jQuery

Succinctly

by Cody Lindley

Syncfusion | Technology Resource Portal

Chapter 1 Core jQuery

Base concept behind jQuery

While some conceptual variations exist (e.g. functions like **$.ajax**) in the jQuery API, the central concept behind jQuery is "find something, do something." More specifically, select DOM element(s) from an HTML document and then do something with them using jQuery methods. This is the big picture concept.

To drive this concept home, reflect upon the code below.

Sample: sample1.html

```
<!DOCTYPE html>
<html lang="en">

    <body>
        <!-- jQuery will change this -->
        <a href=""></a>
        <!-- to this <a href="http://www.jquery.com">jQuery.</a> -->
        <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"
></script>
        <script>
            jQuery('a').text('jQuery').attr('href', 'http://www.jquery.com');
        </script>
    </body>

</html>
```

Notice that in this HTML document we are using jQuery to select a DOM element (**<a>**). With something selected, we then do something with the selection by invoking the jQuery methods **text()**, **attr()**, and **appendTo()**.

The **text** method called on the wrapped **<a>** element and set the display text of the element to be “jQuery.” The **attr** call sets the **href** attribute to the jQuery Web site.

Grokking the "find something, do something" foundational concept is critical to advancing as a jQuery developer.

The concept, behind the concept, behind jQuery

While selecting something and doing something is the core concept behind jQuery, I would like to extend this concept to include creating something as well. Therefore, the concept behind jQuery could be extended to include first creating something new, selecting it, and then doing something with it. We could call this the concept, behind the concept, behind jQuery.

What I am trying to make obvious is that you are not stuck with only selecting something that is already in the DOM. It is additionally important to grok that jQuery can be used to create new DOM elements and then do something with these elements.

In the code example below, we create a new `<a>` element that is not in the DOM. Once created, it is selected and then manipulated.

Sample: sample2.html

```
<!DOCTYPE html>
<html lang="en">

    <body>
        <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"
></script>
        <script>
            jQuery('<a>jQuery</a>').attr('href', 'http://www.jquery.com').appendTo('b
ody');
        </script>
    </body>

</html>
```

The key concept to pick up here is that we are creating the `<a>` element on the fly and then operating on it as if it was already in the DOM.

jQuery requires HTML to run in standards mode or almost-standards mode

There are known issues with jQuery methods not working correctly when a browser renders an HTML page in quirks mode. Make sure when you are using jQuery that the browser interprets the HTML in standards mode or almost standards mode by using a valid doctype.

To ensure proper functionality, code examples in this book use the HTML 5 doctype.

```
<!DOCTYPE html>
```

Waiting on the DOM to be ready

jQuery fires a custom event named **ready** when the DOM is loaded and available for manipulation. Code that manipulates the DOM can run in a handler for this event. This is a common pattern seen with jQuery usage.

The following sample features three coded examples of this custom event in use.

Sample: sample3.html

```
<!DOCTYPE html>
<html lang="en">

    <head>
        <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"
></script>
        <script>
            // Standard.
            jQuery(document).ready(function () { alert('DOM is ready!'); });

            // Shortcut, but same thing as above.
            jQuery(function () { alert('No really, the DOM is ready!'); });

            // Shortcut with fail-safe usage of $. Keep in mind that a reference
            // to the jQuery function is passed into the anonymous function.
            jQuery(function ($) {
                    alert('Seriously its ready!');
                    // Use $() without fear of conflicts.
            });
        </script>
    </head>

    <body></body>

</html>
```

Keep in mind that you can attach as many **ready()** events to the document as you would like. You are not limited to only one. They are executed in the order they were added.

Executing jQuery code when the browser window is completely loaded

Typically, we do not want to wait for the **window.onload** event. That is the point of using a custom event like **ready()** that will execute code before the window loads, but after the DOM is ready to be traversed and manipulated.

However, sometimes we actually do want to wait. While the custom **ready()** event is great for executing code once the DOM is available, we can also use jQuery to execute code once the entire Web page (including all assets) is completely loaded.

This can be done by attaching a load event handler to the **window** object. jQuery provides the **load()** event method that can be used to invoke a function once the window is completely loaded. Below, I provide an example of the **load()** event method in use.

