## Avoiding Conflicts with Other Libraries

The jQuery library and virtually all of its plugins are contained within the jQuery namespace. As a general rule, global objects are stored inside the jQuery namespace as well, so you shouldn't get a clash between jQuery and any other library (like prototype.js, MooTools, or YUI).

That said, there is one caveat: by default, jQuery uses $ as a shortcut for jQuery. Thus, 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.

Putting jQuery Into No-Conflict Mode

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

*<!-- Putting jQuery into no-conflict mode. -->*

*<script src="prototype.js"></script>*

*<script src="jquery.js"></script>*

*<script>*

*var $j = jQuery.noConflict();*

*// $j is now an alias to the jQuery function; creating the new alias is optional.*

*$j(document).ready(function() {*

*$j( "div" ).hide();*

*});*

*// The $ variable now has the prototype meaning, which is a shortcut for*

*// document.getElementById(). mainDiv below is a DOM element, not a jQuery object.*

*window.onload = function() {*

*var mainDiv = $( "main" );*

*}*

*</script>*

In the code above, the $ will revert back to its meaning in original library. You'll still be able to use the full function name jQuery as well as the new alias $j in the rest of your application. The new alias can be named anything you'd like: jq, $J, awesomeQuery, etc.

Finally, if you don't want to define another alternative to the full jQuery function name (you really like to use $ and don't care about using the other library's $ method), then there's still another approach you might try: simply add the $ as an argument passed to your jQuery( document ).ready() function. This is most frequently used in the case where you still want the benefits of really concise jQuery code, but don't want to cause conflicts with other libraries.

*<!-- Another way to put jQuery into no-conflict mode. -->*

*<script src="prototype.js"></script>*

*<script src="jquery.js"></script>*

*<script>*

*jQuery.noConflict();*

*jQuery( document ).ready(function( $ ) {*

*// You can use the locally-scoped $ in here as an alias to jQuery.*

*$( "div" ).hide();*

*});*

*// The $ variable in the global scope has the prototype.js meaning.*

*window.onload = function(){*

*var mainDiv = $( "main" );*

*}*

*</script>*

This is probably the ideal solution for most of your code, considering that there'll be less code that you'll have to change in order to achieve complete compatibility.

Including jQuery Before Other Libraries

The code snippets above rely on jQuery being loaded after prototype.js is loaded. If you include jQuery before other libraries, you may use jQuery when you do some work with jQuery, but the $ will have the meaning defined in the other library. There is no need to relinquish the $ alias by calling jQuery.noConflict().

*<!-- Loading jQuery before other libraries. -->*

*<script src="jquery.js"></script>*

*<script src="prototype.js"></script>*

*<script>*

*// Use full jQuery function name to reference jQuery.*

*jQuery( document ).ready(function() {*

*jQuery( "div" ).hide();*

*});*

*// Use the $ variable as defined in prototype.js*

*window.onload = function() {*

*var mainDiv = $( "main" );*

*};*

*</script>*

## Summary of Ways to Reference the jQuery Function

Here's a recap of ways you can reference the jQuery function when the presence of another library creates a conflict over the use of the $ variable:

Create a New Alias

The jQuery.noConflict() method returns a reference to the jQuery function, so you can capture it in whatever variable you'd like:

*<script src="prototype.js"></script>*

*<script src="jquery.js"></script>*

*<script>*

*// Give $ back to prototype.js; create new alias to jQuery.*

*var $jq = jQuery.noConflict();*

*</script>*

Use an Immediately Invoked Function Expression

You can continue to use the standard $ by wrapping your code in an immediately invoked function expression; this is also a standard pattern for jQuery plugin authoring, where the author cannot know whether another library will have taken over the $. See the Plugins section for more information about writing plugins.

*<!-- Using the $ inside an immediately-invoked function expression. -->*

*<script src="prototype.js"></script>*

*<script src="jquery.js"></script>*

*<script>*

*jQuery.noConflict();*

*(function( $ ) {*

*// Your jQuery code here, using the $*

*})( jQuery );*

*</script>*

Note that if you use this technique, you will not be able to use prototype.js methods inside the immediately invoked function that expect $ to be prototype.js's $.

