Chapter 1

Getting Started with jQuery

The motto of jQuery is *write less, do more*. And by using the jQuery library, you can get started doing exactly that with nothing more than your favorite text editor.

This chapter starts with a *hello world*–like recipe to cover the basics, it discusses the fundamentals of the $ function and how to prevent conflicts with other libraries.

To ensure that your own code is just as concise as that of jQuery, the library offers a set of helper functions. This chapter covers the most important functions, which are designed to help you to eliminate useless boilerplate code or perform common tasks efficiently. You will see many of these functions being used in the following chapters.

Recipe: Introducing the Basic Usage of jQuery

jQuery helps you to clearly separate HTML and JavaScript. Instead of mixing the HTML with a large number of `onclick` attributes, jQuery selects all the elements that need to respond to a `click` event and binds a handler function. Listing 1.1 covers the basics of accessing HTML elements from JavaScript code and binding event handlers.

Listing 1.1 **Introducing the Basic Usage of jQuery**

```
00 <!DOCTYPE html>
01
02 <html lang="en">
03 <head>
04   <title>Introduction</title>
05   <style>
06   /*
07      CSS Style sheets are only inline to keep the
08      code examples together for readability. When
09      using this code in production, please
10      externalize all style sheets.
11   */
12   .myclass {
13     background-color: black;
```

(Continues)

Listing 1.1 **Introducing the Basic Usage of jQuery (Continued)**

```
14     color: white;
15   }
16   </style>
17 </head>
18 <body>
19
20 <h1>Introduction</h1>
21
22 <p>The HTML in the examples is kept as simple as
23   possible.</p>
24
25 <p class="myclass">This is a placeholder for content</p>
26
27 <p id="myid">You can click on this paragraph</p>
28
29 <script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>
30
31 <script>
32 // JavaScripts are only inline to keep the code
33 // examples together for readability. When using
34 // this code in production, please externalize
35 // all JavaScript code.
36
37 $(document).ready(function() {
38
39   $('p').css('font-weight', 'bold');
40
41   $('.myclass').html('Different <em>content</em>');
42
43   $('#myid').click(function() {
44     alert('Hello world!');
45   });
46
47 });
48 </script>
49 </body>
50 </html>
```

Caution

As a rule, you should avoid having any CSS and JavaScript within the HTML. This book breaks that rule for the sake of readability, in an effort to keep all the code of one example together. However, normally you should put your scripts in external JavaScript files and refer to them in the same way as line 29 refers to the jQuery library.

Except for the `script` element, the HTML is clean from JavaScript. The HTML contains a few pointers that make the elements easier to find for jQuery. These are the `class` and the `id`.

jQuery can be loaded from a Content Delivery Network (CDN) and included so as to always point to the latest version of jQuery. This can be done by loading the script from http://code.jquery.com/jquery-latest.min.js. You can find a list of CDN providers at http://docs.jquery.com/Downloading_jQuery. By always including the latest version of jQuery, you ensure that your site upgrades to the latest versions of jQuery automatically. For minor versions, this is advantageous because bugs and security risks are quickly eliminated. For major upgrades, you should ensure that you remove calls to deprecated APIs before a new version removes them. Some developers prefer to defer upgrading to new major versions of software until a first bug fix update is released.

Notice that scripts are loaded at the bottom of the page. This allows the browser to render all HTML before loading the script elements. The scripts would still work if they were placed on top of the page; however, the perceived loading time would be longer. This is generally due to request blocking, wherein the browser is unable to fetch more than a few files at a time, and thus it will actually stop page rendering until the download of the requested files is complete.

Even though the code is positioned at the bottom of the page, it is a good practice to bind the basic JavaScript calls to the `ready` event of the `document`, as you can see in line 37. This way, you can ensure that the page is done rendering before executing the code.

The Introduction stated that jQuery is used to separate HTML and JavaScript in a friendly way. Without jQuery, it would take more code to achieve the same result. HTML that is free of JavaScript also loads faster. If the HTML contains many `onclick` attributes, each time the browser reads an `onclick`, it will pause rendering and interpret the JavaScript. If your web application grows, rendering all of the HTML at once—without interruption and binding the events later—is faster.

