EVENT HANDLERS

Web pages are typically built using an event based architecture. An event is something that occurs where we, as the developer, don't have direct control over its timing. Unlike a classic console application, where you provide a list of options to the user, in a time and order that you choose, a web page presents the user with a set of controls, such as buttons and textboxes that the user can typically click around on whenever they see fit. As a result, being able to manage what happens when an event occurs is critical.

Fortunately, in case you hadn't already guessed, jQuery provides a robust API for registering and managing event handlers, which is the code that will be executed when an event is raised. Event handlers are, at the end of the day, simply JavaScript functions.

REGISTERING EVENT HANDLERS

jQuery provides several ways to register an event handler. As we saw in Module 1, the most common way of registering an event handler is to call the function on the jQuery object that corresponds to the event you wish to capture, such as click. You will notice that the [jQuery API](http://api.jquery.com/category/events/) provides functions for almost any JavaScript or DOM event.

We will discuss many of these event handlers as we continue on with the course, but we won't cover every function available - that would take quite a while, and probably bore almost everyone. But we will cover the ones you'll use most frequently. In addition, we'll also cover the various ways you can register an event handler, which will have an impact on the way that you write your code in many scenarios.

## DISCOVERING MORE ABOUT THE EVENT

When you register an event handler from a location other than the object that will be raising the event, there's a disconnect between those two objects. If you're maintaining a 1:1 ratio between event handlers and events, meaning that each event that you're interested in has its own function, then generally there isn't an issue in assuming you know exactly what happened, and on what object it happened to.

However, even if you are maintaining that 1:1 ratio, that can be a brittle relationship. If something changes on the object, your code might not work with the updates that have been made.

In addition, you will often reuse a single event handler for multiple events. The power of the selector syntax supported in jQuery allows you to wire up event handlers for events raised by numerous controls in one line of code. The code below would register the anonymous function with every element decorated with the sample class. The number of elements with that class could change over the course of the page's development, or even the execution of other scripts.

### this

As we've already discussed, when you're inside of an event handler, this is automatically assigned to the object that raised the event.

The most important thing to remember about this is it will be a DOM object, **not** a jQuery object. This means all of the power jQuery provides won't be available directly off this. In order to use jQuery with the object, it needs to be converted to a jQuery object. This is accomplished by simply calling $(this).

The sample below would update the text of the button that was clicked to **Clicked!!**.

<button type="button">Click!</button>

**$**('button').click(function() {

// this is linked to button that was clicked, but is a DOM object

// convert it to a jQuery object by using the jQuery factory

**$**(this).text('Clicked!!');

});

### event object

Beyond just the object that raised the event, jQuery (and JavaScript) also pass an event object to the event handler. This object can be used to determine additional information, such as where the mouse was when the user performed the operation. You might use that information to create hotspots on an element. The [event object](http://api.jquery.com/category/events/event-object/) offers many properties you can query.

// Write out the x/y coordinates of the mouse on click

**$**('button').click(function(e) {

**$**('#output').text(e.pageX + 'x' + e.pageY);

});

$('.sample').click(function() { alert('Hello!'); });

When creating event handlers, it's often best to not make any assumptions about the object that raised the event, even its ID. Not only can things change that might break your code, if you avoid this link between the event handler and the object your code can become both more reusable and flexible.

## COMMON EVENT HANDLERS

jQuery allows you to use functions to "wire up" event handlers to most every JavaScript or DOM event. Here are some of the most common ones you'll use when creating code using jQuery

### click

By far, the most common event handler is click. click, of course, is raised whenever the user clicks on an item, either by using their mouse or tapping on it with their finger. You can register click for nearly any HTML control, not just buttons. This allows you to "convert" other objects, such as div elements, into buttons. You can use this to provide the rich UI you want without being locked into using specific HTML controls, or essentially create your own controls.

To register the click event, simply use the click function.

**$**('#target').click(function() { alert('hello, world!'); });

### dblclick

As the name implies, dblclick is raised when the user double clicks on an element. The dblclick event can be helpful when creating an application that is designed to behave more like a locally installed app. However, if you are going to use the dblclick event, ensure the user knows to double click on items; that's not a common task on a web page.