Use the Argument That's Passed to the jQuery( document ).ready() Function

*<script src="jquery.js"></script>*

*<script src="prototype.js"></script>*

*<script>*

*jQuery(document).ready(function( $ ) {*

*// Your jQuery code here, using $ to refer to jQuery.*

*});*

*</script>*

Or using the more concise syntax for the DOM ready function:

*<script src="jquery.js"></script>*

*<script src="prototype.js"></script>*

*<script>*

*jQuery(function($){*

*// Your jQuery code here, using the $*

*});*

*</script>*

*Manipulating Elements*

*For complete documentation of jQuery manipulation methods, visit the Manipulation documentation on api.jquery.com.*

*Getting and Setting Information About Elements*

*There are many ways to change an existing element. Among the most common tasks is changing the inner HTML or attribute of an element. jQuery offers simple, cross-browser methods for these sorts of manipulations. You can also get information about elements using many of the same methods in their getter incarnations. For more information on getters and setters, see the Working with Selections section. Here are a few methods you can use to get and set information about elements:*

*.html() – Get or set the HTML contents.*

*.text() – Get or set the text contents; HTML will be stripped.*

*.attr() – Get or set the value of the provided attribute.*

*.width() – Get or set the width in pixels of the first element in the selection as an integer.*

*.height() – Get or set the height in pixels of the first element in the selection as an integer.*

*.position() – Get an object with position information for the first element in the selection, relative to its first positioned ancestor. This is a getter only.*

*.val() – Get or set the value of form elements.*

*Moving, Copying, and Removing Elements*

*While there are a variety of ways to move elements around the DOM, there are generally two approaches:*

*Place the selected element(s) relative to another element.*

*Place an element relative to the selected element(s).*

*For example, jQuery provides .insertAfter() and .after(). The .insertAfter() method places the selected element(s) after the element that provided as an argument. The .after() method places the element provided as an argument after the selected element. Several other methods follow this pattern: .insertBefore() and .before(), .appendTo() and .append(), and .prependTo() and .prepend().*

*The method that makes the most sense will depend on what elements are selected, and whether you need to store a reference to the elements you're adding to the page. If you need to store a reference, you will always want to take the first approach – placing the selected elements relative to another element – as it returns the element(s) you're placing. In this case, .insertAfter(), .insertBefore(), .appendTo(), and .prependTo() should be the tools of choice.*

## Cloning Elements

Methods such as .appendTo() move the element, but sometimes a copy of the element is needed instead. In this case, use .clone() first:

*// Making a copy of an element.*

*// Copy the first list item to the end of the list:*

*$( "#myList li:first" ).clone().appendTo( "#myList" );*

*If you need to copy related data and events, be sure to pass true as an argument to .clone().*

Removing Elements

There are two ways to remove elements from the page: .remove() and .detach(). Use .remove() when you want to permanently remove the selection from the page. While .remove() does return the removed element(s), those elements will not have their associated data and events attached to them if you return them to the page.

Use .detach() if you need the data and events to persist. Like .remove(), it returns the selection, but it also maintains the data and events associated with the selection, so you can restore the selection to the page at a later time.

The .detach() method is extremely valuable if you are doing heavy manipulation on an element. In that case, it's beneficial to .detach() the element from the page, work on it in your code, then restore it to the page when you're done. This limits expensive "DOM touches" while maintaining the element's data and events.

If you want to leave the element on the page but remove its contents, you can use .empty() to dispose of the element's inner HTML.

Creating New Elements

jQuery offers a trivial and elegant way to create new elements using the same $() method used to make selections:

*// Creating new elements from an HTML string.*

*$( "<p>This is a new paragraph</p>" );*

*$( "<li class=\"new\">new list item</li>" );*

*// Creating a new element with an attribute object.*

*$( "<a/>", {*

*html: "This is a <strong>new</strong> link",*

*"class": "new",*

*href: "foo.html"*

*});*

Note that the attributes object in the second argument above, the property name class is quoted, although the property names text and href are not. Property names generally do not need to be quoted unless they are reserved words (as class is in this case).

When you create a new element, it is not immediately added to the page. There are several ways to add an element to the page once it's been created.

*// Getting a new element on to the page.*

*var $myNewElement = $( "<p>New element</p>" );*

*$myNewElement.appendTo( "#content" );*

*$myNewElement.insertAfter( "ul:last" ); // This will remove the p from #content!*

*$( "ul" ).last().after( $myNewElement.clone() ); // Clone the p so now we have two.*

The created element doesn't need to be stored in a variable – you can call the method to add the element to the page directly after the $(). However, most of the time you'll want a reference to the element you added so you won't have to select it later.

