</li>
            </ul>
          </blockquote>
        </section>
      </div>
    </div>
    <script>
      $('#elem').parents('div')     .css('background',      'yellow')
      $('#elem').parentsUntil('div').css('text-decoration', 'underline')
    </script>
  </body>
</html>
```

If you take a look at Figure 25-19, you'll see that the first jQuery statement has set the background color of all the contents to yellow. This is because the ancestry tree has been traversed all the way up to the <html> element using the parents method, and both <div> elements encountered on the way up the tree have been selected (the one containing the list with the <li> element—highlighted in bold—with the ID of elem *and* its parent <div>, which contains both sets of nested elements).



- Item 1
- Item 2
- Item 3

- Item 1
- Item 2
- Item 3

Figure 25-19. Comparing the `parents` *and* `parentsUntil` *methods*

However, the second statement uses `parentsUntil` so that the selection stops at the first `<div>` element encountered. This means that when the underline styling is applied, it is applied only to the closest parent `<div>` that contains the `<li>` element with the ID of `elem`. The outer `<div>` is not reached, and because it doesn't get styled, the second list isn't underlined.

Child Elements

To access the children of an element, use the `children` method, like this:

```
my_children = $('#elem').children()
```

Like the `parent` method, this goes down only one level and returns a list of zero, one, or more matching selections. You can also pass a filter argument to it to select between the children, like this:

```
li_children = $('#elem').children('li')
```

Here, only children that are `<li>` elements will be selected.

To delve deeper down the generations, you need to use the `find` method, which is the inverse of `parents`, like this:

```
li_descendants = $('#elem').find('li')
```

However, unlike with `parents`, you *must* provide a filter selector to the `find` method. If you need to select all descendants, you can use the universal selector, like this:

```
all_descendants = $('#elem').find('*')
```

Sibling Elements

When it comes to selecting siblings, there's an even wider range of methods available, starting with `siblings`.

The `siblings` method will return all the matching elements that are children of the same parent, *except* for the element used for selecting. So, taking the example of the following snippet, if you look up the siblings of the `<li>` element with the ID of `two`, it will return only the first and third `<li>` elements:

```
<ul>
  <li>Item 1</li>
  <li id='two'>Item 2</li>
  <li>Item 3</li>
</ul>
```

For example, the following statement will cause the first and third sibling elements to be bold:

```
$('#two').siblings().css('font-weight', 'bold')
```

You can also use a filter in the `siblings` method to further narrow down the siblings returned. For example, to select only those siblings that use the class `new`, you might use a statement such as this:

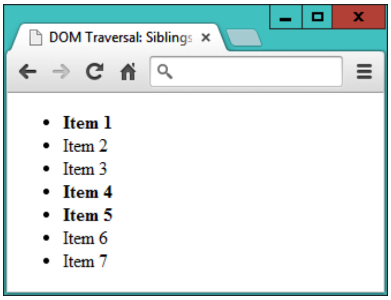
```
$('#two').siblings('.new').css('font-weight', 'bold')
```

Example 25-22 (liberally whitespaced to line up the attributes in columns) displays an unordered list of seven items, of which four use the class `new`. The second item also has the ID of `two`.

Example 25-22. Selecting and filtering sibling elements

```
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Traversal: Siblings</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <ul>
      <li          class='new'>Item 1</li>
      <li id='two' class='new'>Item 2</li>
      <li                     >Item 3</li>
      <li          class='new'>Item 4</li>
      <li          class='new'>Item 5</li>
      <li                     >Item 6</li>
      <li                     >Item 7</li>
    </ul>
    <script>
      $('#two').siblings('.new').css('font-weight', 'bold')
    </script>
  </body>
</html>
```

When loaded into a browser, the jQuery statement results in Figure 25-20, in which only Item 1, Item 4, and Item 5 are in bold, even though Item 2 also uses the class new (because the method is called on that element, and so it is excluded from the selection).



- **Item 1**
- Item 2
- Item 3
- **Item 4**
- **Item 5**
- Item 6
- Item 7

Figure 25-20. Selecting sibling elements

Since the siblings method omits the element on which it is called (which I will refer to as the *callee*), it cannot be used to select from *all* children of a parent element. However, to achieve that with the preceding example, you could use a statement such as the following, which will return all siblings (including the callee) that use the class new:

```
$('#two').parent().children('.new')
.css('font-weight', 'bold')
```

You can alternatively add the addBack method to the selection to achieve the same result, like this:

```
$('#two').siblings('.new').addBack()
.css('font-weight', 'bold')
```

Selecting the Next and Previous Elements

When you need finer control over the selection of siblings, you can further narrow down the elements returned using the next and prev methods and their extended versions. For example, to refer to the element immediately following a selector, you can use a statement such as this (which sets the matched element(s) to display in bold):

```
$('#new').next().css('font-weight', 'bold')
```

In the case of the following liberally whitespaced snippet, for example, the third item has the ID new, and therefore the fourth item is returned:

```
<ul>
  <li         >Item 1</li>
  <li         >Item 2</li>
  <li id='new'>Item 3</li>
  <li         >Item 4</li>
  <li         >Item 5</li>
</ul>
```

So far, so simple. But what if you would like to reference *all* the siblings following a particular element? Well, you can do that with the `nextAll` method, like this (which in the preceding snippet would style the last two items):

```
$('#new').nextAll().css('font-weight', 'bold')
```

When calling `nextAll`, you can also supply a filter to select from the elements that are matched, such as in the following example, which will style only the following siblings that use the class `info` (in the previous snippet, however, there are no elements that use that class, so the statement will do nothing):

```
$('#new').nextAll('.info').css('font-weight', 'bold')
```

Or consider the case of this snippet, in which one item has the ID `new` and another has the ID `old`:

```
<ul>
  <li         >Item 1</li>
  <li id='new'>Item 2</li>
  <li         >Item 3</li>
  <li id='old'>Item 4</li>
  <li         >Item 5</li>
</ul>
```

Now it's possible to select only those siblings following the one with the ID of `new`, up to (but not including) the one with the ID of `old`, like this (in which just the third item will be styled):

```
$('#new').nextUntil('#old').css('font-weight', 'bold')
```

If no argument is supplied to `nextUntil`, it behaves exactly like `nextAll`, returning all the following siblings. You can also supply a second argument to `nextUntil` to act as a filter for selecting from the elements that are matched by it, like this:

```
$('#new').nextUntil('#old', '.info').css('font-weight', 'bold')
```

In this statement, only those elements that use the class `info` will be styled, which in the case of the preceding snippet is none of them, so no action will be taken.

You can do exactly the same, working backward through groups of siblings, by using the `prev`, `prevAll`, and `prevUntil` methods.

Traversing jQuery Selections

In addition to traversing the DOM, once you have returned a set of elements as a jQuery selection you can also traverse those elements, choosing which ones to act on.

For example, to style just the first element returned by a selection, you can use the `first` method, like this (to set the first list item in the first unordered list to display underlined):

```
$('ul>li').first().css('text-decoration', 'underline')
```

Or you can choose to style only the last item by using the `last` method, like this:

```
$('ul>li').last().css('font-style', 'italic')
```

Or, to access an element by index (starting from 0), you can use the `eq` method, like this (which styles the second item in the list, because numbering starts at 0):

```
$('ul>li').eq(1).css('font-weight', 'bold')
```

You can also apply a filter to a selection using the `filter` method, like this (which changes the background color of every other element starting with the first, element 0):

```
$('ul>li').filter(':even').css('background', 'cyan')
```

Remember that when you are indexing into jQuery selections, the first element is the zeroth. So, for example, when you use the selector `:even` in this manner, elements 1, 3, 5, and so on will be selected (not 0, 2, 4, 6, etc.).

To exclude one or more elements, you can apply the `not` method, like this (where elements that *don't* use the ID `new` are styled in blue):

```
$('ul>li').not('#new').css('color', 'blue')
```

You can also select an element depending on what descendants it has. To select only elements that have descendant `<ol>` elements, for example, you could use this statement to place a line through those that match:

```
$('ul>li').has('ol').css('text-decoration', 'line-through')
```

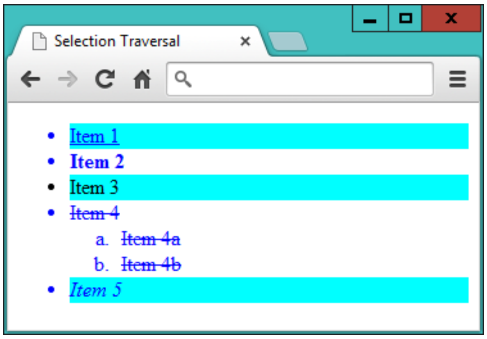
Example 25-23 brings all these together to style an unordered list, one of whose elements also contains an ordered list.

Example 25-23. Traversing a jQuery selection