The HTML adopts some parts of HTML5 but does not yet use all of the new element names to avoid compatibility problems on older browsers. For that same reason, this book uses HTML rather than XHTML.

Line 39 works on all paragraph elements in the document and changes a CSS property. Changing CSS properties this way is not recommended. This book uses the `css()` function mostly to indicate which elements are selected. Line 41 selects a paragraph of class `myclass` and changes its HTML by means of the `.html()` function. This allows you to either get or change the contents of the selector. Finally, line 43 selects a paragraph with id `myid` and binds a click handler that displays an alert box with *Hello World*.

The selectors in lines 39, 41, and 43 are CSS selectors. jQuery supports the majority of CSS3 selectors and adds its own extensions. The simpler and more standard the selections are, the faster they can perform. Selectors are covered in greater detail in Chapter 2, "Selecting Elements."

Recipe: Using jQuery with Other Libraries

jQuery's use of the `$` function name might seem odd if you are used to other programming languages. In many languages, `$` is reserved and cannot be used as a variable name on its own. As a result, it seems like the `$` is part of a language rather than a library.

JavaScript allows `$` as a variable name or as a function name, or as a variable name that points to a function, or to an object. As a result, many JavaScript libraries use the `$` as a shorthand notation for important functions.

jQuery uses the `$` as an alias for `jQuery`. This means that you can use the longer variable name `jQuery` when you call jQuery functions. Listing 1.2 demonstrates how jQuery helps you to avoid naming clashes with other libraries.

Listing 1.2 **Assigning the $ to a Different Function**

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>$.noConflict()</title>
</head>
<body>

<p>This example shows how to mix jQuery with non-jQuery code
   that uses the $ as a function name. Click on this paragraph
   to test it.</p>

<script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>

<script>
// please externalize this code to an external .js file
$(document).ready(function() {

  $.noConflict();

  $ = function(name) {
    alert('Hello, ' + name);
  };

  var clickHandler = function() {
    $('Reader');
  };

  (function($) {
    $('p').click(clickHandler);
  })(jQuery);
```

Listing 1.2 **Assigning the $ to a Different Function (Continued)**

```
31
32 });
33 </script>
34 </body>
33 </html>
```

After line 18, the `$` is no longer bound to jQuery and can be used for different purposes again. Lines 20–26 show how you can use the `$` for your own functions if you insist. It might as well be a different library that uses the `$`, instead.

Even after calling `noConflict`, there is a clean way to access jQuery code with the `$` without naming clashes. Lines 28–30 demonstrate how to pass the jQuery object to a function, ensuring that the `$` only is in scope inside that function. This function is executed directly after it is loaded.

Recipe: Determining the jQuery Version

You can ask jQuery to return its current version. This could be useful when you are upgrading and have multiple versions of jQuery running at the same time, and you want to ensure that you are working with the correct version. Listing 1.3 shows a different use of the `jquery` property. In case you are in doubt as to whether you are dealing with a jQuery object or a different kind of object, you can use the `jquery` property to help you to determine this.

Listing 1.3 **Testing Whether an Object is a jQuery Object**

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>jQuery Version</title>
</head>
<body>

<p>The jQuery version is: <span id="placeholder"></span></p>


<script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>

<script>
// please externalize this code to an external .js file
$(document).ready(function() {
```

(Continues)

Listing 1.3 **Testing Whether an Object is a jQuery Object (Continued)**

```
16
17   var a = {b: 1, c: 2};
18   var b = $('#placeholder');
19   var jqVersion = $.fn.jquery;
20
21   if(a.jquery) {
22     a.html(jqVersion + (' (a)'));
23   }
24
25   if(b.jquery) {
26     b.html(jqVersion + (' (b)'));
27   }
28
29 });
30 </script>
31 </body>
32 </html>
```

The `a` variable on line 17 is clearly not a jQuery object; the `b` variable on line 18 clearly is one. Line 19 retrieves the jQuery version, regardless of any variable. The code will never execute line 22. Instead, line 26 displays the jQuery version and confirms that `b` is a jQuery object, as expected.