Sample: sample4.html

```
<!DOCTYPE html>
<html lang="en">

    <head>
        <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"
></script>
        <script>
            // Pass window to the jQuery function and attach
            // event handler using the load() method shortcut.
            jQuery(window).load(function(){     alert('The page and all its assets
are loaded!'); });
        </script>
    </head>

    <body></body>

</html>
```

Include all CSS files before including jQuery

As of jQuery 1.3, the library no longer guarantees that all CSS files are loaded before it fires the custom **ready()** event. Because of this change in jQuery 1.3, you should always include all CSS files before any jQuery code. This will ensure that the browser has parsed the CSS before it moves on to the JavaScript included later in the HTML document. Of course, images that are referenced via CSS may or may not be downloaded as the browser parses the JavaScript.

Using a hosted version of jQuery

When embedding jQuery into a Web page, most people choose to download the source code and link to it from a personal domain/host. However, there are other options that involve someone else hosting the jQuery code for you.

Google hosts several versions of the jQuery source code with the intent of it being used by anyone. This is actually very handy. In the code example below I am using a `<script>` element to include a minified version of jQuery that is hosted by Google.

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></scrip
t>
```

Google also hosts several previous versions of the jQuery source code, and for each version, minified and non-minified variants are provided. I recommend using the non-minified variant during development, as debugging errors is always easier when you are dealing with non-minified code.

A benefit of using a Google hosted version of jQuery is that it is reliable, fast, and potentially cached.

Executing jQuery code when DOM is parsed without using `ready()`

The custom `ready()` event is not entirely needed. If your JavaScript code does not affect the DOM, you can include it anywhere in the HTML document. This means you can avoid the `ready()` event altogether if your JavaScript code is not dependent on the DOM being intact.

Most JavaScript nowadays, especially jQuery code, will involve manipulating the DOM. This means the DOM has to be fully parsed by the browser in order for you to operate on it. This fact is why developers have been stuck on the `window.onload` roller coaster ride for a couple of years now.

To avoid using the `ready()` event for code that operates on the DOM, you can simply place your code in an HTML document before the closing `</body>` element. Doing so ensures the DOM is completely loaded, simply because the browser will parse the document from top to bottom. If you place your JavaScript code in the document after the DOM elements it manipulates, there is no need to use the `ready()` event.

In the example below, I have forgone the use of `ready()` by simply placing my script before the document body closes. This is the technique I use throughout this book and on the majority of sites I build.

Sample: sample5.html

```
<!DOCTYPE html>
<html lang="en">
<body>
    <p>
```

```
        Hi, I'm the DOM! Script away!</p>
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></s
cript>
    <script>        alert(jQuery('p').text()); </script>
</body>
</html>
```

If I were to place the **<script>** before the **<p>** element, it would execute before the browser had loaded the **<p>** element. This would cause jQuery to assume the document does not contain any **<p>** elements. However, if I were to use the jQuery custom **ready()** event, then jQuery would not execute the code until the DOM was ready. But why do this, when we have control over the location of the **<script>** element in the document? Placing jQuery code at the bottom of the page avoids having to using the **ready()** event. In fact, placing all JavaScript code at the bottom of a page is a proven performance strategy.

Grokking jQuery chaining

Once you have selected something using the jQuery function and created a wrapper set, you can actually chain jQuery methods to the DOM elements contained inside the set. Conceptually, jQuery methods continue the chain by always returning the jQuery wrapper set, which can then be used by the next jQuery method in the chain. Note: Most jQuery methods are chainable, but not all.

You should always attempt to reuse the wrapped set by leveraging chaining. In the code below, the **text()**, **attr()**, and **addClass()** methods are being chained.

Sample: sample6.html

```
<!DOCTYPE html>
<html lang="en">
<body>
    <a></a>
    <script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
    <script> (function ($) {
     $('a').text('jQuery') // Sets text to jQuery, and then returns $('a').
      .attr('href', 'http://www.jquery.com/') // Sets the href attribute and then returns
$('a').
      .addClass('jQuery'); // Sets class and then returns $('a').
 })(jQuery) </script>
</body>
</html>
```

Breaking the chain with destructive methods

As mentioned before, not all jQuery methods maintain the chain. Methods like **`text()`** can be chained when used to set the text node of an element. However, **`text()`** actually breaks the chain when used to get the text node contained within an element.

In the example below, **`text()`** is used to set and then get the text contained within the **`<p>`** element.

Sample: sample7.html

