

# **jQuery Fundamentals Training**

**Custom Events**

Lesson 1, Activity 2: 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 lesson, we'll use jQuery's custom events system to make a simple form designing application.

## About Custom Events

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.
- The event source does not need to know in detail how to tell the target element what to do -- it merely needs to tell the target that the event occurred, and the target can then decide what to do

Why should you care? An example is probably the best way to explain. Suppose you have a form designer application using Ajax. Users can click on a button in a dialog box to add an element, like a button, to the interface they are designing.

**Without custom events**, you could use one of two approaches:

1. The code for the interface you are designing (or just the page as a whole) could register with the button for the event. Problems with this approach include:
  - There are probably multiple buttons in the dialog for different elements or purposes, so the code would need to register for all of them
  - If the dialog box is changed, perhaps to use a dropdown combo box with a *change* event instead of a button click event, the event-registering code will need to be changed.
2. The code in the dialog could listen for the click event, and invoke related methods on the interface.
  - There is one main drawback here -- the dialog code will need to know specific details about the interface (like what method to call and what data to pass to it).

## Examples Without Using Custom Events

Without custom events, you might write some code like this, where the dialog box code would invoke a specific function in the interface.

### Code Sample:

---

[jqy-custom-events/Demos/formdesigner-push.html](http://jqy-custom-events/Demos/formdesigner-push.html)

```
FormDesigner - Push Strategy

h4 { text-align: center; border-bottom: 2px solid black; }
.left { text-align: left; }
#ui {
position:absolute;
top:100px;
right:10px;
height:400px;
width:500px;
border:5px ridge #000077;
background-color:#CCCCff;
```

```
padding:24px;
}
#dialog {
position:absolute;
top:100px;
left:10px;
height:200px;
width:500px;
border:5px ridge blue;
background-color:#CCCCCC;
padding:24px;
}
table.dialog {
width:100%;
}
```

In this example, the designer dialog pushes information to the UI object by invoking its methods.

This is the UI.

This is the dialog.

Name:	<input type="text"/>	Label:	<input type="text"/>
<input type="button" value="Add"/>			

This demo uses a push strategy, where the dialog pushes information to the UI by invoking its `addTextBox` method. The main issue here is that the dialog must know the method to call in the affected object (`ui`), and what parameters to pass. Defining the situation is the responsibility of the affected object, and it's up to the dialog to adhere to that. In OOP terms, the dialog "must know the semantics of the UI". Also, if there were multiple objects interested in the event, the dialog would need to invoke each one's appropriate function.

### Code Sample:

---

[jqy-custom-events/Demos/formdesigner-pull.html](http://jqy-custom-events/Demos/formdesigner-pull.html)

```
FormDesigner - Pull Strategy
```

```
h4 { text-align: center; border-bottom: 2px solid black; }
.left { text-align: left; }
#ui {
position:absolute;
top:100px;
right:10px;
height:400px;
width:500px;
border:5px ridge #000077;
background-color:#CCCCff;
padding:24px;
}
#dialog {
position:absolute;
top:100px;
left:10px;
height:200px;
width:500px;
border:5px ridge blue;
background-color:#CCCCCC;
```

```
padding:24px;  
}  
table.dialog {  
width:100%;  
}
```

In this example, the UI dialog registers for low-level events with the UI object.

This is the UI.

This is the dialog.

### Add Element

Name:  Label:

This demo uses a pull strategy, where the UI pulls information from the dialog when the event occurs. Here the situation is reversed -- now the UI must know the semantics of the dialog. Also, if there were multiple objects interested in the event, each one would have fairly complicated code to register for event notifications.

## Examples Using Custom Events

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

### Code Sample:

---

[jqy-custom-events/Demos/formdesigner-custom-event.html](http://jqy-custom-events/Demos/formdesigner-custom-event.html)

```
FormDesigner - Custom Events
```

```
h4 { text-align: center; border-bottom: 2px solid black; }
.left { text-align: left; }
#ui {
position:absolute;
top:100px;
right:10px;
height:400px;
width:500px;
border:5px ridge #000077;
background-color:#CCCCff;
padding:24px;
}
#dialog {
position:absolute;
top:100px;
left:10px;
height:200px;
width:500px;
border:5px ridge blue;
background-color:#CCCCCC;
padding:24px;
}
table.dialog {
width:100%;
}
```

In this example, the designer dialog fires a custom event, which the UI registers for.

This is the UI.

This is the dialog.

### Add Element

Name:  Label:

Here, the dialog just fires off an event to any interested listener. Defining the data is now the responsibility of the dialog, and the handling object can register to listen for the events, and unpack the data provided by the dialog when the event occurs.

### Recap: `$.fn.bind`, `$.fn.trigger`, and `$.fn.triggerHandler`.

In the world of custom events, there are three important jQuery methods: `$.fn.bind`, `$.fn.trigger`, and `$.fn.triggerHandler`. In the Events lesson, we saw how to use the `bind` method for working with user events. We also saw that we could trigger events using a convenience method. We could have triggered the events using `trigger` or `triggerHandler`, but, for browser events usually the convenience method is easiest. For this chapter, it's important to remember a few things:

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



event handling function as individual arguments after the event object.

- Native events, like 'click', that are fired using trigger will bubble up the DOM tree.
- Events triggered using this method will invoke a default action, but that action can be canceled with `e.preventDefault()`.
- This method operates on all elements in the collection it is invoked upon.

