Chapter 1

Core jQuery

1.1 $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**.
What 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.

1.2 $(document).ready()

A page can't be manipulated safely until the document is "ready".
jQuery detects this state of readiness for you. Code included inside *$(document).ready()* will only run once the page Document Object Model (DOM) is ready for JavaScript code to execute.
Code included inside *$(window).load(function() ...)* will run once the entire page (images or iframes), not just the DOM, is ready.

```
// A $( document ).ready() block.
```

```
$( document ).ready(function() {
console.log( "ready!" );
});
```

You can also pass a named function to *$(document).ready()* instead of passing an anonymous function.

```
// Passing a named function instead of an anonymous function.
function readyFn( jQuery ) {
// Code to run when the document is ready.
}
$( document ).ready( readyFn );
// or:
$( window ).load( readyFn );
```

1.3 Avoiding Conflicts with Other Libraries

1.3.1 Putting jQuery in no-Conflict Mode

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. -->
<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, 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>
```

1.3.2 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>
```

1.4 Attributes

The *.attr()* method acts **as both a getter and a setter**. As a setter, *.attr()* can accept either a key and a value, or an object containing one or more key/value pairs.

.attr() as a setter:

```
$( "a" ).attr( "href", "allMyHrefsAreTheSameNow.html" );
$( "a" ).attr({
title: "all titles are the same too!",
href: "somethingNew.html"
}); \\In this case I've used an object
```

.attr() as a getter:

```
$( "a" ).attr( "href" );// Returns the href for the first a element
\\in the document
```

1.5 Selecting Elements

1.5.1 Selecting Elements by ID

```
$( "#myId" ); // Note IDs must be unique per page.
```

1.5.2 Selecting Elements by Class Name

```
$( ".myClass" );
```

1.5.3 Selecting Elements by Attribute

```
$( "input[name='first_name']" );
// Beware, this can be very slow in older browsers
```

1.5.4 Selecting Elements by Compound CSS Selector

```
$( "#contents ul.people li" );
```

1.5.5 Does My Selection Contain Any Elements?

Once you've made a selection, you'll often want to know whether you have anything to work with. The best way to determine **if there are any elements is to test the selection's *.length* property**, which tells you how many elements were selected. If the answer is 0, the *.length* property will evaluate to false when used as a boolean value:

```
// Testing whether a selection contains elements.
if ( $( "div.foo" ).length ) {
...
}
```

1.5.6 Saving Selections

jQuery doesn't cache elements for you. If you've made a selection that you might need to make again, you should **save the selection in a variable** rather than making the selection repeatedly.

```
var divs = $( "div" );
```

Once the selection is stored in a variable, you can **call jQuery methods on the variable** just like you would have called them on the original selection.

A selection **only fetches the elements that are on the page at the time the selection is made**. If elements are added to the page later, you'll have to repeat the selection or otherwise add them to the selection stored in the variable. Stored selections don't magically update when the DOM changes.

1.5.7 Refining and Filtering Selections

Sometimes the selection contains more than what you're after. jQuery offers **several methods for refining and filtering selections**.

```
// Refining selections.
$( "div.foo" ).has( "p" ); // div.foo elements that contain <p> tags
$( "h1" ).not( ".bar" ); // h1 elements that don't have a class of bar
$( "ul li" ).filter( ".current" ); // unordered list items with class of current
$( "ul li" ).first(); // just the first unordered list item
$( "ul li" ).eq( 5 ); // the sixth
```

1.5.8 Selecting Form Elements

jQuery offers several pseudo-selectors that help **find elements in forms**. These are especially helpful because it can be difficult to distinguish between form elements based on their state or type using standard CSS selectors.

:button Using the *:button* pseudo-selector targets any *<button>* elements and elements with a *type="button"*:

```
$( "form :button" );
```

In order to get the best performance using *:button*, it's best to first select elements with a standard jQuery selector, **then use *.filter(":button")***.
The same concept applies to all selections on form elements.