**$**('#target').dblclick(function() { alert('hello, world!'); });

### blur

The blur event is raised when a form element loses what's known as focus. This simply means the user has clicked elsewhere on the page or has tabbed away from the form control they were on. blur is often used for validation, as you know the user has left the form field.

While this was originally designed for input fields, many modern browsers allow the blur event to be raised for any element type.

To register a handler with the blur event, you can simply call the blur function:

**$**('#target').blur(function() {

// retrieve the value using val

var value = **$**('#target').val();

alert(value);

})

### change

The change event is raised whenever an element is modified. However, the change event **is limited** to input, textarea and select elements only.

Besides just validation, the change event can be used to create interfaces such as cascading dropdown lists. If the list of available values for one dropdown list depends on the selected value of another, change is perfect for that. You would register the event handler on the parent, and when it changes update the child dropdown list.

The CodePen below provides a simple demonstration of cascading dropdown lists. When you select an option on the left, named parent, it will create a custom set of options for the dropdown list on the right, named child based on the selected value.

In the event handler, the logic first calls val on parent to retrieve the selected value. Then empty is called on child to clear out all of the options in the second dropdown. Two append calls are made on child to add the new options. Note the option elements are built by simply using the option tag and adding in a number (1 and 2) followed by the selected value.

Note: We will cover val, parent, empty and append later in this module. I just wanted to give you a functional demonstration of change.

### focus

focus is essentially the opposite of blur. focus is raised when the user clicks, taps, or tabs into a particular control, typically because they want to change its value. You can use focus to provide inline help for a form, such as letting the user know what format you expect for a phone number. This is demonstrated below.

One thing to note is you have the ability to "chain" registrations and other calls when using jQuery. Many jQuery functions return the object you originally accessed (the element with the id phone below). As a result, you can simply call blur right after callling focus.

<!-- sample HTML -->

<div>

<label for="phone">Phone</label>

<input type="text" id="phone" />

<span id="phone-help"></span>

</div>

// sample JavaScript

**$**('#phone').focus(function() {

// Control has focus. Display help

**$**('#phone-help').text('Please enter your phone number as all digits');

}).blur(function() {

// Control lost focus. Clear help

**$**('#phone-help').text('');

});

### mouseenter and mouseleave

mouseenter is raised whenever the user moves their mouse over an element, and mouseleave is raised when the user moves their mouse away from an element. You might use this to provide a tooltip or other contextual information.

<!-- sample HTML -->

<div>

<span id="target">Basic data</span>

<span id="target-help"></span>

</div>

**$**('#target').mouseenter(function() {

**$**('#target-help').text('More data');

}).mouseleave(function() {

**$**('#target-help').text('');

});

### hover

The hover event is logically equivalent to both the mouseenter and mouseleave events. What makes hover special is you can actually register two event handlers in one call. The first parameter hover accepts is for mouseenter, or when the hover begins, and the second parameter is for mouseout.

The code below is syntactically equivalent to the sample above.

**$**('#target').hover(function() {

// mouseenter

**$**('#target-help').text('More data');

}, function() {

// mouseleave

**$**('#target-help').text('');

});

## MODIFYING ELEMENTS

You've always had the ability to modify the DOM using JavaScript. Where jQuery comes into play is in how it eases the process. As we've seen, jQuery makes it easier to select items, and handles cross browser support for us. This allows us to focus more attention on building the UI for our users, and implement the necessary logic, and less time on how to make the page behave the way we want.

In this section, we'll talk about some of the functions jQuery offers for accessing and replacing the values of elements, attributes and CSS. Some of this was introduced in Module 1, but there are a few new features we'll introduce here.

### val

If you wish to retrieve the value of an input control, all you have to do is call val().

var value = **$**('#some-input-control').val();

If you wish to set the value, simply pass the new value in as a parameter to the val function. If you wish to set a textbox to a blank value, simply use an empty string.

// Empty a textbox

**$**('#some-textbox').val('');

### html and text

The two most basic functions for working with the content of an element are text and html. Both functions will return the entire contents of the element (if called without a parameter), or replace the entire contents with the parameter you provide. The difference between the two is encoding.