Recipe: Iterating Arrays with each()

One of the reasons why you can do more with less code with jQuery is because JavaScript can be used as a functional language. Listing 1.4 demonstrates how you can use the `each()` function instead of creating a `for` loop.

Listing 1.4 **Numbering Each Value in a List**

```
00 <!DOCTYPE html>
01
02 <html lang="en">
03 <head>
04   <title>The each() function</title>
05 </head>
06 <body>
07
08 <h2>This example demonstrates the each() function by
09    prepending a letter before each paragraph below</h2>
10
11 <p>First</p>
12 <p>Second</p>
13 <p>Third</p>
```

Listing 1.4 **Numbering Each Value in a List (Continued)**

```
14
15 <script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>
16
17 <script>
18 // please externalize this code to an external .js file
19 $(document).ready(function() {
20
21   var values = ['a', 'b', 'c', 'd'];
22
23   $.each(values, function(index, value) {
24     values[index] = value.toUpperCase();
25   });
26
27   $('p').each(function(index, el) {
28     $(el).prepend(' - ');
29   });
30
31   $('p').each(function(index) {
32     $(this).prepend(values[index]);
33   });
34
35 });
36 </script>
37 </body>
38 </html>
```

On lines 21–25, the `each()` function is used to iterate over a regular array. It calls a function that changes all characters within the array to uppercase.

Lines 27–29 use the `each()` function to iterate over a set of elements selected by jQuery. Keep in mind that jQuery offers many shorthand notations for its own functions that work without the `each` function. It is preferable to avoid `each()` when jQuery provides a better method.

For readability, you might prefer `each()` over a `for` loop. For performance, a `for` loop might be faster in some cases. This is a matter of preference. This book favors `each()` over `for` loops.

Recipe: Manipulating Arrays by Using map()

Although `each()` seems cleaner than a `for` loop, there is a cleaner way to manipulate arrays. Listing 1.5 illustrates how to use `map()` for a similar purpose as Listing 1.4. As it turns out, `each()` is more appropriate for making function calls based on array elements. The `map()` function is specifically meant to change array elements by processing them into a new array. In `map()`, the variable assignment is replaced by a simple `return`.

Listing 1.5 **Modifying All Elements in an Array**

```
00 <!DOCTYPE html>
01
02 <html lang="en">
03 <head>
04   <title>The map() function</title>
05 </head>
06 <body>
07
08 <h2>This example shows the basic usage of a map.</h2>
09
10 <p id="before">Before map(): </p>
11
12 <p id="after">After map(): </p>
13
14 <script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>
15 <script>
16 // please externalize this code to an external .js file
17
18 $(document).ready(function() {
19
20   var arr = [1, 2, 3, 4, 5];
21
22   $('#before').append(JSON.stringify(arr));
23
24   arr = $.map(arr, function(value, index) {
25     return 'done something with ' + value;
26   });
27
28   $('#after').append(JSON.stringify(arr));
29
30 });
31 </script>
32 </body>
33 </html>
```

The change that line 25 makes to the array elements might not be useful in practice; however, it shows that you can easily change an array of integers into an array of strings. JavaScript is weakly typed, after all.

Recipe: Working with Arrays of Elements

So far, you have seen arrays of integers, arrays of strings, and jQuery objects implicitly containing a list of HTML elements. Listing 1.6 displays how you can access the HTML elements within the jQuery object and transform the list into a regular jQuery-free array.