:checkbox Using the *:checkbox* pseudo-selector targets any *<input>* elements with a *type="checkbox"*:

```
$( "form :checkbox" );
```

:checked Not to be confused with *:checkbox*, *:checked* targets **checked checkboxes**, but keep in mind that this selector works also for **checked radio buttons, and select elements** (for select elements only, use the *:selected* selector):

```
$( "form :checked" );
```

The *:checked* pseudo-selector works when used with checkboxes, radio buttons and selects.

Many More!! Look at the jQuery documentation for all the pseudo-selections available for forms. There are many of them!

1.6 Working with Selections

jQuery "overloads" its methods, so **the method used to set a value generally has the same name as the method used to get a value**. When a method is used to set a value, it's called a **setter**. When a method is used to get (or read) a value, it's called a **getter**. Setters affect all elements in a selection. Getters get the requested value only for the first element in the selection.

```
// The .html() method used as a setter:
$( "h1" ).html( "hello world" );
// The .html() method used as a getter:
$( "h1" ).html();
```

Setters return a jQuery object, allowing you to **continue calling jQuery methods** on your selection. Getters return whatever they were asked to get, so you **can't continue to call jQuery methods** on the value returned by the getter.

```
// Attempting to call a jQuery method after calling a getter.
// This will NOT work:
$( "h1" ).html().addClass( "test" );
```

1.6.1 Chaining

If you call a method on a selection and that method returns a jQuery object, you can continue to call jQuery methods on the object without pausing for a semicolon. **This practice is referred to as "chaining"**:

```
$( "#content" ).find( "h3" ).eq( 2 ).html( "new text for the third h3!" );
```

Chaining is extraordinarily powerful, and it is a feature that many libraries have adapted since it was made popular by jQuery. However, **it must be used with care**, extensive chaining can make code extremely difficult to modify or debug. There is no hard-and-fast rule to how long a chain should be just know that it's easy to get carried away.

1.7 Manipulating Elements

1.7.1 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. 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.

Changing things about elements is trivial, but remember that **the change will affect all elements in the selection**. If you just want to change one element, be sure to **specify that in the selection before calling a setter method**.

```
// Changing the HTML of an element.
$( "#myDiv p:first" ).html( "New <strong>first</strong> paragraph!" );
```

1.7.2 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 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.

```
// Moving elements using different approaches.
// Make the first list item the last list item:
var li = $( "#myList li:first" ).appendTo( "#myList" );
// Another approach to the same problem:
$( "#myList" ).append( $( "#myList li:first" ) );
// Note that there's no way to access the list item
// that we moved, as this returns the list itself.
```

1.7.3 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().

1.7.4 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.

1.7.5 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 html 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" );
// This will remove the p from #content!
myNewElement.insertAfter( "ul:last" );
// Clone the p so now we have two.
$( "ul" ).last().after( myNewElement.clone() );
```

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( "" ) );
```

1.7.6 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: "nofollow"
});

// Using a function to determine an attribute's new value.
$( "#myDiv a:first" ).attr({
rel: "nofollow",
href: function( idx, href ) {
return "/new/" + href;
}
});
$( "#myDiv a:first" ).attr( "href", function( idx, href ) {
return "/new/" + href;
});
```

1.8 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**.

1.8.1 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**.
DOM elements are described by a **type**, such as * *, *<a>*, or *<p>*, and any number of **attributes** such as *src*, *href*, *class* and so on.
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.

1.8.2 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. Example:

```
\\using JavaScript
var target = document.getElementById( "target" );
target.innerHTML = "<td>Hello <b>World</b>!</td>"; \\This may not work with
Internet Explorer

\\using jQuery
var target = document.getElementById( "target" );
$( target ).html( "<td>Hello <b>World</b>!</td>" );
\\Note how the target object is used...
```

Convenience There are also a lot of common DOM manipulation use cases that are awkward to accomplish with pure DOM methods.

