jQuery Quick Start

August 17, 2014

Contents

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 *<div>*, *<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 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" );
```

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 *<div>* 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.

Chapter 2

jQuery Events

2.1 jQuery Event Basics

jQuery offers convenience methods for most native browser events. These methods - including *.click()*, *.focus()*, *.blur()*, *.change()*, etc. - are shorthand for jQuery's *.on()* method.

```
// Event setup using a convenience method
$( "p" ).click(function() {
  console.log( "You clicked a paragraph!" );
});
// Equivalent event setup using the `.on()` method
$( "p" ).on( "click", function() {
  console.log( "click" );
});
```

2.1.1 Extending Events to New Page Elements

It is important to note that *.on()* can only create event listeners **on elements that exist at the time you set up the listeners**. Similar elements created after the event listeners are established **will not automatically pick up event behaviours** you've set up previously.
Example:

```
$( document ).ready(function(){
  // Sets up click behavior on all button elements with the alert
      class
  // that exist in the DOM when the instruction was executed
  $( "button.alert" ).on( "click", function() {
    console.log( "A button with the alert class was clicked!" );
  });
  // Now create a new button element with the alert class. This
      button
  // was created after the click listeners were applied above, so
      it
  // will not have the same click behavior as its peers
  $( "button" ).addClass( "alert" ).appendTo(document.body );
});
```

2.1.2 Inside the Event Handler Function

Every event handling function receives an **event object**, which contains many properties and methods. The event object is most commonly used to prevent the default action of the event via the *.preventDefault()* method. However, the event object contains a number of **other useful properties and methods**, including:

- **target**: The DOM element that initiated the event.
- **namespace**: The namespace specified when the event was triggered.
- **timeStamp**: The difference in milliseconds between the time the event occurred in the browser and January 1, 1970.
- **preventDefault()**: Prevent the default action of the event (e.g. following a link).
- **stopPropagation()**: Stop the event from bubbling up to other elements.

In addition to the event object, the event handling function also has access to the DOM element that the handler was bound to via the keyword *this*. To turn the DOM element into a jQuery object that we can use jQuery methods on, we simply do *$(this)*, often following this idiom:

```
var element = $( this );
```

Example:

```
// Preventing a link from being followed
$( "a" ).click(function( eventObject ) {
  var elem = $( this );
  if ( elem.attr( "href" ).match( /evil/ ) ) {
    eventObject.preventDefault();
    elem.addClass( "evil" );
  }
});
```

2.1.3 Setting Up Multiple Event Responses

Quite often elements in your application will be bound to **multiple events**. If multiple events are to share the same handling function, you can provide the event types as a space-separated list to *.on()*:

```
// Multiple events, same handler
$( "input" ).on(
  "click change", // bind listeners for multiple events
  function() {
    console.log( "An input was clicked or changed!" )
  }
);
```

When each event has its own handler, you can **pass an object** into *.on()* with one or more **key/value pairs**, with the key being the event name and the value being the function to handle the event.

```
// Binding multiple events with different handlers
$( "p" ).on({
  "click": function() { console.log( "clicked!" ); },
  "mouseover": function() { console.log( "hovered!" ); }
});
```

2.1.4 Namespacing Events

For complex applications and for plugins you share with others, it can be useful to **namespace your events** so you don't unintentionally disconnect events that you didn't or couldn't know about.

```
// Namespacing events
$( "p" ).on( "click.myNamespace", function() { /* ... */ } );
$( "p" ).off( "click.myNamespace" );
$( "p" ).off( ".myNamespace" ); // unbind all events in the
    namespace
```

2.1.5 Tearing Down Event Listeners

To **remove an event listener**, you use the *.off()* method and pass in the event type to off. If you attached a named function to the event, then you can isolate the event tear down to just that named function by passing it as the second argument.