Listing 1.6 **Retrieving Arrays and Elements in Multiple Ways**

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>The get() function and alternatives</title>
</head>
<body>

<h2>Various ways to get an array or a specific element</h2>

<p>First</p>
<p>Second</p>
<p>Third</p>

<button id="get">Get</button><br>
<button id="to-array">To array</button><br>
<button id="make-array">Make array</button><br>
<button id="first">First</button><br>
<button id="get-first">Get first</button><br>
<button id="get-last">Get last</button>

<script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>

<script>
// please externalize this code to an external .js file
$(document).ready(function() {

  // same as next
  $('#get').click(function() {
    alert('Get p = ' + $('p').get());
  });

  // same as previous, better readability
  $('#to-array').click(function() {
    alert('Get p = ' + $('p').toArray());
  });

  // yet another alternative with the same functionality
  $('#make-array').click(function() {
    alert('Get p = ' + $.makeArray($('p')));
  });

  // same as next
  $('#first').click(function() {
```

(Continues)

Listing 1.6 **Retrieving Arrays and Elements in Multiple Ways (Continued)**

```
44     alert('First p = ' + $('p')[0].innerHTML);
45   });
46   // same as previous
47   $('#get-first').click(function() {
48     alert('Get first p = ' + $('p').get(0).innerHTML);
49   });
50
51   // ease of use: get the last element
52   $('#get-last').click(function() {
53     alert('Get last p = ' + $('p').get(-1).innerHTML);
54   });
55
56 });
57 </script>
58 </body>
59 </html>
```

Lines 29, 34, and 39 perform the same function: transform a jQuery object containing a selection into an array of HTML elements. The `get()` method works well but is not the most readable one for this purpose. The `getArray()` and `makeArray()` functions are mostly different in the way they are called and can both be used, depending on preference.

Using `get()` to obtain specific elements from the current selection is similar to accessing the jQuery object by using brackets `[]`. One advantage is the possibility to get the last element by asking for position `-1`.

Recipe: Getting the Position of an Element by Using index()

The last recipe showed how to convert jQuery selections into arrays of regular HTML elements. All HTML elements have a position, not only within the jQuery selection, but in the complete HTML document. Listing 1.7 displays how to ascertain this position.

Listing 1.7 **Determining the Index of Paragraphs**

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>The index() function</title>
</head>
<body>

```

Listing 1.7 **Determining the Index of Paragraphs (Continued)**

```
<h1>Click on the paragraphs to see their index</h1>

<p>First</p>
<span>Not a paragraph so it does not respond to your clicks</span>
<p>Second</p>
<p>Third</p>
<p>Some nested paragraphs:
  <p>Nested 1</p>
  <p>Nested 2</p>
</p>
<p>And a final non-nested paragraph</p>

<script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>

<script>
// please externalize this code to an external .js file
$(document).ready(function() {

  $('p').click(function() {
    alert('Index of clicked item is: ' + $(this).index())
  });

});
</script>
</body>
</html>
```

By selecting only paragraph elements from a document that mixes multiple element types, it becomes apparent that the returned value of the `index()` function is independent of the jQuery selection.

Recipe: Finding Elements in an Array by Using grep()

To find an element inside an array, you can use a `grep()`function (like the Unix command to find text in files), which shares similarities with `map()` and `each()`. Listing 1.8 shows how to use `grep()` to select the months that have the character "r" in their name.

Listing 1.8 **Selecting the Months in a Year That Have an "r" in Their Name**

```
<!DOCTYPE html>

<html lang="en">
<head>
```

(Continues)

Listing 1.8 **Selecting the Months in a Year That Have an "r" in Their Name (Continued)**

```
  <title>The grep() function</title>
</head>
<body>

<h2>All months</h2>
<p id="all-months"></p>

<h2>Months when you should take extra vitamin</h2>
<p id="vitamin"></p>
<script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>

<script>
// please externalize this code to an external .js file
$(document).ready(function() {

  var arr = ['January', 'February', 'March', 'April', 'May',
   'June', 'July', 'August', 'September', 'October',
     'November', 'December'];

  var rInMonth = $.grep(arr, function(value, index) {
   return value.indexOf('r') >= 0;
  });

  $('#all-months').html(arr.join('<br> '));
  $('#vitamin').html(rInMonth.join('<br> '));

});
</script>
</body>
</html>
```

Where `each()` is useful for calling other functions for each array element and `map()` is useful for changing all values in an array, `grep()` is useful for selecting a subset of an array. All function calls expect either `true` or `false`, depending on whether the current element should be in the result set.