If you use text, all special HTML characters, such as < and >, will be replaced with their encoded equivalents, such as &< and &<. This can be helpful when working with data from an untrusted source, as it will not be written out to the page as HTML.

// write text out to the screen

// this will be displayed as the literal

// **value**

// the text value will not be bolded

**$**('#output').text('**value**');

If you ran the code above, the user would see: <strong>value</strong>.

html, however, will return the contents and display the new contents as HTML

// write text out to the screen

// this will be displayed as value

// with the word bolded

**$**('#output').html('**value**');

The above JavaScript will display as **value**.

## WORKING WITH ATTRIBUTES

To work with and manipulate attributes, jQuery provides the attr function. Among the many overloads of the function, the two most common are the ones that take one or two parameters.

If you need to retrieve the value of an attribute, you can do this by simply calling attr(name).

To change the value of an attribute, simply call attr(name, newValue).

Even special attributes such as id, name and class are attributes, and can be both accessed and updated using the exact same syntax. However, certain attributes, like class and style have functions all of their own.

## WORKING WITH CLASSES

When you're updating a UI the simplest, and most common, way of making changes is through CSS classes. To assist you, jQuery provides addClass and removeClass. As you might expect, addClass will add the class you provide as a parameter, and removeClass will remove it.

One nice thing about removeClass is it will not throw an error if the class doesn't exist. As a result, if you know you don't want a class, but you don't know if it's currently being applied, you can simply call removeClass knowing the code won't fail.

## WORKING WITH CSS

As a general rule, if you want to adjust the style of an element the best way to do this is through classes. By working with classes, you gain the ability to reuse that style in other places. In addition, you separate the style from the code, which allows you to easily change the style as needed without having to update your JavaScript.

But you may still find yourself needing to modify the CSS of an element directly. If this is the case, you could manipulate an item's style by simply using attr.

**$**('#target').attr('style', 'color: red');

That would work, but it's not ideal. Remember that attr is going to replace the style attribute, not update it, so any existing styles would be lost. Trying to add or remove styles using attr becomes a challenge.

Setting styles is simply a matter of setting a bunch of property/value pairs. Because of that fact, jQuery exposes an item's style as these pairs through a special function, css.

css will parse an items style property as the set of key value pairs, allowing you to both read and write each property individually.

If you want to retrieve the value of a CSS property, you call css(property).

// Retrieve an item's color

var color = **$**('#target').css('color');

if you wish to set a property, call css(property, newValue)

// Change an item's color to red

**$**('#target').css('color', 'red');

If you want to set multiple properties, you can create a JavaScript object with the property/value pairs, and then pass the object into the css function. You can create the object like you would a normal JavaScript object.

The catch, of course, is properties like background-color. Obviously, JavaScript can't work with a hyphen in a name. You could create an object with that property by either using camel casing like { backgroundColor: 'red' }, or set the property names as strings like {'background-color': 'red' }. Either way works just fine.

The end result of the JavaScript below would be an element with red text and a yellow background.

var style = {

color: 'red',

backgroundColor: yellow

};

**$**('#target').css(style);

## DYNAMIC EVENT HANDLERS

Chances are, when you're creating event handlers, you'll use the function that matches the event the vast majority of the time. After all, it's straight forward to type out, and later read,$('selector').click(function() {}); when you're comfortable with jQuery. But this approach does have only drawback: the only event you're going to register with that syntax is click. What happens if you want to be able to register the same event handler for multiple events? Or what about a scenario where you need to determine the event based on something else that happens on the screen - maybe changing mouseenter to click?

With on and off, you can register and unregister event handlers, respectively, while passing in the name of the event, or events, as a string.

### on and off

Both on and off share the same syntax:

$('selector').('events', 'selector (optional)', function)

1. events: A space separated list of the events you wish to register, such as 'click' or 'click mouseenter'
2. selector (optional): An optional selector. **We will discuss this in the next unit**
3. function: The function you wish to call. As per usual, this can be either a regular function or, typically, an anonymous function.

