

# jQuery Fundamentals Training

jQuery Core

## Lesson 1, Activity 2: `$` vs `jQuery`

The jQuery script defines a global object named `jQuery`, containing all the elements we have used so far. It then assigns that to another variable, `$`. So, `$` and `jQuery` are synonymous. jQuery does cache the original value of `$`. Invoking the `jQuery.noConflict()` method, discussed in the next activity, reassigns the original value of `$` from the cached value.

## Lesson 1, Activity 3: \$ vs \$()

Until now, we've been dealing entirely with methods that are called on a jQuery object. For example:

```
$('h1').remove();
```

Most jQuery methods are called on jQuery objects as shown above; these methods are said to be part of the `$.fn` namespace, or the "jQuery prototype", and are best thought of as jQuery object methods.

However, there are several methods that do not act on a selection; these methods are said to be part of the jQuery namespace, and are best thought of as core jQuery methods.

This distinction can be incredibly confusing to new jQuery users. Here's what you need to remember:

- Methods called on jQuery selections are in the `$.fn` namespace, and automatically receive and return the selection as `this`.
- Methods in the `$` namespace are generally utility-type methods, and do not work with selections; they are not automatically passed any arguments, and their return value will vary.

There are a few cases where object methods and core methods have the same names, such as `$.each` and `$.fn.each`. In these cases, be extremely careful when reading the documentation that you are exploring the correct method.

A common practice is to use `$(selector)` to obtain collections, and `jQuery` (rather than `$`) to invoke the core methods.

## Lesson 1, Activity 6: Utility Methods

jQuery offers several utility methods in the `$` namespace. These methods are helpful for accomplishing routine programming tasks. Below are examples of a few of the utility methods; for a complete reference on jQuery utility methods, visit <http://api.jquery.com/category/utilities/>.

`$.trim` - removes leading and trailing whitespace

```
$.trim(' lots of extra whitespace ');  
// returns 'lots of extra whitespace'
```

`$.each` - iterates over arrays and objects.

```
$.each([ 'foo', 'bar', 'baz' ], function(idx, val) {  
    console.log('element ' + idx + 'is ' + val);  
});  
  
$.each({ foo : 'bar', baz : 'bim' }, function(k, v) {  
    console.log(k + ' : ' + v);  
});
```

Remember that there is also a method `$.fn.each`, which is used for iterating over a selection of elements.

`$.inArray` - returns a value's index in an array, or -1 if the value is not in the array.

```
var myArray = [ 1, 2, 3, 5 ];  
  
if ($.inArray(4, myArray) !== -1) {  
    console.log('found it!');  
}
```

```
}
```

`$.extend` - changes the properties of the first object using the properties of subsequent objects.

```
var firstObject = { a : 1, foo : 'bar' };
var secondObject = { b : 2, foo : 'baz' };

var newObject = $.extend(firstObject, secondObject);
// The structure of firstObject will now be:
// { a : 1, b : 2, foo : 'baz' }
```

If you don't want to change any of the objects you pass to `$.extend`, pass an empty object as the first argument.

```
var firstObject = { a : 1, foo : 'bar' };
var secondObject = { b : 2, foo : 'baz' };

var newObject = $.extend({}, firstObject, secondObject);
// The structure of firstObject will now be unchanged:
// { a : 1, foo : 'bar' }
// The structure of newObject will now be:
// { a : 1, b : 2, foo : 'baz' }
```

`$.proxy` - returns a function that will always run in the provided scope, that is, sets the meaning of `this` inside the passed function to the second argument.

There are two forms of this method. The first takes two parameters, a function reference and a scope variable. The second takes an object and a string with the name of the method within that object. In this form, the meaning of `this` in the second parameter is locked to the first parameter object.

```

var myFunction = function() { console.log(this); };
var myObject = { foo : 'bar' };

myFunction(); // logs window object

var myProxyFunction = $.proxy(myFunction, myObject);
myProxyFunction(); // logs myObject object

// If you have an object with methods, you can pass the object
// and the name of a method to return a function that will
// always run in the scope of the object.
var myObject = {
  myFn : function() {
    console.log(this);
  }
};

$('#foo').click(myObject.myFn); // logs DOM element #foo
$('#foo').click($.proxy(myObject, 'myFn')); // logs myObject

```

## Checking Types

As mentioned in the "JavaScript Basics" section, jQuery offers a few basic utility methods for determining the type of a specific value.