Recipe: Determining the Size of an Element Set by Using length()

In case you need to know how many items are selected by jQuery, the `length()` function comes in handy. Also note that it will return the same value as the `size()` function. Listing 1.9 shows how to use this function.

Listing 1.9 **Determining the Number of Paragraphs**

```
00 <!DOCTYPE html>
01
02 <html lang="en">
03 <head>
04   <title>The length() function</title>
05 </head>
06 <body>
07
08 <h2>By clicking on the button below, you can see
09   the length of the selected element set.</p>
10
11 <p>First</p>
12 <p>Second</p>
13 <p>Third</p>
14
15 <button id="get-length">Get length</button>
16
17 <script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>
18
19 <script>
20 // please externalize this code to an external .js file
21 $(document).ready(function() {
22
23   // same as next
24   $('#get-length').click(function() {
25     alert('Length = ' + $('p').length);
26   });
27
28 });
29 </script>
30 </body>
31 </html>
```

The use of `length()` is straightforward. Line 25 returns the number of paragraph elements selected.

Recipe: Retrieving HTML5 data- Attributes

With HTML5, you can add your own attributes to HTML elements when they start with `data-`. This can be useful for web applications that need to transfer many small pieces of data connected to the DOM tree that should remain hidden to the web site visitor. Listing 1.10 demonstrates how jQuery helps you to read the contents of these `data-` attributes.

Listing 1.10 **Reading Hidden Text from data-myattribute**

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>Using data() for HTML5 data- attributes</title>
</head>
<body>

<p data-myattribute="just some random content"
id="test-data">If you press the button, you can reveal
the text hidden as an attribute inside the paragraph
element as an HTML5 data attribute.</p>

<button>Get data attribute</button>

<script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>

<script>
// please externalize this code to an external .js file
$(document).ready(function() {

  $('button').click(function() {
    alert('The data is: ' + $('#test-data').data('myattribute'));
  });

});
</script>
</body>
</html>
```

If your attribute is called `data-myattribute`, you can fetch the data with the `data()` function, specifying `myattribute` as an argument. In this case, you can consider `data()` as a convenience function. The `data()` function has more uses as you can see in the next recipe.

Recipe: Storing Element Data by Using data()

Manipulating the HTML document is relatively slow. Some web applications abuse the HTML document to store hidden data that is associated with specific HTML elements. An application might use attributes like `data-myattribute` to achieve that.

Previously, storing data inside the HTML document was considered a bad practice. This was due to web developers haphazardly jamming in custom elements that would cause validation errors and could cause potential problems with the page rendering. HTML5 has added support for `data-*` attributes and jQuery provides a means in the form of the `data()` function to use the data stored in them. In the last recipe, you saw how this function helps you fetch `data-` attributes. That is not the main function. Listing 1.11 demonstrates how to use `data()` to store element specific data in a central storage, outside the document.

Listing 1.11 **Storing Element Data without Affecting Other Elements**

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>The data() function: storing</title>
</head>
<body>

<button id="store">Store some data in the paragraph
   below</button><br>

<button id="show-data">Show the data in the paragraph
  below</button>

<p id="store-data">Even if you look in Firebug, you
  cannot see any data that is stored related to this
  element after storing it.</p>

<button id="show-empty">Show that the next paragraph
  does not contain the data</button>

<p id="empty-data">To show you that the data belongs
  to the other paragraph, this paragraph is intentionally
  left without any data.</p>

<script src="http://code.jquery.com/jquery-latest.min.js"></script>

<script>
// please externalize this code to an external .js file
$(document).ready(function() {


  $('#store').click(function() {
     $('#store-data').data('myattribute', 'some data');
  });

  $('#show-data').click(function() {
   alert('The data is: ' + $('#store-data').data('myattribute'));
  });
```

(Continues)

Listing 1.11 **Storing Element Data without Affecting Other Elements (Continued)**

```

  $('#show-empty').click(function() {
    alert('The data is: ' + $('#empty-data').data('myattribute'));
  });
});
</script>
</body>
</html>
```

Open the example in your browser and use developer tools to see what happens with the generated HTML tree; or better, what does not happen to it. The data is stored outside the document. Nevertheless, the data is still associated with specific elements.