```
// Tearing down all click handlers on a selection
$( "p" ).off( "click" );
// Tearing down a particular click handler, using a reference to
    the function
var foo = function() { console.log( "foo" ); };
var bar = function() { console.log( "bar" ); };
$( "p" ).on( "click", foo ).on( "click", bar );
$( "p" ).off( "click", bar ); // foo is still bound to the click
    event
```

2.1.6 Setting Up Events to Run Only Once

Sometimes you need a particular handler to **run only once**; after that, you may want no handler to run, or you may want a different handler to run. jQuery provides the *.one()* method for this purpose.

```
// Switching handlers using the `$.fn.one` method
$( "p" ).one( "click", firstClick );
function firstClick() {
  console.log( "You just clicked this for the first time!" );
  // Now set up the new handler for subsequent clicks;
  // omit this step if no further click responses are needed
  $( this ).click( function() { console.log( "You have clicked this
      before!" ); } );
}
```

Note that in the code snippet above, the *firstClick* function will be executed **for the first click on each paragraph element** rather than the function being removed from all paragraphs when any paragraph is clicked for the first time.
.one() can also be used to bind **multiple events**:

```
// Using .one() to bind several events
$( "input[id]" ).one( "focus mouseover keydown", firstEvent);
function firstEvent( eventObject ) {
  console.log( "A " + eventObject.type + " event occurred for the
      first time on the input with id " + this.id );
}
```

2.2 Event Helpers

jQuery offers **two event-related helper functions** that save you a few keystrokes.

2.2.1 .hover()

The *.hover()* method lets you pass **one or two functions** to be run when the *mouseenter* and *mouseleave* events occur on an element. If you pass one function, it will be run for both events; if you pass two functions, the first will run for *mouseenter*, and the second will run for *mouseleave*.

```
// The hover helper function
$( "#menu li" ).hover(function() {
  $( this ).toggleClass( "hover" );
});
```

2.2.2 .toggle()

The method is **triggered by the "click" event** and accepts two or more functions. Each time the click event occurs, the next function in the list is called. Generally, *.toggle()* is used with just two functions; however, it will accept an unlimited number of functions. Be careful, though: providing a long list of functions can be difficult to debug.

```
// The toggle helper function
$( "p.expander" ).toggle( function() {
  $( this ).prev().addClass( "open" );
}, function() {
  $( this ).prev().removeClass( "open" );
});
```

2.3 Introducing Events

2.3.1 Introduction

Users perform a countless number of actions such as moving their mice over the page, clicking on elements, and typing in textboxes; all of these are **examples of events**. In addition to these user events, there are a slew of others that occur, like when the page is loaded, when video begins playing or is paused, etc. Whenever something interesting occurs on the page, **an event is fired**, meaning that the browser basically announces that something has happened. It's this announcement that allows developers to "listen" for events and react to them appropriately.

2.3.2 Ways to listen for events

There are many ways to listen for events. Actions are constantly occurring on a webpage, but the developer is only notified about them if they're listening for them. Listening for an event basically means you're waiting for the browser to tell you that a specific event has occurred and then you'll specify how the page should react.
To specify to the browser what to do when an event occurs, you **provide a function**, also known as **an event handler**. This function is executed whenever the event occurs (or until the event is unbound).
Example:

```
// Event binding using a convenience method
$( "#helloBtn" ).click(function( event ) {
  alert( "Hello." );
});
```

The *$(#"helloBtn")* code selects the button element using the *$* (a.k.a. jQuery) function and returns a jQuery object. The jQuery object has a bunch of methods (functions) available to it, one of them named *click*, which resides in the jQuery object's prototype. We call the *click* method on the jQuery object and pass along an anonymous function event handler that's going to be executed when a user clicks the button, alerting "Hello." to the user.
Note using jQuery I don't have to handle the differences between different browsers (e.g. IE before version 9 does not support *addEventLis-*

tener).
There are a **number of ways** that events can be listened for using jQuery:

```
// The many ways to bind events with jQuery
// Attach an event handler directly to the button using jQuery's
// shorthand `click` method.
$( "#helloBtn" ).click(function( event ) {
  alert( "Hello." );
});
// Attach an event handler directly to the button using jQuery's
// `bind` method, passing it an event string of `click`
$( "#helloBtn" ).bind( "click", function( event ) {
  alert( "Hello." );
});
// As of jQuery 1.7, attach an event handler directly to the button
// using jQuery's `on` method.
$( "#helloBtn" ).on( "click", function( event ) {
  alert( "Hello." );
});
// As of jQuery 1.7, attach an event handler to the `body` element that
// is listening for clicks, and will respond whenever *any* button is
// clicked on the page.
$( "body" ).on({
  click: function( event ) {
alert( "Hello." );
}
}, "button" );
// An alternative to the previous example, using slightly different syntax.
$( "body" ).on( "click", "button", function( event ) {
  alert( "Hello." );
});
```

As of jQuery 1.7, **all events are bound via the *on* method**, whether you call it directly or whether you use an alias/shortcut method such as bind or click, which are mapped to the on method internally.
With this in mind, **it's beneficial to use the *on* method** because the others are all just syntactic sugar, and utilizing the on method is going to result in faster and more consistent code.
While examples 1-3 are functionally equivalent, example 4 is different in that the body element is listening for click events that occur on **any button element**, not just *#helloBtn*.
The final example above is exactly the same as the one preceding it, but instead of passing an object, we pass an event string, a selector string, and the callback. Both of these are examples of **event delegation**, a process by which an element higher in the DOM tree listens for events occurring on its children.
Event delegation works because of the notion of **event bubbling**. For most events, whenever something occurs on a page (like an element is clicked), the event travels from the element it occurred on, up to its parent, then up to the parent's parent, and so on, until it reaches the

root element, a.k.a. the *window*.
While event bubbling and delegation work well, **the delegating element should always be as close to the delegatees as possible** so the event doesn't have to travel way up the DOM tree before its handler function is called.
The **two main pros of event delegation** over binding directly to an element (or set of elements) are **performance** and the aforementioned **event bubbling**.
Imagine having a large table of 1,000 cells and binding to an event for each cell. That's 1,000 separate event handlers that the browser has to attach, even if they're all mapped to the same function. Instead of binding to each individual cell though, we could instead use delegation to listen for events that occur on the parent table and react accordingly. One event would be bound instead of 1,000, resulting in way better performance and memory management.

2.3.3 The event object

Consider this example:

```
// Binding a named function
function sayHello( event ) {
  alert( "Hello." );
}
$( "#helloBtn" ).on( "click", sayHello );
```

In this slightly different example, we're defining a **function** called *sayHello* and then passing that function into the on method **instead of an anonymous function**.
So many online examples show anonymous functions used as event handlers, but it's important to realize that you **can also pass defined functions as event handlers** as well. This is important if different elements or different events should perform the same functionality.
What about that ***event* argument** in the *sayHello* function; what is it and why does it matter?
In all DOM event callbacks, **jQuery passes an event object argument** which contains information about the event, such as precisely when and where it occurred, what type of event it was, which element the event occurred on, and a plethora of other information.
Of course **you don't have to call it *event***; you could call it *e* or whatever you want to, but *event* is a pretty common convention.
Using the event object we can prevent the default behaviour and stop propagation in the DOM tree:

```
// Preventing a default action from occurring and stopping the
   event bubbling
$( "form" ).on( "submit", function( event ) {
  // Prevent the form's default submission.
  event.preventDefault();
  // Prevent event from bubbling up DOM tree, prohibiting
      delegation
  event.stopPropagation();
  // Make an AJAX request to submit the form data
});
```

It's also important to note that the event object contains a property called *originalEvent*, which is the event object that **the browser itself created**.
To **inspect the event** itself and see all of the data it contains, you should log the event in the browser's console using console.log. This will allow you to see all of an event's properties (including the *originalEvent*) which can be really helpful for debugging.