```
<!DOCTYPE html>
<html lang="en">
<body>
    <p></p>
    <script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
    <script> (function ($) {
    var theText = $('p').text('jQuery').text(); // Returns the string "jQuery".
     alert(theText);
     // Cannot chain after text(). The chain is broken.
     // A string is returned, not the jQuery object.
 })(jQuery) </script>
</body>
</html>
```

Getting the text contained within an element using **`text()`** is a prime example of a broken chain because the method will return a string containing the text node, but not the jQuery wrapper set.

It should be no surprise that if a jQuery method does not return the jQuery wrapper set, the chain is thereby broken. This method is considered to be destructive.

Using destructive jQuery methods and exiting destruction using `end()`

jQuery methods that alter the original jQuery wrapper set selected are considered to be destructive. The reason is that they do not maintain the original state of the wrapper set. Not to worry; nothing is really destroyed or removed. Rather, the former wrapper set is attached to a new set.

However, fun with chaining does not have to stop once the original wrapped set is altered. Using the **`end()`** method, you can back out of any destructive changes made to the original wrapper set. Examine the usage of the **`end()`** method in the following example to understand how to operate in and out of DOM elements.

Sample: sample8.html

```
<!DOCTYPE html>
<html lang="en">
<body>
    <style>
        .last
        {
            background: #900;
        }
    </style>
    <ul id="list">
        <li></li>
        <li>
            <ul>
                <li></li>
                <li></li>
                <li></li>
            </ul>
        </li>
        <li></li>
        <li></li>
    </ul>
    <script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
    <script>  (function ($) {
      $('#list') // Original wrapper set.
        .find('> li') // Destructive method.
            .filter(':last') // Destructive method.
                .addClass('last')
            .end() // End .filter(':last').
            .find('ul') // Destructive method.
                 .css('background', '#ccc')
                .find('li:last') // Destructive method.
                    .addClass('last')
                .end() // End .find('li:last')
            .end() // End .find('ul')
        .end() // End .find('> li')
        .find('li') // Back to the orginal $('#list')
            .append('I am an &lt;li&gt;');
  })(jQuery); </script>
</body>
</html>
```

Aspects of the jQuery function

The jQuery function is multifaceted. We can pass it differing values and string formations that it can then use to perform unique functions. Here are several uses of the jQuery function:

- Select elements from the DOM using CSS expressions and custom jQuery expressions, as well as selecting elements using DOM references: **`jQuery('p > a')`** or **`jQuery(':first')`** and **`jQuery(document.body)`**
- Create HTML on the fly by passing HTML string structures or DOM methods that create DOM elements: **`jQuery('<div id="nav"></div>')`** or **`jQuery(document.createElement('div')`**
- A shortcut for the **`ready()`** event by passing a function to the jQuery function: **`jQuery(function($){`** /* Shortcut for ready() */ **`})`**

Each of these usages is detailed in the code example below.

Sample: sample9.html

```
<!DOCTYPE html>
<html lang="en">
<body>
    <script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
    <script>  jQuery(function($){ // Pass jQuery a function.
      // Pass jQuery a string of HTML.
      $('<p></p>').appendTo('body');
      // Pass jQuery an element reference.
      $(document.createElement('a')).text('jQuery').appendTo('p');
      // Pass jQuery a CSS expression.
      $('a:first').attr('href', 'http://www.jquery.com');
      // Pass jQuery a DOM reference.
      $(document.anchors[0]).attr('jQuery');
  }); </script>
</body>
</html>
```

Grokking when the keyword `this` refers to DOM elements

When attaching events to DOM elements contained in a wrapper set, the keyword **`this`** can be used to refer to the current DOM element invoking the event. The following example contains jQuery code that will attach a custom **`mouseenter`** event to each **`<a>`** element in the page. The native JavaScript **`mouseover`** event fires when the cursor enters or exits a child element, whereas jQuery's **`mouseenter`** does not.

Sample: sample10.html

```
<!DOCTYPE html>
<html lang="en">
<body>
    <a id="link1">jQuery.com</a>
    <a id="link2">jQuery.com</a>
    <a id="link3">jQuery.com</a>
```