**Writing Stateful Plugins with the jQuery UI Widget Factory: Part 3**

**Adding Callbacks**

One of the easiest ways to make your plugin extensible is to add callbacks so users can react when the state of your plugin changes. We can see below how to add a callback to our progress bar to signify when the progress has reached 100%. The \_trigger method takes three parameters: the name of the callback, a native event object that initiated the callback, and a hash of data relevant to the event. The callback name is the only required parameter, but the others can be very useful for users who want to implement custom functionality on top of your plugin. For example, if we were building a draggable plugin, we could pass the native mousemove event when triggering a drag callback; this would allow users to react to the drag based on the x/y coordinates provided by the event object.

**Example 8.9. Providing callbacks for user extension**

$.widget("nmk.progressbar", {

options: {

value: 0

},

\_create: function() {

this.element.addClass("progressbar");

this.\_update();

},

\_setOption: function(key, value) {

this.options[key] = value;

this.\_update();

},

\_update: function() {

var progress = this.options.value + "%";

this.element.text(progress);

if (this.options.value == 100) {

this.\_trigger("complete", null, { value: 100 });

}

}

});

Callback functions are essentially just additional options, so you can get and set them just like any other option. Whenever a callback is executed, a corresponding event is triggered as well. The event type is determined by concatenating the plugin name and the callback name. The callback and event both receive the same two parameters: an event object and a hash of data relevant to the event, as we’ll see below.

If your plugin has functionality that you want to allow the user to prevent, the best way to support this is by creating cancelable callbacks. Users can cancel a callback, or its associated event, the same way they cancel any native event: by calling event.preventDefault() or using return false. If the user cancels the callback, the \_trigger method will return false so you can implement the appropriate functionality within your plugin.

**Example 8.10. Binding to widget events**

var bar = $("<div></div>")

.appendTo("body")

.progressbar({

complete: function(event, data) {

alert( "Callbacks are great!" );

}

})

.bind("progressbarcomplete", function(event, data) {

alert("Events bubble and support many handlers for extreme flexibility.");

alert("The progress bar value is " + data.value);

});

bar.progressbar("option", "value", 100);

**The Widget Factory: Under the Hood**

When you call jQuery.widget, it creates a constructor function for your plugin and sets the object literal that you pass in as the prototype for your plugin instances. All of the functionality that automatically gets added to your plugin comes from a base widget prototype, which is defined as jQuery.Widget.prototype. When a plugin instance is created, it is stored on the original DOM element using jQuery.data, with the plugin name as the key.

Because the plugin instance is directly linked to the DOM element, you can access the plugin instance directly instead of going through the exposed plugin method if you want. This will allow you to call methods directly on the plugin instance instead of passing method names as strings and will also give you direct access to the plugin’s properties.

var bar = $("<div></div>")

.appendTo("body")

.progressbar()

.data("progressbar" );

// call a method directly on the plugin instance

bar.option("value", 50);

// access properties on the plugin instance

alert(bar.options.value);

One of the biggest benefits of having a constructor and prototype for a plugin is the ease of extending the plugin. By adding or modifying methods on the plugin’s prototype, we can modify the behavior of all instances of our plugin. For example, if we wanted to add a method to our progress bar to reset the progress to 0% we could add this method to the prototype and it would instantly be available to be called on any plugin instance.

$.nmk.progressbar.prototype.reset = function() {

this.\_setOption("value", 0);

};