You can also create an element as you're adding it to the page, but note that in this case you don't get a reference to the newly created element:

*// Creating and adding an element to the page at the same time.*

*$( "ul" ).append( "<li>list item</li>" );*

The syntax for adding new elements to the page is easy, so it's tempting to forget that there's a huge performance cost for adding to the DOM repeatedly. If you're adding many elements to the same container, you'll want to concatenate all the HTML into a single string, and then append that string to the container instead of appending the elements one at a time. Use an array to gather all the pieces together, then join them into a single string for appending:

*var myItems = [];*

*var $myList = $( "#myList" );*

*for ( var i = 0; i < 100; i++ ) {*

*myItems.push( "<li>item " + i + "</li>" );*

*}*

*$myList.append( myItems.join( "" ) );*

# Manipulating Attributes

jQuery's attribute manipulation capabilities are extensive. Basic changes are simple, but the .attr() method also allows for more complex manipulations. It can either set an explicit value, or set a value using the return value of a function. When the function syntax is used, the function receives two arguments: the zero-based index of the element whose attribute is being changed, and the current value of the attribute being changed.

*// Manipulating a single attribute.*

*$( "#myDiv a:first" ).attr( "href", "newDestination.html" );*

*// Manipulating multiple attributes.*

*$( "#myDiv a:first" ).attr({*

*href: "newDestination.html",*

*rel: "super-special"*

*});*

*// Using a function to determine an attribute's new value.*

*$( "#myDiv a:first" ).attr({*

*rel: "super-special",*

*href: function( idx, href ) {*

*return "/new/" + href;*

*}*

*});*

*$( "#myDiv a:first" ).attr( "href", function( idx, href ) {*

*return "/new/" + href;*

*});*

## The jQuery Object

When creating new elements (or selecting existing ones), jQuery returns the elements in a collection. Many developers new to jQuery assume that this collection is an array. It has a zero-indexed sequence of DOM elements, some familiar array functions, and a .length property, after all. Actually, the jQuery object is more complicated than that.

DOM and DOM Elements

The Document Object Model (DOM for short) is a representation of an HTML document. It may contain any number of DOM elements. At a high level, a DOM element can be thought of as a "piece" of a web page. It may contain text and/or other DOM elements. DOM elements are described by a type, such as <div>, <a>, or <p>, and any number of attributes such as src, href, class and so on. For a more thorough description, refer to the official DOM specification from the W3C.

Elements have properties like any JavaScript object. Among these properties are attributes like .tagName and methods like .appendChild(). These properties are the only way to interact with the web page via JavaScript.

The jQuery Object

It turns out that working directly with DOM elements can be awkward. The jQuery object defines many methods to smooth out the experience for developers. Some benefits of the jQuery Object include:

**Compatibility** – The implementation of element methods varies across browser vendors and versions. The following snippet attempts to set the inner HTML of a <tr> element stored in target:

*var target = document.getElementById( "target" );*

*target.innerHTML = "<td>Hello <b>World</b>!</td>";*

This works in many cases, but it will fail in most versions of Internet Explorer. In that case, the recommended approach is to use pure DOM methods instead. By wrapping the target element in a jQuery object, these edge cases are taken care of, and the expected result is achieved in all supported browsers:

*// Setting the inner HTML with jQuery.*

*var target = document.getElementById( "target" );*

*$( target ).html( "<td>Hello <b>World</b>!</td>" );*

**Convenience** – There are also a lot of common DOM manipulation use cases that are awkward to accomplish with pure DOM methods. For instance, inserting an element stored in newElement after the target element requires a rather verbose DOM method:

*// Inserting a new element after another with the native DOM API.*

*var target = document.getElementById( "target" );*

*var newElement = document.createElement( "div" );*

*target.parentNode.insertBefore( target.nextSibling, newElement )*

By wrapping the target element in a jQuery object, the same task becomes much simpler:

*// Inserting a new element after another with jQuery.*

*var target = document.getElementById( "target" );*

*var newElement = document.createElement( "div" );*

*$( target ).after( newElement );*

For the most part, these details are simply "gotchas" standing between you and your goals.

Getting Elements Into the jQuery Object

When the jQuery function is invoked with a CSS selector, it will return a jQuery object wrapping any element(s) that match this selector. For instance, writing:

*// Selecting all <h1> tags.*

*var headings = $( "h1" );*

headings is now a jQuery element containing all the <h1> tags already on the page. This can be verified by inspecting the .length property of headings:

*// Viewing the number of <h1> tags on the page.*

*var allHeadings = $( "h1" );*

*alert( allHeadings.length );*

If the page has more than one <h1> tag, this number will be greater than one. If the page has no <h1> tags, the .length property will be zero. Checking the .length property is a common way to ensure that the selector successfully matched one or more elements.

If the goal is to select only the first heading element, another step is required. There are a number of ways to accomplish this, but the most straight-forward is the .eq() function.

*// Selecting only the first <h1> element on the page (in a jQuery object)*

*var headings = $( "h1" );*

*var firstHeading = headings.eq( 0 );*

Now firstHeading is a jQuery object containing only the first <h1> element on the page. And because firstHeading is a jQuery object, it has useful methods like .html() and .after(). jQuery also has a method named .get() which provides a related function. Instead of returning a jQuery-wrapped DOM element, it returns the DOM element itself.