```
// Logging an event's information
$( "form" ).on( "submit", function( event ) {
// Prevent the form's default submission.
  event.preventDefault();
// Log the event object for inspectin'
  console.log( event );
// Make an AJAX request to submit the form data
});
```

2.4 Handling Events

jQuery provides a method *.on()* **to respond to any event on the selected elements**. This is called an **event binding**. Although *.on()* isn't the only method provided for event binding, it is a **best practice** to use this for jQuery 1.7+.
The *.on()* method provides **several useful features**:

- Bind any event triggered on the selected elements to an event handler
- Bind multiple events to one event handler
- Bind multiple events and multiple handlers to the selected elements
- Use details about the event in the event handler
- Pass data to the event handler for custom events
- Bind events to elements that will be rendered in the future

2.4.1 Examples

Simple Event Binding

```
// When any <p> tag is clicked, we expect to see '<p> was clicked'
    in the console.
$( "p" ).on( "click", function() {
  console.log( "<p> was clicked" );
});
```

Many events, but only one event handler

Suppose you want to trigger the same event whenever the mouse hovers over or leaves the selected elements. The best practice for this is to use "*mouseenter mouseleave*". Note the difference between this and the next example.

```
// When a user focuses on or changes any input element, we expect a
     console message
// bind to multiple events
$( "div" ).on( "mouseenter mouseleave", function() {
  console.log( "mouse hovered over or left a div" );
});
```

Many events and handlers

Suppose that instead you want **different event handlers** for when the mouse enters and leaves an element. This is more common than the previous example.
.on() accepts an object **containing multiple events and handlers**.

```
$( "div" ).on({
mouseenter: function() {
  console.log( "hovered over a div" );
},
mouseleave: function() {
  console.log( "mouse left a div" );
},
click: function() {
  console.log( "clicked on a div" );
}
});
```

The event object

Handling events can be tricky. It's often helpful to use the **extra information contained in the event object passed to the event handler** for more control. To become familiar with the event object, use this code to inspect it in your browser console after you click on a *<div>* in the page.

```
$( "div" ).on( "click", function( event ) {
  console.log( "event object:" );
  console.dir( event );
});
```

Passing data to the event handler

You can pass your own data to the event object.

```
$( "p" ).on( "click", {
  foo: "bar"
}, function( event ) {
  console.log( "event data: " + event.data.foo + " (should be 'bar
      ')" );
});
```

Binding events to elements that don't exist yet

This is called **event delegation**. Here's an example just for completeness, but see the page on Event Delegation for a full explanation.

```
$( "ul" ).on( "click", "li", function() {
  console.log( "Something in a <ul> was clicked, and we detected
      that it was an <li> element." );
});
```

2.4.2 Connecting Events to Run Only Once

Sometimes you need a particular handler **to run only once**; after that, you may want no handler to run, or you may want a different handler to run.
jQuery provides the *.one()* method for this purpose.

```
// Switching handlers using the `.one()` method
$( "p" ).one( "click", function() {
  console.log( "You just clicked this for the first time!" );
  $( this ).click(function() {
  console.log( "You have clicked this before!" );
  });
});
```

The *.one()* method is especially useful if you need to do some **complicated setup** the first time an element is clicked, but not subsequent times.
.one() accepts the **same arguments as *.on()*** which means it supports multiple events to one or multiple handlers, passing custom data and event delegation.

2.4.3 Disconnecting Events

Although all the fun of jQuery occurs in the *.on()* method, it's counterpart is just as important if you want to be a responsible developer.
.off() **cleans up that event binding when you don't need it anymore**.
Using the *.off()* method diligently is a best practice to ensure that you **only have the event bindings that you need**, when you need them.

