**Part 11. Custom Events: Part 1**

**Introducing Custom Events**

We’re all familiar with the basic events — click, mouseover, focus, blur, submit, etc. — that we can latch on to as a user interacts with the browser. Custom events open up a whole new world of event-driven programming. In this section, we’ll use jQuery’s custom events system to make a simple Twitter search application.

It can be difficult at first to understand why you'd want to use custom events, when the built-in events seem to suit your needs just fine. It turns out that custom events offer a whole new way of thinking about event-driven JavaScript. Instead of focusing on the element that triggers an action, custom events put the spotlight on the element being acted upon. This brings a bevy of benefits, including:

* Behaviors of the target element can easily be triggered by different elements using the same code.
* Behaviors can be triggered across multiple, similar, target elements at once.
* Behaviors are more clearly associated with the target element in code, making code easier to read and maintain.

Why should you care? An example is probably the best way to explain. Suppose 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').bind('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');

});

This last bit of code is not that exciting, but something important has happened: we’ve moved the behavior of the lightbulb to the lightbulb, and away from the switches and the clapper.

Let’s make our example a little more interesting. 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. We’ll make use of them in the changeState custom event, and use some logic to decide which one the master switch should trigger:

$('.lightbulb')

.bind('changeState', function(e) {

var $light = $(this);

if ($light.hasClass('on')) {

$light.trigger('turnOff');

} else {

$light.trigger('turnOn');

}

})

.bind('turnOn', function(e) {

$(this).removeClass('off').addClass('on');

})

.bind('turnOff', function(e) {

$(this).removeClass('off').addClass('on');

});

$('.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.

**Note**

If you’re accustomed to object-oriented programming, you may find it useful to think of custom events as methods of objects. Loosely speaking, the object to which the method belongs is created via the jQuery selector. Binding the changeState custom event to all $(‘.light’) elements is akin to having a class called Light with a method of changeState, and then instantiating new Light objects for each element with a classname of light.

**Recap: $.fn.bind and $.fn.trigger**

In the world of custom events, there are two important jQuery methods: $.fn.bind and $.fn.trigger. In the Events chapter, we saw how to use these methods for working with user events; for this chapter, it's important to remember two things:

* The $.fn.bind method takes an event type and an event handling function as arguments. Optionally, it can also receive event-related data as its second argument, pushing the event handling function to the third argument. Any data that is passed will be available to the event handling function in the data property of the event object. The event handling function always receives the event object as its first argument.
* The $.fn.trigger method takes an event type as its argument. Optionally, it can also take an array of values. These values will be passed to the event handling function as arguments after the event object.

Here is an example of the usage of $.fn.bind and $.fn.trigger that uses custom data in both cases:

$(document).bind('myCustomEvent', { foo : 'bar' }, function(e, arg1, arg2) {

console.log(e.data.foo); // 'bar'

console.log(arg1); // 'bim'

console.log(arg2); // 'baz'

});

$(document).trigger('myCustomEvent', [ 'bim', 'baz' ]);