Recipe: Removing Element Data by Using removeData()

If your application processes a lot of data, it is wise to think about memory usage. When you no longer need data associated with certain elements, you should remove it. Listing 1.12 shows the use of `removeData()` for this purpose.

Listing 1.12 **Demonstrating Data Removal and the Storage of Objects**

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>The data() function: removing</title>
</head>
<body>

<button id="store-data">Store data</button><br>

<button id="show-data">Show the data</button><br>

<button id="remove-data">Remove the data</button>

<p id="store-data">This paragraph element is the placeholder
   for the data to be stored. Showing the data will reveal
   a little bit of the jQuery internals for storing data.
   You can ignore the technical details.</p>


<script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>

```

Listing 1.12 **Demonstrating Data Removal and the Storage of Objects (Continued)**

```
22 <script>
23 // please externalize this code to an external .js file
24 $(document).ready(function() {
25
26
27   $('#store-data').click(function() {
28     $('#store-data').data('myattribute',
29       {a: 'first', b: 'second'});
30   });
31
32   $('#show-data').click(function() {
33     alert('The data is: ' +
34       JSON.stringify($('#store-data').data()));
35   });
36
37   $('#remove-data').click(function() {
38      $('#store-data').removeData();
39   });
40 });
41 </script>
42 </body>
43 </html>
```

Line 28 shows the `data()` function being used to store data. Line 29 shows the data that will be saved. This data is in JavaScript Object Notation (JSON) format. Using JSON, you can store data in key-value pairs. You can learn more about JSON by visiting www.json.org/. Line 38 removes all data associated to the button with id `show-data`. If you only want to remove `myattribute`, you can pass this as an argument to the `removeData()` function. This code example contains more than just the `removeData()`. Line 29 and 34 demonstrate that you can store complete objects by using the `data()` function and remove them with one call.

Recipe: Testing and Manipulating Variables

JavaScript is a weakly typed language. This means you can never be 100 percent sure what kind of data is stored in variables, especially not if multiple developers are involved or when multiple libraries are manipulating data.

Listing 1.13 demonstrates jQuery's helper functions for testing data types.

Listing 1.13 **Showing Variable Types and Modifying Arrays**

```
00 <!DOCTYPE html>
01
02 <html lang="en">
```

(Continues)

Listing 1.13 **Showing Variable Types and Modifying Arrays (Continued)**

```
<head>
  <title>Various variable testing functions</title>
</head>
<body>

<p>This is a list of variable type tests, followed by
  a few modification functions:</p>
<p id="placeholder"></p>

<script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>

<script>
// please externalize this code to an external .js file
$(document).ready(function() {

  var testObj = {},
      testVar = 1,
      testFun = function() {},
      testArr = [1, 2, 2];

  var results = [];

  results.push('testObj = ' + JSON.stringify(testObj));
  results.push('testVar = ' + JSON.stringify(testVar));
  results.push('testFun = ' + JSON.stringify(testFun));
  results.push('testArr = ' + JSON.stringify(testArr));

  results.push('<br>');

  results.push('type(testObj) = ' + $.type(testObj));
  results.push('type(testVar) = ' + $.type(testVar));
  results.push('type(testFun) = ' + $.type(testFun));
  results.push('type(testArr) = ' + $.type(testArr));

  results.push('<br>');

  results.push('inArray(3, testArr) = '
                + $.inArray(3, testArr));

  results.push('isArray(testArr) = '
               + $.isArray(testArr));

  results.push('isEmptyObject(testObj) = '
               + $.isEmptyObject(testObj));

```

Listing 1.13 **Showing Variable Types and Modifying Arrays (Continued)**