```
// Unbinding all click handlers on a selection
$( "p" ).off( "click" );
// Unbinding a particular click handler, using a reference to the
    function
var foo = function() {
  console.log( "foo" );
};
var bar = function() {
  console.log( "bar" );
};
$( "p" ).on( "click", foo ).on( "click", bar );
// foo will stay bound to the click event
$( "p" ).off( "click", bar );
```

2.5 Inside the Event Handling Function

Every event handling function **receives an event object**, which contains **many properties and methods**. The event object is most commonly used to prevent the default action of the event via the *.preventDefault()* method. However, the event object contains a number of other useful properties and methods, including:

- pageX, pageY: The mouse position at the time the event occurred, relative to the top left of the page.
- type: The type of the event (e.g. "click").
- which:The button or key that was pressed.
- data: Any data that was passed in when the event was bound.
- target: The DOM element that initiated the event.
- preventDefault(): Prevent the default action of the event (e.g. following a link).
- stopPropagation(): Stop the event from bubbling up to other elements.

In addition to the event object, the event handling function also **has access to the DOM element that the handler was bound to via the keyword *this***.
To turn the DOM element into a jQuery object that we can use jQuery methods on, we simply do *$(this)*, often following this idiom:

```
var elem = $( this );
```

Another example:

```
// Preventing a link from being followed
$( "a" ).click(function( event ) {
  var elem = $( this );
  if ( elem.attr( "href" ).match( "evil" ) ) {
    event.preventDefault();
    elem.addClass( "evil" );
  }
});
```

2.6 Understanding Event Delagation

Event delegation allows us **to attach a single event listener, to a parent element, that will fire for all descendants matching a selector, whether those descendants exist now or are added in the future**.
In other words Event delegation refers to the process of using event propagation (**bubbling**) to handle events at a higher level in the DOM than the element on which the event originated. It allows us to attach a single event listener for elements that exist **now or in the future**.
Example:

```
<html>
<body>
  <div id="container">
    <ul id="list">
      <li><a href="http://domain1.com">Item #1</a></li>
      <li><a href="/local/path/1">Item #2</a></li>
      <li><a href="/local/path/2">Item #3</a></li>
      <li><a href="http://domain4.com">Item #4</a></li>
    </ul>
  </div>
</body>
</html>
```

When an anchor in our *$list* group is clicked, we want to log its text to the console. Normally we could directly bind to the click event of each anchor using the *.on()* method:

```
// attach a directly bound event
$( "#list a" ).on( "click", function( event ) {
  event.preventDefault();
  console.log( $( this ).text() );
});
```

While this works perfectly fine, there are **drawbacks**. Consider what happens when we add a new anchor after having already bound the above listener:

```
// add a new element on to our existing list
$( "#list" ).append( "<li><a href='http://newdomain.com'>Item #5</a
    ></li>" );
```

If we were to click our newly added item, **nothing would happen**. This is because of the directly bound event handler that we attached previously. Direct events are only attached to elements **at the time the *.on()***

method is called. In this case, since our new anchor did not exist when *.on()* was called, it does not get the event handler.

2.6.1 Event Propagation

Understanding how events propagate is an important factor in being able to leverage Event Delegation. Any time one of our anchor tags is clicked, a click event is fired for that anchor, and then **bubbles up the DOM tree**, triggering each of its parent click event handlers:

- <a>
- <li>
- <ul #list>
- <div #container>
- <body>
- <html>
- document root

This means that anytime you click one of our bound anchor tags, you are effectively clicking the entire document body! This is called **event bubbling or event propagation**.
Since we know how events bubble, we can create a **delegated event**:

```
// attach a delegated event
$( "#list" ).on( "click", "a", function( event ) {
  event.preventDefault();
  console.log( $( this ).text() );
});
```

Notice how we have moved the a part from the selector to the second parameter position of the *.on()* method. This second, selector parameter tells the handler to listen for the specified event, and when it hears it, check to see if the triggering element for that event matches the second parameter. In this case, the triggering event is our anchor tag, which matches that parameter. Since it matches, our anonymous function will execute.
We have now attached a single click event listener to our *<ul>* that **will listen for clicks on its descendant anchors**, instead of attaching an unknown number of directly bound events to the existing anchor tags only.

Using the Triggering Element

What if we wanted to open the link in a new window if that link is an external one (as denoted here by beginning with "http")?