*// Selecting only the first <h1> element on the page.*

*var firstHeadingElem = $( "h1" ).get( 0 );*

**Alternatively, because the jQuery object is "array-like," it supports array subscripting via brackets:**

*// Selecting only the first <h1> element on the page (alternate approach).*

*var firstHeadingElem = $( "h1" )[ 0 ];*

In either case, firstHeadingElem contains the native DOM element. This means it has DOM properties like .innerHTML and methods like .appendChild(), but not jQuery methods like .html() or .after(). The firstHeadingElem element is more difficult to work with, but there are certain instances that require it. One such instance is making comparisons.

**Not All jQuery Objects are Created ===**

An important detail regarding this "wrapping" behavior is that each wrapped object is unique. This is true even if the object was created with the same selector or contain references to the exact same DOM elements.

*// Creating two jQuery objects for the same element.*

*var logo1 = $( "#logo" );*

*var logo2 = $( "#logo" );*

Although logo1 and logo2 are created in the same way (and wrap the same DOM element), they are not the same object. For example:

*// Comparing jQuery objects.*

*alert( $( "#logo" ) === $( "#logo" ) ); // alerts "false"*

However, both objects contain the same DOM element. The .get() method is useful for testing if two jQuery objects have the same DOM element.

*// Comparing DOM elements.*

*var logo1 = $( "$logo" );*

*var logo1Elem = logo1.get( 0 );*

*var logo2 = $( "#logo" );*

*var logo2Elem = logo2.get( 0 );*

*alert( logo1Elem === logo2Elem ); // alerts "true"*

Many developers prefix a $ to the name of variables that contain jQuery objects in order to help differentiate. There is nothing magic about this practice – it just helps some people keep track of what different variables contain. The previous example could be re-written to follow this convention:

*// Comparing DOM elements (with more readable variable names).*

*var $logo1 = $( "#logo" );*

*var logo1 = $logo1.get( 0 );*

*var $logo2 = $( "#logo" );*

*var logo2 = $logo2.get( 0 );*

*alert( logo1 === logo2 ); // alerts "true"*

This code functions identically to the example above, but it is a little more clear to read.

Regardless of the naming convention used, it is very important to make the distinction between jQuery object and native DOM elements. Native DOM methods and properties are not present on the jQuery object, and vice versa. Error messages like "event.target.closest is not a function"' and "TypeError: Object [object Object] has no method 'setAttribute'" indicate the presence of this common mistake.

jQuery Objects Are Not "Live"

Given a jQuery object with all the paragraph elements on the page:

*// Selecting all <p> elements on the page.*

*var allParagraphs = $( "p" );*

…one might expect that the contents will grow and shrink over time as <p> elements are added and removed from the document. jQuery objects do not behave in this manner. The set of elements contained within a jQuery object will not change unless explicitly modified. This means that the collection is not "live" – it does not automatically update as the document changes. If the document may have changed since the creation the jQuery object, the collection should be updated by creating a new one. It can be as easy as re-running the same selector:

*// Updating the selection.*

*allParagraphs = $( "p" );*

Wrapping Up

Although DOM elements provide all the functionality one needs to create interactive web pages, they can be a hassle to work with. The jQuery object wraps these elements to smooth out this experience and make common tasks easy. When creating or selecting elements with jQuery, the result will always be wrapped in a new jQuery object. If the situation calls for the native DOM elements, they may be accessed through the .get() method and/or array-style subscripting.

## Traversing

Once you've made an initial selection with jQuery, you can traverse deeper into what was just selected. Traversing can be broken down into three basic parts: parents, children, and siblings. jQuery has an abundance of easy-to-use methods for all these parts. Notice that each of these methods can optionally be passed string selectors, and some can also take another jQuery object in order to filter your selection down. Pay attention and refer to the API documentation on traversing to know what variation of arguments you have available.

**Parents**

The methods for finding the parents from a selection include .parent(), .parents(), .parentsUntil(), and .closest().