### Checking the Type of an Arbitrary Value

```

var myValue = [1, 2, 3];

// Using JavaScript's typeof operator to test for primitive types
typeof myValue == 'string';    // false
typeof myValue == 'number';    // false
typeof myValue == 'undefined'; // false
typeof myValue == 'boolean';   // false

// Using strict equality operator to check for null
myValue === null; // false

// Using jQuery's methods to check for non-primitive types
jQuery.isFunction(myValue);    // false
jQuery.isPlainObject(myValue); // false
jQuery.isArray(myValue);       // true

```

## Storing and Retrieving Data Related to an Element

As your work with jQuery progresses, you'll find that there's often data about an element that you want to store with the element. In plain JavaScript, you might do this by adding a property to the DOM element, but you'd have to deal with memory leaks in some browsers. jQuery offers a straightforward way to store data related to an element, and it manages the memory issues for you.

## Using `$.fn.data`

There are several forms of the `$.fn.data` function, which stores data into the elements found in the collection. Possible parameter lists are:

- `key, value` - stores the value under the key for each element in the collection; the value can be any type of JavaScript value
- `object` - stores all the properties of the object as data; basically a bulk form of the above method
- `key` - retrieves the value stored under the key
- *no parameters* - retrieves all the data as a single object with key-value pairs

`$.fn.removeData(key)` will remove the data stored under the key in the collection's elements. If no key is passed, all data will be removed.

```
$('#myDiv').data('myData', { foo : 'bar' });  
$('#myDiv').data('myData'); // returns { foo : 'bar' }
```

You can store any kind of data on an element, and it's hard to overstate the importance of this when you get into complex application development. For the purposes of this class, we'll mostly use `$.fn.data` to store references to other elements.

For example, we may want to establish a relationship between a list item and a `div` that's inside of it. We could establish this relationship every single time we interact with the list item, but a better solution would be to establish the relationship once, and then store a pointer to the `div` in the list item using `$.fn.data`.

It's usually best to store data under single elements, either by using a query that finds only one element, or by using the `$.fn.each` function to establish unique data for each element in a collection.

## Storing a Relationship Between Elements Using `$.fn.data`

```
$('#myList li').each(function() {
  var $li = $(this), $div = $li.find('div.content');
  $li.data('contentDiv', $div);
});

// later, we don't have to find the div again;
// we can just read it from the list item's data
var $firstLi = $('#myList li:first');
$firstLi.data('contentDiv').html('new content');
```

In addition to passing `$.fn.data` a single key-value pair to store data, you can also pass an object containing one or more pairs.

## Data vs Closures

You can use a closure instead of data to remember values, but this approach isn't as flexible. With a closure, the values are held in a variable belonging to an enclosing function, and therefore only visible to that function and any other functions defined within it. Values stored with data are available to any code.

Usually closures are used within an iterating function passed to the



`each()` method, so that the individual instance of the iterating function passed to a specific element can hold the closure for that element. Note that delegating event handlers cannot use closures to remember values, since there will not be a separate function defined for each item that triggers events. We will see delegating event handlers in an upcoming chapter.

```
$('#myList li').each(function() {
  var $li = $(this);
  var $div = $li.find('div.content');
  $li.click(function() {
    // we are within the scope of the function passed to each(),
    // so we can use a closure on the local variable $div
    $div.html('new content');
  });
});
```