```
48   results.push('isPlainObject(testObj) = '
49               + $.isPlainObject(testObj));
50
51   results.push('isFunction(testFun) = '
52               + $.isFunction(testFun));
53
54   results.push('<br>');
55
56   results.push('merge(testArr, [3, 3, 4]) = '
57               + $.merge(testArr, [3, 3, 4]));
58
59   results.push('unique(testArr) = '
60               + $.unique(testArr));
61
62   results.push('merge(testArr, [5, 6]) = '
63               + $.merge(testArr, [5, 6]));
64
65   $('#placeholder').append(results.join('<br>'));
66
67 });
68 </script>
69 </body>
70 </html>
```

Although this code won't win any beauty contests, it demonstrates many different functions that are available for testing data types. Lines 32–35 show how the `type()` function of jQuery works. This function checks for the existence of an internal JavaScript [[Class]]. When `type()` finds one, it displays it. The main difference between the jQuery `type()` and the standard JavaScript `typeof()` function is in the return. The `typeof()` function returns an "object," whereas the `type()`returns an "array".

Lines 39–43 deal with array handling. The `inArray()` function searches through an array for a specific value and returns either the index of the value in the array or a -1. The `isArray()` function performs what it is name suggests; it checks to see if the object it is testing is an array.

Lines 45–52 demonstrate the use of object detection. The `isEmptyObject()` function checks an object for the presence of a value or data content, whereas the `isPlainObject()` is used to determine if the object being tested was created with either `{}` or `new Object`. Lines 51 and 52 show the use of the `isFunction()` function, which as you can probably surmise checks the argument passed to see if it is a function object. Something to keep in mind when using `isFunction()` is that the official jQuery documentation states that after version 1.3 of jQuery, checking browser functions such as `alert()` might not work correctly in some browsers (such as Internet Explorer).

Lines 56–63 demonstrate how to merge arrays and filter the unique elements out of it. When you run this code, you discover that these functions have side effects. They change the array that is passed as an argument.

Recipe: Extending Objects by Using extend()

When you are working with objects, sometimes you want to merge two objects into one or extend one object with functions and properties of the other. Listing 1.14 displays how jQuery's `extend()` function helps to combine two objects.

Listing 1.14 **Displaying the Side Effects of Extending Objects**

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>The extend() function</title>
</head>
<body>

<p>First object:</p>

<p id="object-1"></p>

<p>Second object:</p>

<p id="object-2"></p>

<p>Result object:</p>

<p id="object-result"></p>

<button id="extend">Extend</button>

<button id="extend-new">Extend into empty</button>

<button id="reset">Reset</button>

<script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>

<script>
// please externalize this code to an external .js file
$(document).ready(function() {

  var reset,
        object1 = object2 = objectresult = {};

```

Listing 1.14 **Displaying the Side Effects of Extending Objects (Continued)**

```
35   var show = function() {
36     $('#object-1').html(JSON.stringify(object1));
37     $('#object-2').html(JSON.stringify(object2));
38     $('#object-result').html(JSON.stringify(objectresult));
39   };
40
41   (reset = function() {
42     object1 = {
43       a: 'original a',
44       b: 'original b',
45       c: 'original c'
46     };
47     object2 = {
48       c: 'different c',
49       d: 'different d'
50     };
51     objectresult = {};
52     show();
53    })();
54
55    $('#extend').click(function() {
56      // has side effects for object1
57      objectresult = $.extend(object1, object2);
58      show();
59    });
60
61    $('#extend-new').click(function() {
62      // without side effects for object1
63      objectresult = $.extend({}, object1, object2);
64      show();
65    });
66
67    $('#reset').click(function() {
68      reset();
69    });
70
71 });
72 </script>
73 </body>
74 </html>
```

Compare lines 57 and 63. Line 57 merges the second object into the first object, overwriting functions and properties that already existed in the first object. Moreover, the first object is changed by this function.