3. `$.fn.triggerHandler` is similar in terms of parameters, but different in terms of bubbling and default action.

- Native events that are fired using `triggerHandler` will not bubble up the DOM tree.
- Events triggered using this method will not invoke a default action.
- This method operates only on the first element in the collection it is invoked upon.

## Lesson 1, Activity 4: Custom Events

Duration: 20 to 30 minutes.

1. We would like to add a checkbox capability to our form designer.
2. The HTML is already in place, and we have changed the form element names.
3. You will need to obtain the values for the new text fields in the dialog.
4. Then fire a new custom event for adding a checkbox.
5. Create a handler for the new event.
6. Bind your handler to the dialog.

### Solution:

---

<jqy-custom-events/Solutions/formdesigner-custom-event.html>

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>FormDesigner - Custom Events</title>
<style>
h4 { text-align: center; border-bottom: 2px solid black; }
.left { text-align: left; }
#ui {
  position: absolute;
  top: 100px;
  right: 10px;
  height: 400px;
  width: 500px;
  border: 5px ridge #000077;
  background-color: #CCCCff;
  padding: 24px;
}
#dialog {
  position: absolute;
  top: 100px;
  left: 10px;
  height: 200px;
  width: 600px;
  border: 5px ridge blue;
  background-color: #CCCCCC;
  padding: 24px;
}
table.dialog {
  width: 100%;
}
</style>
```

```

<script src="../../jqy-lib/jquery.js"></script>
<script>
$(document).ready(function() {
  var $dialog = $('#dialog');
  var $ui = $('#ui');
  $dialog.tbLabel = $dialog.find('[name=tbLabel]');
  $dialog.tbName = $dialog.find('[name=tbName]');
  // get the values associated with adding a check box
  $dialog.cbLabel = $dialog.find('[name=cbLabel]');
  $dialog.cbName = $dialog.find('[name=cbName]');
  $dialog.cbValue = $dialog.find('[name=cbValue]');

  $dialog.find('[name=addTB]').click(
    function(e) {
      $dialog.triggerHandler(
        "addTextBox",
        {
          label: $dialog.tbLabel.val(), // values from other elements
          name: $dialog.tbName.val()    // in the dialog
        }
      );
    }
  );
  // bind a click handler for the checkbox Add button
  $dialog.find('[name=addCB]').click(
    function(e) {
      $dialog.triggerHandler(
        "addCheckBox",
        {
          label: $dialog.cbLabel.val(), // values from other elements
          name: $dialog.cbName.val(),   // in the dialog
          value: $dialog.cbValue.val()
        }
      );
    }
  );

  // in the ui div
  $ui.addTextBox = function(e, data) {
    $ui.children('form').append($ (
      '<label>' + data.label +
      '<input type="text" name="' + data.name + '" /></label><br />'));
  };
  $dialog.bind(
    "addTextBox",      // event name we made up
    $ui.addTextBox     // function in the ui
  );
  // define an addCheckBox function for ui
  $ui.addCheckBox = function(e, data) {
    $ui.children('form').append($ (
      '<label>' + data.label +

```

```

    '<input type="checkbox" name="' + data.name +
    ' value="' + data.value + '"/></label><br />');
    alert(
        '<label>' + data.label +
        '<input type="checkbox" name="' + data.name +
        ' value="' + data.value + '"/></label><br />');
};
// bind it as a handler for an addCheckBox event
$dialog.bind(
    "addCheckBox",    // event name we made up
    $ui.addCheckBox   // function in the ui
);

});
</script>
</head>
<body>
<h3>In this example, the designer dialog fires a custom event, which the UI registers
for.</h3>
<div id="ui">
    <h4>This is the UI.</h4>
    <form action="javascript:void 0;"></form>
</div>
<div id="dialog">
    <h4>This is the dialog.</h4>
    <form action="javascript:void 0;">
        <table class="dialog">
            <tbody>
                <tr>
                    <th colspan="6" class="left">Add Text Box </th>
                </tr>
                <tr>
                    <td>Name:</td><td><input type="text" name="tbName"></td>
                    <td>Label:</td><td colspan="3"><input type="text" name="tbLabel"></td>
                </tr>
                <tr>
                    <th colspan="6"><input type="button" name="addTB" value="    Add    "></th>
                </tr>
            </tbody>
            <tbody>
                <tr>
                    <th colspan="6" class="left">Add Check Box </th>
                </tr>
                <tr>
                    <td>Name:</td><td><input type="text" name="cbName"></td>
                    <td>Label:</td><td><input type="text" name="cbLabel"></td>
                    <td>Value:</td><td><input type="text" name="cbValue"></td>
                </tr>
                <tr>
                    <th colspan="6"><input type="button" name="addCB" value="    Add    "></th>
                </tr>
            </tbody>
        </table>
    </form>
</div>

```

```
    </tbody>
  </table>
</form>
</div>
</body>
</html>
```

You should be able to follow the new logic from the comments in the code.

## Lesson 1, Activity 6: **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. If you take the time at the outset to spell out the pieces of your application, as well as the behaviors those pieces need to exhibit, custom events can provide a powerful way for you to "talk" to those pieces, either one at a time or en masse. Once the behaviors of a piece have been described, it becomes trivial to trigger those behaviors from anywhere, allowing for rapid creation of and experimentation with interface options. Finally, custom events can enhance code readability and maintainability, by making clear the relationship between an element and its behaviors.