## Code Sample:

---

### [jqy-core/Demos/data-vs-closures.html](#)

```
<!doctype HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Data vs Closures</title>
<script src="../../jqy-lib/jquery.js"></script>
<script>
jQuery(document).ready(function() {

  // Using data to remember a paragraph under an h2
  $('#dataDiv>h2').each(function() {
    var $h2 = $(this);
    var $p = $h2.next('p');
    $h2.data('para', $p);
  });

  // Later, from separate code, we don't have to find the p tag again,
  // we can just read it from the h2 element's data
  $('#dataHeading').click(
    function(e) {
      $(this).data('para').html('New Content from Data');
```

```

    }
  );

  // We can also access an element's data from totally unrelated code
  $('#other').click(
    function(e) {
      $('#dataHeading').data('para')
        .html('New Content from unrelated code');
    }
  );

  // Using closure
  $('#closureDiv>h2').each(function() {
    var $h2 = $(this);
    var $p = $h2.next('p');
    // Click handler using a closure of $p to find the paragraph
    $h2.click(
      function(e) {
        $p.html('New Content from Closure');
      }
    );
  });

  // But, there is no way to access that closure from code
  // outside of the each function above.

});
</script>
</head>
<body>
<h1>Data and Closures</h1>
<h2>For "Remembering" Data Associated with an Element</h2>
<hr />
<div id="dataDiv">
  <h2 id="dataHeading">Using Data</h2>
  <p>Click the heading above to see information from data.</p>
</div>
<hr>
<div id="closureDiv">
  <h2>Using Closures</h2>
  <p>Click the heading above to see information from closure.</p>
</div>
<hr>
<div>
  <h3 id="other">Click here to access data from unrelated code.</h3>
</div>
</body>
</html>

```

We locate the first heading in the `div.module` and assign it as data held by the `div`, as well as establishing a click event handler for the `div` that retrieves and uses the data.

We do the same thing for the last heading, using a closure to keep the knowledge within a second click handler on the `div`.

Finally, we create a click handler for the `#other div`, which locates `div.module`, retrieves the data, and uses it to modify the heading text. This shows one benefit of using data instead of a closure - a totally unrelated function can use an element's data.

## Using `$.data`

In addition to the `data` method for jQuery collections, there is also a utility method `$.data(element, key, value)`, which stores data directly into the DOM element passed in (and which therefore can be more efficient). The behavior depends on the parameters passed in:

- `element, key, value` - sets the value into the element's data under the key
- `element, key` - gets the value stored in the element's data under the key
- `element` - gets all the data stored in the element as a single object with key-value pairs

Since iterating through a collection using `$.fn.each` provides DOM elements, not jQuery objects, `$.data` can be effectively used to set unique data for each element.

`$.removeData(element, key)` will remove the data stored

under the key in that element. If no key is passed, all data will be removed.

## DOM-Related Utilities

### **`$.contains(container, contained)`**

The `$.contains` function returns `true` if `contained` is a descendant of `container`.

## Feature & Browser Detection

Although jQuery eliminates most JavaScript browser quirks, there are still occasions when your code needs to know about the browser environment.

jQuery offers the `$.support` object, as well as the deprecated `$.browser` object, for this purpose. For complete documentation on these objects, visit <http://api.jquery.com/jquery.support/> and <http://api.jquery.com/jquery.browser/>.

The `$.support` object is dedicated to determining what features a browser supports; it is recommended as a more "future-proof method of customizing your JavaScript for different browser environments."

The `$.browser` object was deprecated in favor of the `$.support` object, but it will not be removed from jQuery anytime soon. It provides direct detection of the browser brand and version.

## Avoiding Conflicts with Other Libraries

If you are using another JavaScript library that uses the `$` variable, you can run into conflicts with jQuery. In order to avoid these conflicts, you

need to put jQuery in no-conflict mode immediately after it is loaded onto the page and before you attempt to use jQuery in your page.

When you put jQuery into no-conflict mode, you have the option of assigning a variable name to replace \$.

## Putting jQuery into No-Conflict Mode

```
<script src="prototype.js"></script>
<script src="jquery.js"></script>
<script>var $j = jQuery.noConflict();</script>
```

You can continue to use the standard \$ by wrapping your code in a self-executing anonymous function; this is a standard pattern for plugin authoring, where the author cannot know whether another library will have taken over the \$.

## Using the \$ Inside a Self-executing Anonymous Function

```
<script src="prototype.js"></script>
<script src="jquery.js"></script>
<script>
jQuery.noConflict();
(function($) {
// your code here, using the $
})(jQuery);
</script>
```