```
// Inserting a new element after another with the native DOM API.
var target = document.getElementById( "target" );
var newElement = document.createElement( "div" );
target.parentNode.insertBefore( newElement, target.nextSibling );

// Inserting a new element after another with jQuery.
var target = document.getElementById( "target" );
var newElement = document.createElement( "div" );
$( target ).after( newElement );
```

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**. Example:

```
// Selecting all <h1> tags.
var headings = $( "h1" );
```

headings is now a jQuery element containing all the *<h1>* tags already on the page. We can inspect the *.length* property:

```
var allHeadings = $( "h1" );
alert( allHeadings.length );
```

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" );
// Comparing jQuery objects.
alert( $( "#logo" ) === $( "#logo" ) ); // alerts "false"
```

Although *logo1* and *logo2* are created in the same way (and wrap the same DOM element), they are not the same object.
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"
```

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"

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.

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.

1.9 Traversing

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.

Parent

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 se
// 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" );
```

1.9.1 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" );
```

1.9.2 Siblings

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();
// 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();
```

1.10 CCS, Styling and 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.

1.10.1 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.

1.10.2 Dimensions

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

```
// 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();
```

1.11 Data Methods

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" }
```

For example, you may want to establish a relationship between a list item and a * * that's inside of it.

```
// 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" );
```

1.12 Utility Methods

jQuery offers several utility methods in the $namespace. These methods are helpful for accomplishing routine programming tasks.

$.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 );
});
```

$.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
```

1.13 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.

1.13.1 $.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.
This example:

```
var sum = 0;
var arr = [ 1, 2, 3, 4, 5 ];
for ( var i = 0, l = arr.length; i < l; i++ ) {
sum += arr[ i ];
}
console.log( sum );
```

can be replaced by this

```
var sum = 0;
var arr = [ 1, 2, 3, 4, 5 ];
$.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()*.
Another example:

```
var sum = 0;
var obj = {
foo: 1,
bar: 2
}
for (var item in obj) {
sum += obj[ item ];
}
console.log( sum ); // 3
```

can be replaced by this:

```
var sum = 0;
var obj = {
```

```
foo: 1,
bar: 2
}
$.each( obj, function( key, value ) {
sum += value;
});
console.log( sum ); // 3
```

Note that *$.each()* is for plain objects, arrays, array-like objects that are **not jQuery collections**.
This would be NOT correct:

```
// Incorrect:
$.each( $( "p" ), function() {
// Do something
});
```

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

1.13.2 .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.
Example

```
<ul>
        <li><a href="#">Link 1</a></li>
        <li><a href="#">Link 2</a></li>
        <li><a href="#">Link 3</a></li>
</ul>
$( "li" ).each( function( index, element ){
console.log( $( this ).text() );
});
// Logs the following:
// Link 1
// Link 2
// Link 3
```

1.13.3 Sometimes .each() Isn't Necessary

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

```
$( "li" ).each( function( index, el ) {
$( el ).addClass( "newClass" );
});
```

and this is fine:

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

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.
Look here (http://learn.jquery.com/using-jquery-core/iterating/) for the list of methods which do require *.each()* .

1.13.4 .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()*.
Instead of doing:

```
var newArr = [];
$( "li" ).each( function() {
newArr.push( this.id );
});
```

We can do:

```
$( "li" ).map( function(index, element) {
return this.id;
}).get();
```

Notice the *.get()* chained at the end.

1.13.5 $.map

$.map() works **on plain JavaScript arrays** while *.map()* works on jQuery element collections. Because it's working on a plain array, *$.map()* returns a plain array and *.get()* does not need to be called.
A word of **warning**: *$.map()* switches the order of callback arguments.
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>
```

1.14 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. The four different signatures are:

- *.index()* with No Arguments
- *.index()* with a String Argument
- *.index()* with a jQuery Object Argument
- *.index()* with a DOM Element Argument

Look at the **jQuery API** (`http://api.jquery.com/` to understand the differences.