If you want to have an extended version of `object1` without affecting `object1` itself, line 63 demonstrates how to merge both objects into a new object. The resulting new object has the same functions and properties as the result from line 57. The difference is that now you have three objects instead of two, and `object1` itself is unchanged.

Recipe: Serializing the Data in a Form

If you need the current data of a form before submitting it, you can select all form elements, iterate over them and read the values. There are shorter ways to achieve the same result. Listing 1.15 shows two ways to get the current input that was entered into a form with a single function call.

Listing 1.15 **Serializing the Current Form Input into Two Different Formats**

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>The serialize() and serializeArray() function</title>
</head>
<body>


<h2>Press the buttons to see the result of two
  different serialization functions</h2>

<form action="" method="post">
  <label for="first_field">First field</label>
  <input type="text" name="first_field"
    value="" id="first_field"><br>
  <label for="second_field">Second field</label>
  <input type="text" name="second_field"
    value="" id="second_field"><br>
  <label for="third_field">Third field</label>
  <input type="text" name="third_field"
    value="" id="third_field"><br>
  <label for="fourth_field">Fourth field</label>
  <input type="text" name="fourth_field"
    value="" id="fourth_field"><br>
</form>

<input type="button" name="serialize"
  value="Serialize" id="serialize">
<input type="button" name="serialize-array"
  value="SerializeArray" id="serialize-array">

<div id="placeholder"></div>
```

Listing 1.15 **Serializing the Current Form Input into Two Different Formats (Continued)**

```
33
34 <script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>
35
36 <script>
37 // please externalize this code to an external .js file
38 $(document).ready(function() {
39
40   $('#serialize').click(function() {
41     $('#placeholder').html($('form').serialize());
42   });
43
44   $('#serialize-array').click(function() {
45     $('#placeholder').html(JSON.stringify(
46       $('form').serializeArray()));
47   });
48
49 });
50 </script>
51 </body>
52 </html>
```

Line 41 shows the `serialize()` method to retrieve form content. The result is an escaped query string that could be directly used in an HTTP call to the server.

Line 46 uses `serializeArray()` for the same purpose. This function is an internal helper that is called during `serialize()`. It returns an object with keys and values. If you want to transform the form content into a different object, the result from `serializeArray()` might be easier to use.

Recipe: Testing Browsers for Feature Support

One of the goals of jQuery is to help you to handle browser incompatibilities. Many small differences between browser vendors and browser versions are covered by jQuery itself. Still, some differences remain with which jQuery cannot help you. Listing 1.16 shows how jQuery provides properties that indicate whether the current browser supports certain features and characteristics.

Listing 1.16 **Listing All Support Testing Properties**

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>The support property</title>
</head>
```

(Continues)

Listing 1.16 **Listing All Support Testing Properties (Continued)**

```
<body>

<p>The paragraph under this contains the supported
  properties of this browser:</p>

<p id="placeholder"></p>

<script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>

<script>
// please externalize this code to an external .js file
$(document).ready(function() {

var result = [];
 $.each(
 ('ajax boxModel changeBubbles checkClone checkOn cors cssFloat ' +
 'hrefNormalized htmlSerialize leadingWhitespace noCloneChecked ' +
 'noCloneEvent opacity optDisabled optSelected style ' +
 'submitBubbles tbody').split(' '), function(index, name) {
  result.push(name + ' = ' + $.support[name] + '<br>');
});

  $('#placeholder').html(result.join('') );

});
</script>
</body>
</html>
```

The `ajax` property indicates the browser's support for calls to the server, for example. And `opacity` indicates whether you can create a see-through effect between multiple elements.

A complete reference of all these properties is beyond the scope of this book. For more information about them, go to http://api.jquery.com/jQuery.support/. jQuery also contains a `browser` property that should not be used. Although `support` is preferable over `browser`, it is still better to avoid using it as long as you can.

Summary

This chapter introduced the basics of using jQuery. It demonstrated how to use the `$` variable for other purposes than jQuery. After that, a long list of support functions was demonstrated. Most of these functions assist in manipulating variables, objects, arrays, and data elements. Using these functions can help you to keep your code concise.