*<div class="grandparent">*

*<div class="parent">*

*<div class="child">*

*<span class="subchild"></span>*

*</div>*

*</div>*

*<div class="surrogateParent1"></div>*

*<div class="surrogateParent2"></div>*

*</div>*

*// Selecting an element's direct parent:*

*// returns [ div.child ]*

*$( "span.subchild" ).parent();*

*// Selecting all the parents of an element that match a given selector:*

*// returns [ div.parent ]*

*$( "span.subchild" ).parents( "div.parent" );*

*// returns [ div.child, div.parent, div.grandparent ]*

*$( "span.subchild" ).parents();*

*// Selecting all the parents of an element up to, but \*not including\* the selector:*

*// returns [ div.child, div.parent ]*

*$( "span.subchild" ).parentsUntil( "div.grandparent" );*

*// Selecting the closest parent, note that only one parent will be selected*

*// and that the initial element itself is included in the search:*

*// returns [ div.child ]*

*$( "span.subchild" ).closest( "div" );*

*// returns [ div.child ] as the selector is also included in the search:*

*$( "div.child" ).closest( "div" );*

**Children**

The methods for finding child elements from a selection include .children() and .find(). The difference between these methods lies in how far into the child structure the selection is made. .children() only operates on direct child nodes, while .find() can traverse recursively into children, children of those children, and so on.

*// Selecting an element's direct children:*

*// returns [ div.parent, div.surrogateParent1, div.surrogateParent2 ]*

*$( "div.grandparent" ).children( "div" );*

*// Finding all elements within a selection that match the selector:*

*// returns [ div.child, div.parent, div.surrogateParent1, div.surrogateParent2 ]*

*$( "div.grandparent" ).find( "div" );*

**Siblings**

The rest of the traversal methods within jQuery all deal with finding sibling selections. There are a few basic methods as far as the direction of traversal is concerned. You can find previous elements with .prev(), next elements with .next(), and both with .siblings(). There are also a few other methods that build onto these basic methods: .nextAll(), .nextUntil(), .prevAll() and .prevUntil().

*// Selecting a next sibling of the selectors:*

*// returns [ div.surrogateParent1 ]*

*$( "div.parent" ).next();*

*// Selecting a prev sibling of the selectors:*

*// returns [] as No sibling exists before div.parent*

*$( "div.parent" ).prev();*

*// Selecting all the next siblings of the selector:*

*// returns [ div.surrogateParent1, div.surrogateParent2 ]*

*$( "div.parent" ).nextAll();*

*// returns [ div.surrogateParent1 ]*

*$( "div.parent" ).nextAll().first();*

*// returns [ div.surrogateParent2 ]*

*$( "div.parent" ).nextAll().last();*

*// Selecting all the previous siblings of the selector:*

*// returns [ div.surrogateParent1, div.parent ]*

*$( "div.surrogateParent2" ).prevAll();*

*// returns [ div.surrogateParent1 ]*

*$( "div.surrogateParent2" ).prevAll().first();*

*// returns [ div.parent ]*

*$( "div.surrogateParent2" ).prevAll().last();*

**Use .siblings() to select all siblings:**

*// Selecting an element's siblings in both directions that matches the given selector:*

*// returns [ div.surrogateParent1, div.surrogateParent2 ]*

*$( "div.parent" ).siblings();*

*// returns [ div.parent, div.surrogateParent2 ]*

*$( "div.surrogateParent1" ).siblings();*

Be cautious when traversing long distances in documents – complex traversal makes it imperative that the document's structure remain the same, which is difficult to guarantee even if you're the one creating the whole application from server to client. One- or two-step traversal is fine, but it's best to avoid traversals that go from one container to another.

## CSS, Styling, & Dimensions

jQuery includes a handy way to get and set CSS properties of elements:

*// Getting CSS properties.*

*$( "h1" ).css( "fontSize" ); // Returns a string such as "19px".*

*$( "h1" ).css( "font-size" ); // Also works.*

*// Setting CSS properties.*

*$( "h1" ).css( "fontSize", "100px" ); // Setting an individual property.*

*// Setting multiple properties.*

*$( "h1" ).css({*

*fontSize: "100px",*

*color: "red"*

*});*

Note the style of the argument on the second line – it is an object that contains multiple properties. This is a common way to pass multiple arguments to a function, and many jQuery setter methods accept objects to set multiple values at once.

CSS properties that normally include a hyphen need to be camelCased in JavaScript. For example, the CSS property font-size is expressed as fontSize when used as a property name in JavaScript. However, this does not apply when passing the name of a CSS property to the .css() method as a string – in that case, either the camelCased or hyphenated form will work.

It's not recommended to use .css() as a setter in production-ready code, but when passing in an object to set CSS, CSS properties will be camelCased instead of using a hyphen.

**Using CSS Classes for Styling**

As a getter, the .css() method is valuable. However, it should generally be avoided as a setter in production-ready code, because it's generally best to keep presentational information out of JavaScript code. Instead, write CSS rules for classes that describe the various visual states, and then change the class on the element.