```
<!DOCTYPE html>
<html>
  <head>
```

```
    <title>Selection Traversal</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
      <li id='new'>Item 3</li>
      <li>Item 4
        <ol type='a'>
          <li>Item 4a</li>
          <li>Item 4b</li>
        </ol></li>
      <li>Item 5</li>
    </ul>
    <script>
      $('ul>li').first()        .css('text-decoration', 'underline')
      $('ul>li').last()         .css('font-style',      'italic')
      $('ul>li').eq(1)          .css('font-weight',     'bold')
      $('ul>li').filter(':even').css('background',      'cyan')
      $('ul>li').not('#new')    .css('color',           'blue')
      $('ul>li').has('ol')      .css('text-decoration', 'line-through')
    </script>
  </body>
</html>
```

As you will see from studying Figure 25-21, every element in each list has been styled by one or more of the jQuery statements.



- Item 1
- **Item 2**
- Item 3
- ~~Item 4~~
    a. ~~Item 4a~~
    b. ~~Item 4b~~
- *Item 5*

Figure 25-21. Uniquely addressing elements in a jQuery selection

The is Method

There is also a jQuery selector that returns a Boolean value for use in plain JavaScript: the `is` method. Unlike the jQuery filtering methods shown in earlier sections, this function doesn't create a new jQuery object that can then have other methods appended to it, or that can then be further filtered. Instead, it returns just `true` or `false`, making the method most suitable for use in conditional statements.

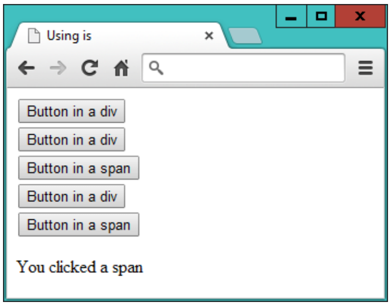
Example 25-24 uses the `is` method attached to a call to `parent` in an event handler for a set of buttons. When any button is clicked, the handler is called, and the `is` method returns a value of `true` or `false` when asked whether the parent element is a `<div>` (Figure 25-22).

Example 25-24. Reporting the `parent` element with `is`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Using is</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <div><button>Button in a div</button></div>
    <div><button>Button in a div</button></div>
    <span><button>Button in a span</button></span>
    <div><button>Button in a div</button></div>
    <span><button>Button in a span</button></span>
    <p id='info'></p>
    <script>
      $('button').click(function()
      {
        var elem = ''

        if ($(this).parent().is('div')) elem = 'div'
        else                            elem = 'span'

        $('#info').html('You clicked a ' + elem)
      })
    </script>
  </body>
</html>
```



You clicked a span

Figure 25-22. Using the `is` method to report the `parent` element

Using jQuery Without Selectors

A couple of jQuery methods are provided for use with standard JavaScript objects, rendering their handling a lot simpler. These are `$.each` and `$.map`, which are similar but have subtle differences.

The $.each Method

Using `$.each`, you can iterate through arrays or array-like objects by simply attaching a function to be called for each iteration. Example 25-25 shows an array of pet names and types (called `pets`), from which another array (called `guineapigs`) needs to be extracted, containing only the names of the guinea pigs.

Example 25-25. Calling the `$.each` method

```
<!DOCTYPE html>
<html>
  <head>
    <title>Using each</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body>
    <div id='info'></div>
    <script>
      pets =
      {
        Scratchy : 'Guinea Pig',
        Squeaky  : 'Guinea Pig',
        Fluffy   : 'Rabbit',
        Thumper  : 'Rabbit',
        Snoopy   : 'Dog',
        Tiddles  : 'Cat'
      }

      guineapigs = []

      $.each(pets, function(name, type)
      {
        if (type == 'Guinea Pig') guineapigs.push(name)
      })

      $('#info').html('The guinea pig names are: ' + guineapigs.join(' & '))
    </script>
  </body>
</html>
```

To do this, the `$.each` method is passed the array, along with an anonymous function to process it. The function takes two arguments, the index into the array (called `name`) and the contents of each element (called `type`).

The value in `type` is then tested to see whether it is `Guinea Pig`, and if so, the value in `name` is pushed onto the `guineapigs` array. Upon completion, the code displays the contents of the `guineapigs` array by writing them into the `<div>` element with the ID of `info`. To separate the items in the array, the JavaScript `join` method is used with the `&` symbol as a separator. The result of loading this example into a browser is simply to display the text "The guinea pig names are: Scratchy & Squeaky."

The $.map Method

Another way to achieve this is with the `$.map` method, which returns all the values that your function returns in an array.