// the following two code blocks are semantically the same

**$**('#demo').on('click', function() { alert('hello!!'); });

**$**('#demo').click(function() { alert('hello!!'); });

### Which should you use?

Often the choice between using click(function() {}) or on('click', function() {}) is a matter of personal preference. For me I'll use the function click, or whichever one matches the event, whenever possible. This is both because of the Intellisense offered in Visual Studio (or other auto-complete technology in other editors) and readability. Obviously if you need to dynamically choose the event, or if you need multiple events for the same event handler, using on is your only choice. In addition, on offers one more feature, known as delegation, which we'll explore on the next page.

## DELEGATION AND DYNAMIC ELEMENTS

One of the concepts we'll explore later is power jQuery offers developers for modifying the DOM programmatically. You can easily add buttons, div elements, etc., as needed.

The question then becomes, what would happen in the following scenario?

<button>Click</button>

<div id="placeholder"></div>

**$**(function() {

// document.ready (on load)

// register a click event handler with all button elements

**$**('button').click(function() { alert('hello'); });

// create a new button

**$**('#placeholder').html('<button>New button</button>');

});

When the JavaScript code executes, a new button will be created **after** we registered an event handler for all buttons. Does the new button get the same click event handler?

### Introducing delegate

In order for click, or, in some cases, on, to work, the element must already be on the page. As a result, if you registered an event handler, and then later created a new element that would match the selector you used, the event handler wouldn't apply.

Enter delegate. Registering an event handler using delegate, as demonstrated above, is similar to on, with one major difference: New elements will automatically have the event handler applied.

Using the above example, if we updated the click event handler registration to the new code below, both the existing button and new button would have the same event handler.

**$**(document).delegate('button', 'click', function() { alert('hello'); });

The delegate syntax is $(selector).delegate(selector, events, eventHandler)

1. $(selector): In order to use delegate, the jQuery factory must return a parent of the object you wish to wire up.
2. selector: From the parent, provide a selector to retrieve the necessary elements.
3. events: Same events string as before.
4. eventHandler: Same event handler as before.

One important note about using delegate is you are **required** to start at a parent object, rather than at the object itself. For example, in the scenario proposed above, we want to register the same click event handler for all buttons. Normally the selector would be 'button'. But, delegate requires we start at a parent, and then the selector we provide as the first parameter will locate the elements from that parent. Since we want all buttons, we can use document, which will give us the entire document.

### Delegating event handlers with on

With jQuery 1.7, delegate is superseded by on. You may have noticed in the prior unit on had a selector parameter as well. If you do not provide that selector, on behaves like a normal event handler registration. But, if you do provide a selector, it will use delegation to register the event handlers.

**One important thing to note** is the order of parameters for on and delegate. With on, you list the events first and the selector second. With delegate, it's selector followed by events.

// Delegation (note the order of parameters)

**$**(document).on('click', 'button', function() {alert('hello'); });

// Semantically the same as above

// (note the order of parameters)

**$**(document).delegate('button', 'click', function() { alert('hello'); });

## SINGLE EXECUTION

When you register an event handler with an object, that event handler will remain with that object until you unregister it, which you could do with a function such as off.

However, what if you know in advance you only want the event to execute once? For instance, if you're creating a button that will submit a form, you only want the user to click the button once. Wouldn't it be great if you could tell jQuery to only use the event handler one time?

Fortunately, you can!

### one

one shares a similar syntax with on, only there is no delegation option. You simply provide the name of the event, and the event handler.

<button id="single">This only works once</button>

<div id="output"></div>

**$**(function() {

**$**('#single').one('click', function() {

**$**('#output').text('You clicked on the button');

});

});

(note: while there is an overload of one that allows for a selector, it will only filter which children the event applies to. It will not enable delegation.)

## TRIGGERING EVENTS

Typically events are raised by the user performing the action themselves. But there may be scenarios where you want to raise the event programmatically. For example, you may want to allow a user to click on a button to refresh data, or perform the operation on your own through code. jQuery allows you to raise events through one of three methods, either the registration method, such as click, or one of two trigger methods: trigger or triggerHandler.