```
// attach a delegated event
$( "#list" ).on( "click", "a", function( event ) {
  var elem = $( this ); // access the DOM element that the handler
      was bound to
  if ( elem.is( "[href^='http']" ) ) {
    elem.attr( "target", "_blank" );
  }
});
```

2.7 Triggering Event Handlers

jQuery provides a way to **trigger the event handlers** bound to an element without any user interaction via the *.trigger()* method.

2.7.1 What handlers can be .trigger()'d?

When an event handler is added using *.on("click", function() ...)*, it can be triggered using jQuery's *.trigger("click")*.
The *.trigger()* function **cannot be used to mimic native browser events**, such as clicking on a file input box or an anchor tag. This is because, there is no event handler attached using jQuery's event system that corresponds to these events.

```
<a href="http://learn.jquery.com">Learn jQuery</a>
```

```
// This will not change the current page
$( "a" ).trigger( "click" );
```

2.7.2 How can I mimic a native browser event, if not .trigger()?

In order **to trigger a native browser event**, you have to use *document.createEventObject* for < IE9 and *document.createEvent* for all other browsers.
The jQuery UI Team created *jquery.simulate.js* in order to **simplify triggering a native browser event** for use in their automated testing. Its usage is modeled after jQuery's trigger.

```
// Triggering a native browser event using the simulate plugin
$( "a" ).simulate( "click" );
```

This will not only trigger the jQuery event handlers, but also follow the link and change the current page.

2.7.3 Don't use .trigger() simply to execute specific functions

While this method has its uses, it should **not be used simply to call a function** that was bound as a click handler.
Instead, you should store the function you want to call in a variable, and pass the variable name when you do your binding. Then, you can call the function itself whenever you want, without the need for *.trigger()*.

```
var foo = function( event ) {
if ( event ) {
  console.log( event );
} else {
  console.log( "this didn't come from an event!" );
  }
};
$( "p" ).on( "click", foo );
foo(); // instead of $( "p" ).trigger( "click" )
```

2.8 Introducing Custom Events

Custom events open up a whole new world of event-driven programming. Instead of focusing on the element that triggers an action, custom events **put the spotlight on the element being acted upon**.
Example: you have a lightbulb in a room in a house. The lightbulb is currently turned on, and it's controlled by two three-way switches and a clapper:

```
<div class="room" id="kitchen">
  <div class="lightbulb on"></div>
  <div class="switch"></div>
  <div class="switch"></div>
  <div class="clapper"></div>
</div>
```

Triggering the clapper or either of the switches will change the state of the lightbulb. The switches and the clapper don't care what state the lightbulb is in; they just want to change the state.
Without custom events, you might write some code like this:

```
$( ".switch, .clapper" ).click(function() {
  var light = $( this ).parent().find( ".lightbulb" );
  if ( light.hasClass( "on" ) ) {
    light.removeClass( "on" ).addClass( "off" );
  } else {
    light.removeClass( "off" ).addClass( "on" );
  }
});
```

With custom events, your code might look more like this:

```
$( ".lightbulb" ).on( "changeState", function( e ) {
  var light = $( this );
  if ( light.hasClass( "on" ) ) {
    light.removeClass( "on" ).addClass( "off" );
  } else {
    light.removeClass( "off" ).addClass( "on" );
  }
});
$( ".switch, .clapper" ).click(function() {
  $( this ).parent().find( ".lightbulb" ).trigger( "changeState" );
});
```

We have moved the behavior of the lightbulb away from the switches and the clapper and to the lightbulb itself.
We'll add another room to our house, along with a master switch, as shown here:

```
<div class="room" id="kitchen">
  <div class="lightbulb on"></div>
  <div class="switch"></div>
  <div class="switch"></div>
  <div class="clapper"></div>
</div>
<div class="room" id="bedroom">
  <div class="lightbulb on"></div>
  <div class="switch"></div>
  <div class="switch"></div>
  <div class="clapper"></div>
</div>
<div id="master_switch"></div>
```

If there are any lights on in the house, we want the master switch to turn all the lights off; otherwise, we want it to turn all lights on. To accomplish this, we'll add two more custom events to the lightbulbs: *turnOn* and *turnOff*.