This function saves you the trouble of creating an array, as we had to in the preceding example. Instead, you can create and populate the array at the same time, by assigning the array returned by `$.map` to a variable, like this (the end result being the same but with less code):

```
guineapigs = $.map(pets, function(type, name)
{
  if (type == 'Guinea Pig') return name
})
```

Watch out when you change between using the `$.each` and `$.map` methods, because `$.each` passes arguments to the function in the order *index*, *value*, but map uses the order *value*, *index*. This is why the two arguments are swapped in the preceding `$.map` example.

Using Asynchronous Communication

In Chapter 18, I showed in detail how to implement asynchronous communications between JavaScript in a browser and PHP running on a web server. I also provided some handy and compact functions you can call to simplify the process.

But if you have jQuery loaded, you can use its asynchronous functionality instead, if you prefer—it works in a very similar way, in that you choose whether to make a POST or a GET request and then take it from there.

Using the POST Method

Example 25-26 (which loads the Amazon Mobile website into a `<div>` element) is the direct jQuery equivalent to Example 18-1, but since all the asynchronous communication–handling code is tidied away in the jQuery library file, it's much shorter. It requires just a single call to the `$.post` method, passing it the following three items:

- The URL of a PHP program on the server to access
- The data to pass to that URL
- An anonymous function to process the returned data

Example 25-26. Sending a POST asynchronous request

```
<!DOCTYPE html>
<html> <!-- jqueryasyncpost.htm -->
  <head>
    <title>jQuery Asynchronous Post</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body style='text-align:center'>
    <h1>Loading a web page into a DIV</h1>
    <div id='info'>This sentence will be replaced</div>

    <script>
      $.post('urlpost.php', { url : 'amazon.com/gp/aw' }, function(data)
      {
        $('#info').html(data)
      } )
    </script>
  </body>
</html>
```

The *urlpost.php* program remains unchanged from Example 18-2, because this example and Example 18-1 are interchangeable.

Using the GET Method

Communicating asynchronously using the GET method is just as easy, requiring only the following two arguments:

- The URL of a PHP program on the server to access (including a query string containing the data to pass to it)
- An anonymous function to process the returned data

Example 25-27 is therefore the jQuery equivalent to Example 18-3.

Example 25-27. Sending a GET asynchronous request

```
<!DOCTYPE html>
<html> <!-- jqueryasyncget.htm -->
  <head>
    <title>jQuery Asynchronous GET</title>
    <script src='jquery-3.7.0.min.js'></script>
  </head>
  <body style='text-align:center'>
    <h1>Loading a web page into a DIV</h1>
    <div id='info'>This sentence will be replaced</div>

    <script>
      $.get('urlget.php?url=amazon.com/gp/aw', function(data)
      {
        $('#info').html(data)
      } )
    </script>
  </body>
</html>
```

The *urlget.php* program remains unchanged from Example 18-4, because this example and Example 18-3 are interchangeable.

Remember that the security restrictions of asynchronous communication require that the communication takes place with the same server that supplied the main web document. You must also use a web server for asynchronous communication, *not* a local filesystem. These examples are therefore best tested with a production or development server, as described in Chapter 2.

Plug-ins

There's room in this chapter to cover only the core jQuery library, and while that's more than enough for a beginner to accomplish a lot, the time will come when you'll find you need even more features and functionality. Thankfully, other jQuery projects can help you there, because a range of official and third-party plug-ins are now available to provide just about any features you can imagine.

jQuery User Interface

First, there's the jQuery User Interface plug-in, known as jQuery UI, which takes over directly where jQuery leaves off. With it, you can add dragging and dropping, resizing, and sorting methods to your web pages, as well as more animations and effects, animated color transitions, and more easing effects. It also provides a bunch of widgets to create menus and other features such as accordions, buttons, pickers, progress bars, sliders, spinners, tabs, tooltips, and much more.

If you want to see some demos before deciding whether to download it, check out the jQuery UI Demos page.

The whole package is under 400 KB zipped and is usable with almost no restrictions (just the very generous MIT license).

Other Plug-ins

The jQuery Plugin Registry brings together a wide variety of free, ready-made plug-ins to jQuery from numerous developers. These include plug-ins for form handling and verification, slideshows, responsive layout, image manipulation, additional animations, and much more.

If you are using jQuery and developing for mobile browsers, you will also want to take a look at jQuery Mobile (see Chapter 26), which offers sophisticated, touch-optimized ways to navigate the wide range of different types of mobile hardware and software to provide the best possible user experience.

You've come a long way in this chapter, learning material that sometimes takes up entire books. I hope you've found everything clear, though, because jQuery is very easy to learn and use. If you need any other information, check out the jQuery website.