*// Working with classes.*

*var $h1 = $( "h1" );*

*$h1.addClass( "big" );*

*$h1.removeClass( "big" );*

*$h1.toggleClass( "big" );*

*if ( $h1.hasClass( "big" ) ) {*

*...*

*}*

Classes can also be useful for storing state information about an element, such as indicating that an element is selected.

**Dimensions**

jQuery offers a variety of methods for obtaining and modifying dimension and position information about an element.

The code below shows a brief overview of the dimensions functionality in jQuery. For complete details about jQuery dimension methods, visit the dimensions documentation on api.jquery.com.

*// Basic dimensions methods.*

*// Sets the width of all <h1> elements.*

*$( "h1" ).width( "50px" );*

*// Gets the width of the first <h1> element.*

*$( "h1" ).width();*

*// Sets the height of all <h1> elements.*

*$( "h1" ).height( "50px" );*

*// Gets the height of the first <h1> element.*

*$( "h1" ).height();*

*// Returns an object containing position information for*

*// the first <h1> relative to its "offset (positioned) parent".*

*$( "h1" ).position();*

## Data Methods

There's often data about an element 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.

*// Storing and retrieving data related to an element.*

*$( "#myDiv" ).data( "keyName", { foo: "bar" } );*

*$( "#myDiv" ).data( "keyName" ); // Returns { foo: "bar" }*

Any kind of data can be stored on an element. For the purposes of this article, .data() will be used to store references to other elements.

For example, you may want to establish a relationship between a list item and a <div> that's inside of it. This relationship could be established every single time the list item is touched, but a better solution would be to establish the relationship once, then store a pointer to the <div> on the list item using .data():

*// Storing a relationship between elements using .data()*

*$( "#myList li" ).each(function() {*

*var $li = $( this );*

*var $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 .data() a single key-value pair to store data, you can also pass an object containing one or more pairs.

## Utility Methods

jQuery offers several utility methods in the $ namespace. These methods are helpful for accomplishing routine programming tasks. For a complete reference on jQuery utility methods, visit the utilities documentation on api.jquery.com.

**Below are examples of a few of the utility methods:**

***$.trim()***

Removes leading and trailing whitespace:

*// Returns "lots of extra whitespace"*

*$.trim( " 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 );*

*});*

The method .each() can be called on a selection to iterate over the elements contained in the selection. .each(), not $.each(), should be used for iterating over elements in a selection.

**$.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 = { foo: "bar", a: "b" };*

*var secondObject = { foo: "baz" };*

*var newObject = $.extend( firstObject, secondObject );*

*console.log( firstObject.foo ); // "baz"*

*console.log( newObject.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 = { foo: "bar", a: "b" };*

*var secondObject = { foo: "baz" };*

*var newObject = $.extend( {}, firstObject, secondObject );*

*console.log( firstObject.foo ); // "bar"*

*console.log( newObject.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.

*var myFunction = function() {*

*console.log( this );*

*};*

*var myObject = {*

*foo: "bar"*

*};*

*myFunction(); // window*

*var myProxyFunction = $.proxy( myFunction, myObject );*

*myProxyFunction(); // myObject*

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 ); // HTMLElement #foo*

*$( "#foo" ).click( $.proxy( myObject, "myFn" ) ); // myObject*

## Iterating over jQuery and non-jQuery Objects

jQuery provides an object iterator utility called $.each() as well as a jQuery collection iterator: .each(). These are not interchangeable. In addition, there are a couple of helpful methods called $.map() and .map() that can shortcut one of our common iteration use cases.

**$.each()**

$.each() is a generic iterator function for looping over object, arrays, and array-like objects. Plain objects are iterated via their named properties while arrays and array-like objects are iterated via their indices.

$.each() is essentially a drop-in replacement of a traditional for or for-in loop. Given:

*var sum = 0;*

*var arr = [ 1, 2, 3, 4, 5 ];*

Then this:

*for ( var i = 0, l = arr.length; i < l; i++ ) {*

*sum += arr[ i ];*

*}*

*console.log( sum ); // 15*

**Can be replaced with this:**

*$.each( arr, function( index, value ){*

*sum += value;*

*});*

*console.log( sum ); // 15*

Notice that we don't have to access arr[ index ] as the value is conveniently passed to the callback in $.each().

In addition, given:

*var sum = 0;*

*var obj = {*

*foo: 1,*

*bar: 2*

*}*

Then this:

*for (var item in obj) {*

*sum += obj[ item ];*

*}*

*console.log( sum ); // 3*

**Can be replaced with this:**

*$.each( obj, function( key, value ) {*

*sum += value;*

*});*

*console.log( sum ); // 3*

Again, we don't have to directly access obj[ key ] as the value is passed directly to the callback.

Note that $.each() is for plain objects, arrays, array-like objects that are not jQuery collections.