```
$( ".lightbulb" ).on( "changeState", function( e ) {
  var light = $( this );
  if ( light.hasClass( "on" ) ) {
    light.trigger( "turnOff" );
  } else {
    light.trigger( "turnOn" );
  }
}).on( "turnOn", function( e ) {
  $( this ).removeClass( "off" ).addClass( "on" );
}).on( "turnOff", function( e ) {
  $( this ).removeClass( "on" ).addClass( "off" );
});

$( ".switch, .clapper" ).click(function() {
  $( this ).parent().find( ".lightbulb" ).trigger( "changeState" );
});
$( "#master_switch" ).click(function() {
  if ( $( ".lightbulb.on" ).length ) {
    $( ".lightbulb" ).trigger( "turnOff" );
  } else {
    $( ".lightbulb" ).trigger( "turnOn" );
  }
});
```

Note how the behavior of the master switch is attached to the master switch; the behavior of a lightbulb belongs to the lightbulbs.
If you're accustomed to object-oriented programming, you may find it useful to think of **custom events as methods of objects**.

2.8.1 Conclusion

Custom events offer a new way of thinking about your code: they put the emphasis on the target of a behavior, not on the element that triggers it.
Custom events can enhance code readability and maintainability, by making clear the relationship between an element and its behaviors.

Chapter 3

jQuery UI

jQuery UI is a curated set of user interface interactions, effects, widgets, and themes built on top of the jQuery JavaScript Library.

3.1 Getting Started With jQuery UI

jQuery UI is a widget and interaction library built on top of the jQuery JavaScript Library that you can use to build highly interactive web applications.

3.1.1 Build Your Custom jQuery UI Download

Head over to the Download Builder on the jQuery UI website to download a copy of jQuery UI. jQuery UI's Download Builder allows you to choose the components you would like to download and get a custom version of the library for your project.

3.1.2 Basic Overview: Using jQuery UI on a Web Page

Once the download step is complete, open up *index.html* from the downloaded zip in a text editor. You'll see that it references your theme, jQuery, and jQuery UI. Generally, you'll need to include these three files on any page to use the jQuery UI widgets and interactions:

```
<link rel="stylesheet" href="jquery-ui.min.css">
<script src="external/jquery/jquery.js"></script>
<script src="jquery-ui.min.js"></script>
```

Once you've included the necessary files, you can **add some jQuery widgets to your page**. For example, to make a *datepicker* widget, you'll add a text input element to your page and then call *.datepicker()* on it.

Like this:
HTML

```html
<input type="text" name="date" id="date">
```

JavaScript

```javascript
$( "#date" ).datepicker();
```

3.1.3 Customizing jQuery UI to Your Needs

jQuery UI allows you to customize it in several ways. You've already seen how the **Download Builder** allows you to customize your copy of jQuery UI to include only the portions you want, but there are **additional ways to customize** that code to your implementation.

3.1.4 jQuery UI Basics: Using Options

Each plugin in jQuery UI has a default configuration which is catered to the most basic and common use case. But if you want a plugin to behave different from its default configuration, you can **override each of its default settings using "options"**.
For example, the slider widget has an option for orientation, which allows you to specify whether the slider should be horizontal or vertical. To set this option for a slider on your page, you just pass it in as an argument, like this:

```javascript
$( "#mySliderDiv" ).slider({
  orientation: "vertical"
});
```

You can pass **as many different options as you'd like** by following each one with a comma (except the last one):

```javascript
$( "#mySliderDiv" ).slider({
  orientation: "vertical",
  min: 0,
  max: 150,
  value: 50
});
```

3.1.5 Visual Customization: Designing a jQuery UI Theme

If you want to **design your own theme**, jQuery UI has a very slick application for just that purpose. It's called **ThemeRoller**, and you can always get to it by either clicking the "Themes" link in the jQuery UI navigation, or simply going to jQuery UI ThemeRoller.