### Helper method

The easiest method to use to trigger an event is to simply call the registration function without any parameters. For example, $('#demo').mouseenter() will automatically perform the mouseenter event for the element with an id of demo.

### trigger and triggerHandler

Using a similar syntax to on, trigger and triggerHandler allow you to provide the name of the event as a parameter. The difference between the two is trigger will execute for all elements in the collection, while triggerHandler only executes the handler for the first element.

<button type="button" id="first">First button</button>

<button type="button" id="Second">Second button</button>

<button type="button" id="trigger">trigger</button>

<button type="button" id="trigger-handler">triggerHandler</button>

**$**(function() {

**$**('button').click(function() {

// display the id of the button

alert(this.id);

});

**$**('#trigger').click(function() {

// Would alert every button's id

// including the last two

**$**('button').trigger('click');

});

// Would alert "first"

**$**('#trigger-handler').click(function() {

// Would alert "first"

**$**('button').triggerHandler('click');

});

});

## MANIPULATING THE DOM

As we've already seen, html and text can be used to update the page. The problem with those functions, however, is those functions replace the contents of the element they're called on, rather than modifying the content. You could, I suppose, keep appending text, but after a while your code will fall under its own weight.

jQuery offers several functions to easily manipulate the DOM.

### Internal DOM manipulation

If you wish to update the contents of an element by adding or removing elements, you can use append, appendTo, prepend and prependTo. Remember that the element you target will become the **container** for the new element. So using append will add a new element **inside** the target, not after the element. (There's another function for that.)

### prepend and prependTo

prepend and prependTo add new content to the **beginning** of the contents of a target. The difference between them is the order in which the parameters and target are passed in. The lines of code below are semantically identical.

// prepend is called on the target, and accepts the new content as the parameter

**$**('#target').prepend('<div>New content</div>');

// prependTo is called on the new content, and accepts the target as the parameter

**$**('<div>New content</div>').prependTo('#target');

If you had the following starting HTML

<div id="target">

<div>existing content</div>

</div>

And executed either of the lines of JavaScript above, the result would be as follows.

<div id="target">

<div>New content</div>

<div>existing content</div>

</div>

### append and appendTo

append and appendTo add new content to the **end** of the contents of a target. The difference between them is the order in which the parameters and target are passed in. The lines of code below are semantically identical.

// append is called on the target, and accepts the new content as the parameter

**$**('#target').append('<div>New content</div>');

// appendTo is called on the new content, and accepts the target as the parameter

**$**('<div>New content</div>').appendTo('#target');

If you had the following starting HTML

<div id="target">

<div>existing content</div>

</div>

And executed either of the lines of JavaScript above, the result would be as follows.

<div id="target">

<div>existing content</div>

<div>New content</div>

</div>

## EXTERNAL DOM MANIPULATION

Functions such as prepend add new content inside of the target, using the target as a container for the new content. If you wish to add new content before or after an existing element, jQuery provides after, insertAfter, before, and insertBefore.

### after and insertAfter

after and insertAfter both add new content after the target, on the same level of the hierarchy, rather than inside the target. The difference between the two functions is the order of parameters and target.

// after is called on the target, and accepts the new content as a parameter

**$**('target').after('New content');

// insertAfter is called on the new content, and accepts the target as a parameter

**$**('New content').insertAfter('#target');

If you had the following HTML as the starting point:

<div id="target">

<div>existing content</div>

</div>

... calling either of the lines of JavaScript above would produce the following result:

<div id="target">

<div>existing content</div>

</div>

<div>New content</div>

### before and insertBefore

before and insertBefore both add new content after the target, on the same level of the hierarchy, rather than inside the target. The difference between the two functions is the order of parameters and target.

// before is called on the target, and accepts the new content as a parameter

**$**('target').before('New content');

// insertBefore is called on the new content, and accepts the target as a parameter

**$**('New content').insertBefore('#target');

If you had the following HTML as the starting point:

<div id="target">

<div>existing content</div>

</div>

... calling either of the lines of JavaScript above would produce the following result:

<div>New content</div>

<div id="target">

<div>existing content</div>

</div>

## WRAPPING EXISTING CONTENT WITH NEW CONTENT

There will be times you'll need to manipulate the DOM by surrounding existing content with a new element. jQuery provides several wrapping functions to perform that type of an operation.