**This would be considered incorrect:**

*// Incorrect:*

*$.each( $( "p" ), function() {*

*// Do something*

*});*

**For jQuery collections, use .each().**

.each()

.each() is used directly on a jQuery collection. It iterates over each matched element in the collection and performs a callback on that object. The index of the current element within the collection is passed as an argument to the callback. The value (the DOM element in this case) is also passed, but the callback is fired within the context of the current matched element so the this keyword points to the current element as expected in other jQuery callbacks.

For example, given the following markup:

*<ul>*

*<li><a href="#">Link 1</a></li>*

*<li><a href="#">Link 2</a></li>*

*<li><a href="#">Link 3</a></li>*

*</ul>*

.each() may be used like so:

*$( "li" ).each( function( index, element ){*

*console.log( $( this ).text() );*

*});*

*// Logs the following:*

*// Link 1*

*// Link 2*

*// Link 3*

The Second Argument

The question is often raised, "If this is the element, why is there a second DOM element argument passed to the callback?"

Whether intentional or inadvert, the execution context may change. When consistently using the keyword this, it's easy to end up confusing ourselves or other developers reading the code. Even if the execution context remains the same, it may be more readable to use the second parameter as a named parameter. For example:

*$( "li" ).each( function( index, listItem ) {*

*this === listItem; // true*

*// For example only. You probably shouldn't call $.ajax() in a loop.*

*$.ajax({*

*success: function( data ) {*

*// The context has changed.*

*// The "this" keyword no longer refers to listItem.*

*this !== listItem; // true*

*}*

*});*

*});*

Sometimes .each() Isn't Necessary

Many jQuery methods implicitly iterate over the entire collection, applying their behavior to each matched element. For example, this is unnecessary:

*$( "li" ).each( function( index, el ) {*

*$( el ).addClass( "newClass" );*

*});*

And this is fine:

*$( "li" ).addClass( "newClass" );*

Each <li> in the document will have the class "newClass" added.

On the other hand, some methods do not iterate over the collection. .each() is required when we need to get information from the element before setting a new value.

This will not work:

*// Doesn't work:*

*$( "input" ).val( $( this ).val() + "%" );*

*// .val() does not change the execution context, so this === window*

Rather, this is how it should be written:

*$( "input" ).each( function( i, el ) {*

*var elem = $( el );*

*elem.val( elem.val() + "%" );*

*});*

The following is a list of methods that require .each():

.attr() (getter)

.css() (getter)

.data() (getter)

.height() (getter)

.html() (getter)

.innerHeight()

.innerWidth()

.offset() (getter)

.outerHeight()

.outerWidth()

.position()

.prop() (getter)

.scrollLeft() (getter)

.scrollTop() (getter)

.val() (getter)

.width() (getter)

Note that in most cases, the "getter" signature returns the result from the first element in a jQuery collection while the setter acts over the entire collection of matched elements. The exception to this is .text() where the getter signature will return a concatenated string of text from all matched elements.

In addition to a setter value, the attribute, property, CSS setters, and DOM insertion "setter" methods (i.e. .text() and .html()) accept anonymous callback functions that are applied to each element in the matching set. The arguments passed to the callback are the index of the matched element within the set and the result of the 'getter' signature of the method.

For example, these are equivalent:

*$( "input" ).each( function( i, el ) {*

*var elem = $( el );*

*elem.val( elem.val() + "%" );*

*});*

*$( "input" ).val(function( index, value ) {*

*return value + "%";*

*});*

One other thing to keep in mind with this implicit iteration is that traversal methods such as .children() or .parent() will act on each matched element in a connection, returning a combined collection of all children or parent nodes.

**.map()**

There is a common iteration use case that can be better handled by using the .map() method. Anytime we want to create an array or concatenated string based on all matched elements in our jQuery selector, we're better served using .map().

For example instead of doing this:

*var newArr = [];*

*$( "li" ).each( function() {*

*newArr.push( this.id );*

*});*

We can do this:

*$( "li" ).map( function(index, element) {*

*return this.id;*

*}).get();*

Notice the .get() chained at the end. .map() actually returns a jQuery-wrapped collection, even if we return strings out of the callback. We need to use the argument-less version of .get() in order to return a basic JavaScript array that we can work with. To concatenate into a string, we can chain the plain JS .join() array method after .get().

**$.map**

Like $.each() and .each(), there is a $.map() as well as .map(). The difference is also very similar to both .each() methods. $.map() works on plain JavaScript arrays while .map() works on jQuery element collections. Because it's working on a plain array, $.map() returns a plain array and .get() does not need to be called – in fact, it will throw an error as it's not a native JavaScript method.