### wrapping functions

For the wrap and wrapAll demonstrations below, we'll be using this HTML as the starting point:

<div id="target">

<div class="demo">Item one</div>

<div class="demo">Item two</div>

</div>

### wrap

The wrap function wraps each item with the element passed into the function. As a result, if you called:

**$**('#target').wrap('<section></section>');

the result would be:

<section>

<div id="target">

<div class="demo">Item one</div>

<div class="demo">Item two</div>

</div>

</section>

Where things get a little more interesting is if you used a selector that returned multiple items. Imagine the following JavaScript:

**$**('.demo').wrap('<section></section>');

The result would be:

<div id="target">

<section>

<div class="demo">Item one</div>

</section>

<section>

<div class="demo">Item two</div>

</section>

</div>

Notice that wrap will wrap each element returned by the selector with the new element.

### wrapAll

wrapAll behaves differently. Rather than wrapping each returned element, wrapAll wraps all returned content with one new element. As a result, the JavaScript

**$**('.demo').wrapAll('<section></section>');

would result in

<div id="target">

<section>

<div class="demo">Item one</div>

<div class="demo">Item two</div>

</section>

</div>

### wrapInner

wrapInner is different from both wrap and wrapAll in that wrapInner operates on the children of the target, rather than on the target itself. If you started with the following HTML...

<div id="target">

<div>Item one</div>

<div>Item two</div>

</div>

...and executed the JavaScript below...

**$**('#target').wrapInner('<section></section>');

...the result would be:

<div id="target">

<section>

<div>Item one</div>

<div>Item two</div>

</section>

</div>

## ANIMATIONS

Up until now, whenever we've manipulated the DOM the results have just appeared. While this is just fine, it can sometimes be a bit jarring to the user. It would certainly be nice if we could have just a little animation, to either make our page look more professional, or to draw the user's attention to the new or updated content.

### hide, show and toggle

jQuery offers many animations. The first two are hide and show. As you might suspect, hide causes something to disappear from the screen, while show causes something to appear. toggle, on the other hand, determines the current status of the item in question and changes it. So if the item was currently hidden, toggle would call show, and if the item was visible, toggle would call hide.

Behind the scenes, jQuery controls visibility by using the display property in CSS. When you hide an item, jQuery will store the current value of display (such as inline or block) and then set the display to none. When you call show, jQuery resets the display CSS property back to the value it had stored before.

hide, show and toggle all take several parameters, including a set of options that allow you to control things like "easing", which is how the animation is going to take place. Most commonly the only parameter you will provide is the duration you wish the animation to take place over, in milliseconds.

<div id="target">Show or hide</div>

<button type="button" id="btn-toggle">Toggle</button>

**$**(function() {

**$**('#btn-toggle').click(function() {

// animation will take one second

**$**('#target').toggle(1000);

});

});

One final note about this, and all animations. Animations in jQuery return a promise object, which can be used to run code after an animation completes. We'll discuss promises in Module 3.

## ANIMATIONS IN ACTION

In the prior demonstrations and videos, we saw how to use show, hide and toggle to easily modify the visibility of an item, while also adding a little bit of animation to the UI. While this is certainly a neat trick, there's a very good chance you'll want to start a UI with a section already hidden.

Imagine the following scenario:

<form>

<div>

<label>Name:</label>

<input type="text" />

</div>

<div>

<label>Additional information:</label>

<input type="text" />

</div>

</form>

In the above form, we want to prompt the user for their name, and additional information. But what happens if that additional information is optional? It would be nice to provide some form of a button, checkbox, or otherwise, to enable the user to display the additional information section, rather than always displaying it.

Or, imagine if you were building a page where you had additional details, maybe a user agreement that many people will ignore. Displaying that on the page takes up unnecessary real estate.

In both of those scenarios, it'd be great if we could hide the item, and only if the user indicates they want to see the extra data should we show it to them.

We can accomplish this by starting with the following HTML, updated from before:

<style>

.hidden { display: none; }

</style>

<form>

<div>

<label>Name:</label>

<input type="text" />

</div>

<button type="button" id="show-additional-information">

Show additional information

</button>

<div id="additional-information" class="hidden">

<label>Additional information:</label>

<input type="text" />

</div>

</form>

...and then added the following script

**$**(function() {

**$**('#show-additional-information').click(function() {

**$**('#additional-information').show(750);

});

});

You'll notice that we're setting the div element with additional-information as its ID to display:none to start. The display property is what show will modify. We're then setting up the click event handler to call show, which will then cause the item to display.

## FADING

Both show and hide have a vertical animation. The new content sort of "rolls" onto the screen, or "rolls" up from the screen. If you don't want that additional animation, and just want the item to slowly appear or disappear, you can use fading.

### fadeIn, fadeOut and fadeToggle

Like their show/hide counterparts, fadeIn and fadeOut will display and hide an item respectively. And, like toggle, fadeToggle will reverse the current state of the item.

The fading functions perform their work by both modifying the CSS display property, and animates the item by modifying its opacity. If you want an item to start as hidden by default, and display it using fadeIn, you only need to set display: none; there is no need to modify the starting opacity.

Like all animation functions, the fading functions accept various parameters with the most common being the length of time (in milliseconds). Also, the fading functions return a promise, which we'll explore in Module 3.

## SLIDING

Unlike the other animations, the sliding functions cause the entire item to either slide down or slide up.

Keeping with the conventions we've already seen, slideDown will cause an item to appear, slideUp will cause it to disappear, and slideToggle will cause change its state.

The sliding functions perform their work by both modifying the CSS display property, and animates the item by modifying its position. If you want an item to start as hidden by default, and display it using slideDown, you only need to set display: none; there is no need to modify the starting position.

Like all animation functions, the sliding functions accept various parameters with the most common being the length of time (in milliseconds). Also, the sliding functions return a promise, which we'll explore in Module 3.

## REMOVING AND REPLACING ITEMS

Besides just creating new content, there will be times when you want to remove existing content, or replace content with new content. As you might have guessed, jQuery offers you this power as well.

### remove and empty

Both remove and empty completely delete items from the DOM. The difference between the two is what they delete. In the case of remove, it will delete the item you target, while empty deletes the contents of the item you target.

Consider the following starting HTML

<div id="target">

<div>Some cool content</div>

<div>Some more cool content</div>

</div>

If you called $('#target').remove(), the resulting HTML would be, well, nothing. The entire contents of the sample are removed.

Contrast that with calling $('#target').empty(). empty deletes the contents of the target, so the resulting HTML would be as follows

<div id="target">

</div>

### replaceAll and replaceWith

If you wish to replace existing content with new content, jQuery offers replaceAll and replaceWith. In both cases, you'll provide what existing content you wish to replace, and what new content you wish to use. The difference between the two is the order in which you pass the existing and new content.

If you had the following starting HTML...

<div id="target">

<div>Some cool content</div>

<div>Some more cool content</div>

</div>

...and you wanted to finish with the following HTML...

<div>NEW content</div>

...then either method below would work.

// replaceWith replaces the content on the left with the new content in the parameter

**$**('#target').replaceWith('<div>NEW content</div>');

// replaceAll replaces the target in the parameter with the content on the left

**$**('<div>NEW content</div>').replaceAll('#target');

**NOTE** When using either replaceAll or replaceWith, the entire element is replaced, **not** just the contents of the element. Use both functions with caution!

## CLONE

The main reason to use jQuery is because it makes "day to day" programming in JavaScript much easier, offering functionality and capabilities common to other programming environments. Because the basic operations become that much easier, it allows you to focus your attention on the cooler features you wish to add to your pages, adding interactivity, and making your page seem more like a locally installed application.

This is where the humble function clone comes into play. clone allows you to make a copy of jQuery objects.