A word of warning: $.map() switches the order of callback arguments. This was done in order to match the native JavaScript .map() method made available in ECMAScript 5.

For example:

*<li id="a"></li>*

*<li id="b"></li>*

*<li id="c"></li>*

*<script>*

*var arr = [{*

*id: "a",*

*tagName: "li"*

*}, {*

*id: "b",*

*tagName: "li"*

*}, {*

*id: "c",*

*tagName: "li"*

*}];*

*// Returns [ "a", "b", "c" ]*

*$( "li" ).map( function( index, element ) {*

*return element.id;*

*}).get();*

*// Also returns ["a", "b", "c"]*

*// Note that the value comes first with $.map*

*$.map( arr, function( value, index ) {*

*return value.id;*

*});*

*</script>*

Using jQuery’s .index() Function

.index() is a method on jQuery objects that's generally used to search for a given element within the jQuery object that it's called on. This method has four different signatures with different semantics that can be confusing. This article covers details about how to understand the way .index() works with each signature.

.index() with No Arguments

1

2

3

4

5

6

7

<ul>

<div></div>

<li id="foo1">foo</li>

<li id="bar1">bar</li>

<li id="baz1">baz</li>

<div></div>

</ul>

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

var $foo = $( "#foo1" );

console.log( "Index: " + $foo.index() ); // 1

var $listItem = $( "li" );

// This implicitly calls .last()

console.log( "Index: " + $listItem.index() ); // 3

console.log( "Index: " + $listItem.last().index() ); // 3

var $div = $( "div" );

// This implicitly calls .last()

console.log( "Index: " + $div.index() ); // 4

console.log( "Index: " + $div.last().index() ); // 4

In the first example, .index() gives the zero-based index of #foo1 within its parent. Since #foo1 is the second child of its parent, index() returns 1.

Potential confusion comes from the other examples of .index() in the above code. When .index() is called on a jQuery object that contains more than one element, it does not calculate the index of the first element as might be expected, but instead calculates the index of the last element. This is equivalent to always calling $jqObject.last().index();.

.index() with a String Argument

1

2

3

4

5

6

7

8

<ul>

<div class="test"></div>

<li id="foo1">foo</li>

<li id="bar1" class="test">bar</li>

<li id="baz1">baz</li>

<div class="test"></div>

</ul>

<div id="last"></div>

1

2

3

4

5

6

7

8

9

10

11

12

13

14

var $foo = $( "li" );

// This implicitly calls .first()

console.log( "Index: " + $foo.index( "li" ) ); // 0

console.log( "Index: " + $foo.first().index( "li" ) ); // 0

var $baz = $( "#baz1" );

console.log( "Index: " + $baz.index( "li" )); // 2

var $listItem = $( "#bar1" );

console.log( "Index: " + $listItem.index( ".test" ) ); // 1

var $div = $( "#last" );

console.log( "Index: " + $div.index( "div" ) ); // 2

When .index() is called with a string argument, there are two things to consider. First, jQuery will implicitly call .first() on the original jQuery object. It will be find the index of the first element, not the last element in this case. This is inconsistent, so be careful here.

The second point to consider is that jQuery is querying the entire DOM using the passed in string selector and checking the index within that newly queried jQuery object. For example, when using .index( "div" ) in the last example above, jQuery is selecting all of the <div> elements in the document, then searching for the index that contains the first element in the jQuery object .index() is called on.

.index() with a jQuery Object Argument

1

2

3

4

5

6

7

8

<ul>

<div class="test"></div>

<li id="foo1">foo</li>

<li id="bar1" class="test">bar</li>

<li id="baz1">baz</li>

<div class="test"></div>

</ul>

<div id="last"></div>

1

2

3

4

5

6

7

8

9

10

11

12

var $foo = $( "li" );

var $baz = $( "#baz1" );

console.log( "Index: " + $foo.index( $baz ) ); // 2

var $tests = $( ".test" );

var $bar = $( "#bar1" );

// Implicitly calls .first() on the argument.

console.log( "Index: " + $tests.index( $bar ) ); // 1

console.log( "Index: " + $tests.index( $bar.first() ) ); // 1

In this case, the first element of the jQuery object that is passed into .index() is being checked against all of the elements in the original jQuery object. The original jQuery object, on the left side of .index(), is array-like and is searched from index 0 through length - 1 for the first element of the argument jQuery object.

.index() with a DOM Element Argument

In this case, the DOM element that's passed into .index() is being checked against all of the elements in the original jQuery object. Once all other cases are understood, this should be the simplest case. It is very similar to the previous case, except since the DOM element is passed directly, it is not taken from a jQuery object container.