### clone in the real world

There are several scenarios in which clone simplifies content creation. Let's say you were building a page to allow an administrator to create email addresses and passwords. You'd like to ensure the administrator can create as many accounts as they would like on one page, and be able to send up all of the information to the server in one round trip, rather than using Ajax calls or submitting the form for each account.

This requires the ability to dynamically add labels and textboxes. You could do that this way

<button type="button" id="add-line">Add new line</button>

<div id="container">

<div class="user-entry">

<label>Email:</label>

<input type="email" />

<label>Password:</label>

<input type="password" />

</div>

</div>

**$**(function() {

**$**('#add-line').click(function() {

**$**('#container').append('<div class="user-entry"> <label>Email:</label>

<input type="email" />

<label>Password:</label>

<input type="password" />

</div>');

});

});

It would work. But you've got a large string literal in JavaScript, which can be tough to debug when you got to make changes. In addition, whatever tool you're using to create your page isn't going to be able to offer you any support for the HTML that's inside of the string literal. And finally, we have the same markup twice, both in the starting HTML and in the JavaScript. (Granted, we could clean that up by adding the first line in through JavaScript, but that's not really the best solution here.)

A better way to solve this problem would be to clone the starting HTML that makes up the user information form, and then use that clone to add the new content. (The HTML below hasn't changed from above, it's copied for readability.)

<button type="button" id="add-line">Add new line</button>

<div id="container">

<div class="user-entry">

<label>Email:</label>

<input type="email" />

<label>Password:</label>

<input type="password" />

</div>

</div>

**$**(function() {

**$**userForm = **$**('.user-entry').clone;

**$**('#add-line').click(function() {

**$**('#container').append(**$**userForm.clone());

});

});

Let's break down the code just a bit here. For starters, you'll notice on line 2 that we are grabbing a clone of the item with the class of user-entry. This contains the form for inputting the user. You'll also notice we use a variable that starts with $. This is common in jQuery to indicate the object is a jQuery object.

On line 4, you'll notice we use the $userForm in append to add the markup (or jQuery object) we copied earlier. What you'll also notice is we're calling clone() a second time. The reason for this is how JavaScript passes parameters. When you're working with an object, JavaScript passes a reference to the object, not a copy of the object. This would mean that we've added in a pointer to the clone we created earlier, **not** a brand new copy. The end result of this is if we called append a second time, we'd be trying to add **the exact same object** into the container again, **not** a brand new copy. By calling clone() again on line 4, we pass in a copy of our user form, rather than a pointer to the one we already used.

Below is a Pen with the code. You can use this to play around with the JavaScript, and see the impact removing clone has.

## CLONING AND ANIMATIONS

Now that we know how to clone items to simplify creation of dynamic content, and how to animate them to add some additional punch to our pages, let's see how we could bring it all together.

In the cloning section, we had the following demo HTML:

<button type="button" id="add-line">Add new line</button>

<div id="container">

<div class="user-entry">

<label>Email:</label>

<input type="email" />

<label>Password:</label>

<input type="password" />

</div>

</div>

and the following JavaScript:

**$**(function() {

**$**userForm = **$**('.user-entry').clone();

**$**('#add-line').click(function() {

**$**('#container').append(**$**userForm.clone());

});

});

Let's modify our code to provide animation to the new elements. We'll start making sure our cloned item is hidden by using the css function. Because $userForm is a jQuery object, manipulate it just like any other jQuery object.

// Insert after line 2

**$**userForm.css('display', 'none');

When we call append, we know that the new item is going to be the last element in the container. We can access container's children by using children, and access the last one by calling last.

We also know that the item is hidden, because we set it to be hidden. If we call show, it will then display, with the animation. We can accomplish this by cloning it into a variable, adding the new variable, and then calling show on the new variable.

// Replace line 4

var newUserForm = **$**userForm.clone();

**$**('#container').append(**$**newUserForm);

newUserForm.show(750);

Our final JavaScript becomes:

**$**(function() {

**$**userForm = **$**('.user-entry').clone();

**$**userForm.css('display', 'none');

**$**('#add-line').click(function() {

var newUserForm = **$**userForm.clone();

**$**('#container').append(**$**newUserForm);

newUserForm.show(1000);

});

});